Appeared in Weihai Yu & al. (eds.): *Norsk Informatikk-konferanse 2001*, Tapir, Trondheim Norway 2001, pp. 176--185.

Some Statistical Performance Estimation Techniques for Dynamic Machines*

Magne Haveraaen Hogne Hundvebakke
Institutt for Informatikk, Universitetet i Bergen,
P.O.Box 7800, N-5020 Bergen, Norway

Abstract

The advent of computers with a very dynamic run-time behaviour, such as the SGI Origin 2000 series, makes it very difficult to asses the runtime behaviour of programs in order to compare efficiency of implementations. Here we empirically investigate some simple statistical measures for estimating CPU usage under such circumstances, ending up with a very simple and seemingly accurate estimation technique: the minimum run-time of 3-5 executions of the program when the machine is lightly loaded.

1 Introduction

When working with computer software for scientific applications, the run-time efficiency of programs becomes very important. Such applications may execute for several hours on a super-computer, and even a small percentage increase in efficiency may yield significant time savings. Efficiency increases may come from better algorithms, hardware improvements, or from better adaptation of software to hardware characteristics, such as memory access patterns. The two latter groups of improvements can only be quantified by measuring actual software performance on the machines. Even though algorithm quality may be assessed using theoretical comparisons of order of magnitude scalability, when comparing two algorithms on the same data, it is the actual performance on a computer which counts. Thus good techniques for measuring the run-time efficiency of a program on a computer is needed.

The traditional measure for run-time efficiency of computer programs has been to execute the program under controlled circumstances, measuring certain characteristics of its run-time behaviour. Typically one would set the machine in single user mode, and, e.g., measure the CPU usage of the program. Scalability of a program on a computer could be assessed by running the same program for various data sizes, the comparison of two algorithms could be achieved by running the two programs on the same computer and comparing their run-times, or one could benchmark the efficiency of computer systems by running the same program on the

*This research has received support from The Research Council of Norway, including a grant of computing time from the "Programme for Supercomputing".

different computers (see, e.g., [Meu00]). Performance analysis and benchmarking is an area by itself, see the classic book [Jai91], which describe a broad range of statistical techniques available for benchmarking. Some techniques for assessing run-time based on static properties of the code [DRW92] and predicting scalability from multivariate analysis of run-time properties [LKL95] have been proposed.

The development of complex processors with instruction prefetch, dynamic execution ordering of instructions, the ability to handle several outstanding memory accesses, and complex memory hierarchies, such as on the SGI Origin 2000 series of machines, implies that the run-time characteristics of programs under seemingly controlled situations may vary considerably from one execution to the next. Although this problem is well-known to practitioners, how to handle performance measurements does not seem to be systematically addressed. In many cases the mean running time of a few sample runs is taken to be the performance measure, although [Jai91] suggests the *median* can be used. There seems to be no study on the accuracy of these practices, nor on the use of other statistical estimators for finding the run-time of programs under such dynamic circumstances. Some recent work use multi-variate regression analysis for the comparison of memory hierarchy effects on program execution times [SC00, SHCL01]. This gives good information, but requires extensive analysis. Here we are looking for a simple measure for estimating run-times, allowing for straight forward comparisons of programs and machines. We will restrict ourselves to sequential executions in this study.

This paper is organised as follows. First we discuss some assumptions on run-time measurements and our benchmark program. Then we present several estimation techniques and discuss these. Section 4 concludes and discusses possible further work.

2 Benchmark

For benchmark purposes one would assume that a computer program under optimal circumstances achieves a certain maximal speed on a given hardware platform. Any variations in circumstances will possibly reduce the speed of the program. Such variations can be small delays for accessing data on disc, memory misses due to paging, swapping of processes due to time sharing, temporary stopping of large jobs due to high processor load, etc. Many of these circumstances, especially the latter ones which relate to the run-time regime of the computer, will dramatically increase the wall clock time of a program. However, one would not expect the same effect on CPU time measurements, as these should be largely insensitive to a program being suspended during high load and reactivated when load decreases. Some effect on CPU time is expected, as many small waits due to paging and swapping, e.g., may be considered as active waiting by the CPU timing utilities. Thus we will expect variation in the CPU run-time to be linked to machine load.

The SeisMod suite of programs [HFJ99] was chosen as our benchmark program to find a good run-time measuring technique. SeisMod is a collection of seismic wave simulation programs, with specialised versions created for various problem instances. It represents the current, more abstraction (object) oriented style, of writing programs than traditional programs like LINPACK which often is used for such purposes [Don01] We decided to use the sequential, standard isotropic, stan-

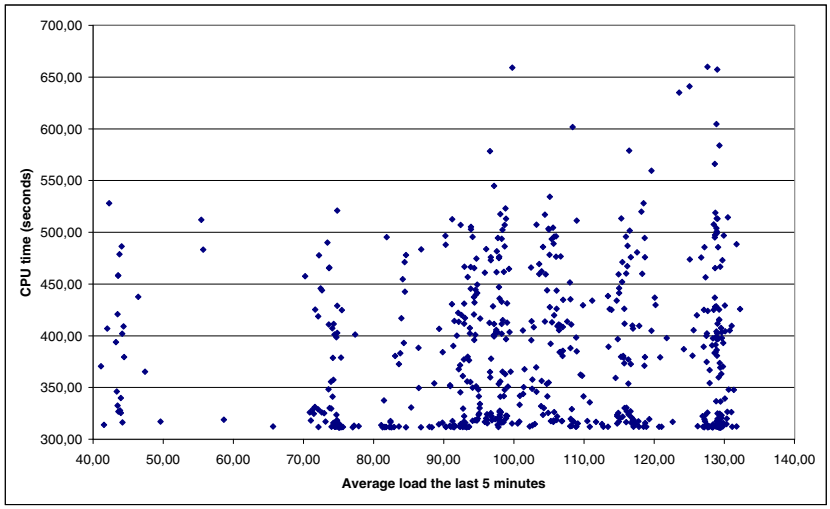

Figure 1: CPU run-time versus load plotted for 600 observations.

dard elastic, with sea surface, version of SeisMod. Using a small data set this program gives a distribution of CPU run-time mostly between 5 and 10 minutes, see Figure 1. The measurements were taken on the SGI Origin 2800/128 with 128 processors and 64GB of RAM (`embla.ntnu.no`) at NTNU in Trondheim, Norway, in January 2001, during the normal operational regime of the machine. The experiments were performed by running batches of 100 program executions submitted to queues with various priorities administrated by LSF. Between each program execution there was a random wait varying between 0 and 590 seconds. Load, defined as the average number of jobs in the run queue, was given by the unix command `uptime`. We used the average for the last 5 minutes, taken immediately after each program execution. Some of the test batches ran simultaneously.

The average CPU time used by an execution in Figure 1 is 381 seconds, with a standard variation of 68 seconds (18%) for 600 observations. Looking at this figure 4 cases seem discernible:

- A *low load* situation where the load factor is below 80. This sample contained 87 observations, the average is 344 seconds with a standard deviation of 54 seconds (16%).

- A transition load situation with load in the interval from 80 to 90 containing 36 observations.

- A *high load* situation with load in the interval from 90 to 125. Here the average is 387 seconds with a standard deviation of 75 seconds (22%) for 338 observations.

- A regime cutoff situation when load is above 125 with 135 observations. Here long-running, low priority jobs are suspended, so we get an artificial clustering effect. Clustering effects may also appear for other load situations due to different priority batch queues having different load cutoffs, but we have not looked into this.

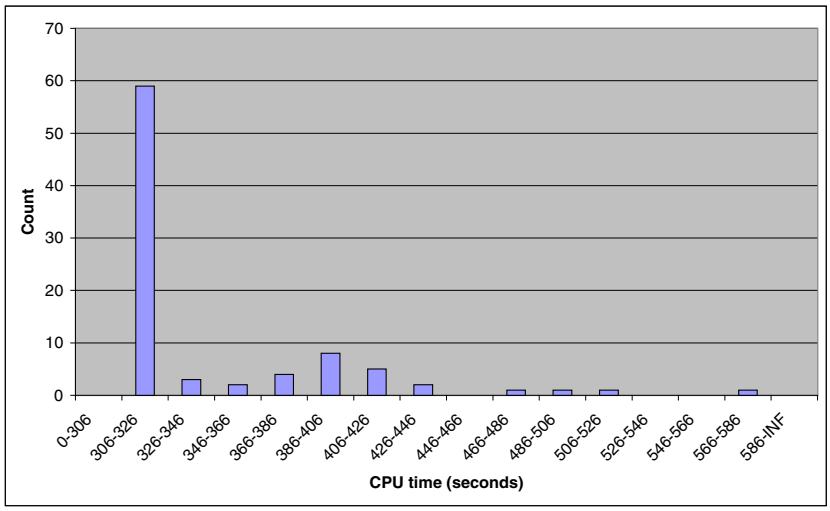

Figure 2: Histogram of CPU run-time on `embla.ntnu.no` for 87 low load observations.

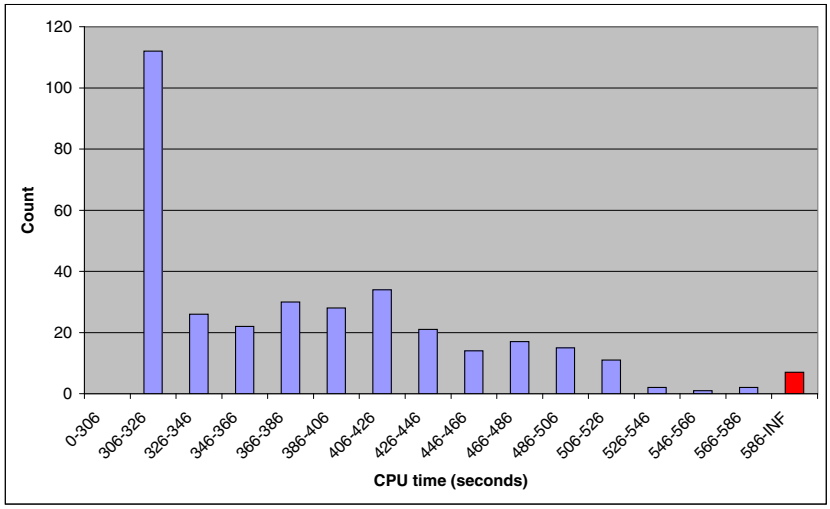

Figure 3: Histogram of CPU run-time on `embla.ntnu.no` for 338 high load observations.

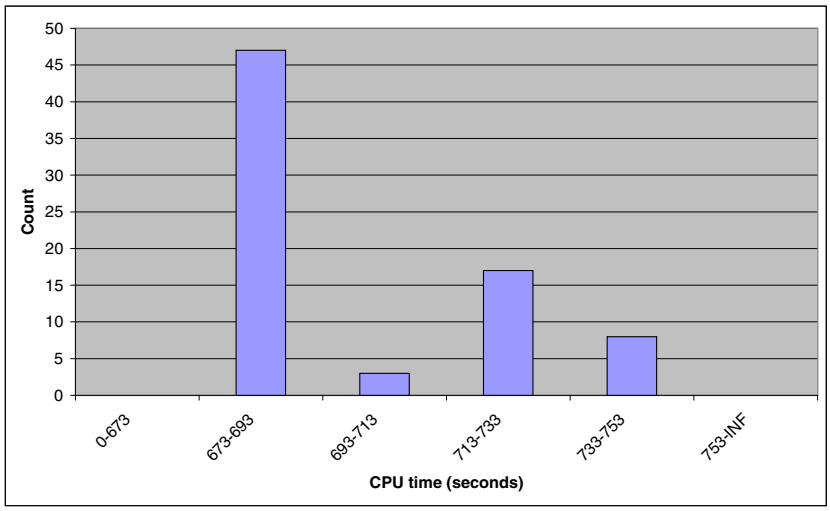

Figure 4: Histogram of CPU run-time on `korkeik.ii.uib.no` for 75 observations with load less than 5.1.

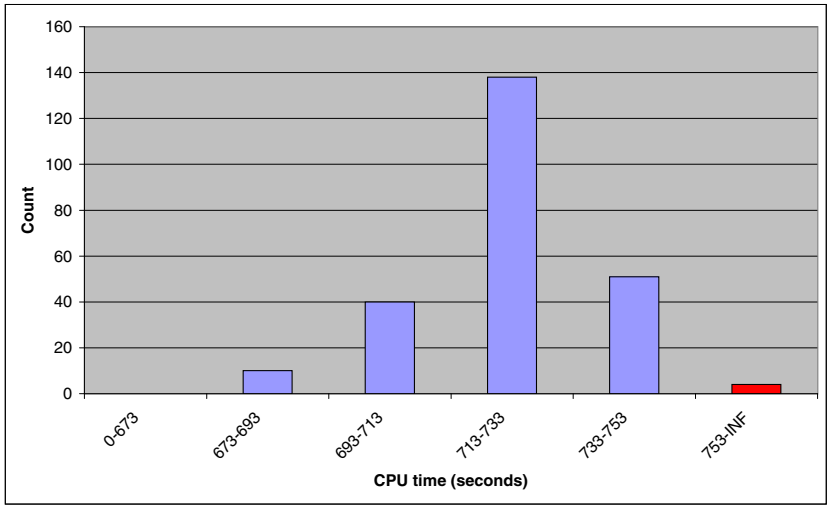

Figure 5: Histogram of CPU run-time on `korkeik.ii.uib.no` for 243 observations with load above 5.1.

Plotting histograms of the observed run-times for low and high load situations, Figures 2 and 3, we see that there is a difference in statistical properties between these two situations. Our initial assumpion that we get a cluster of execution times along a minimum is verified, but we also get a tail of observations which account for a larger proportion of the test runs when the load increases. It also appears that the tail approaches some form of gaussian distribution as the load increases. This is even more accentuated by 400 test runs of SeisMod performed June 2001 on `korkeik.ii.uib.no`, a 4 processor SUN Ultra-4 with 1GB of RAM at the University of Bergen. Up to 4 instances of the test program was run in parallel, providing a mix of loads as the individual test runs started and finished independently of each other. The distributions in Figures 4 and 5 show that for low loads, less than 5.1, we get strong minimum with a noticable tail showing, but for high loads, above 5.1, the spike at the minimum has dissappeared, and the observations approach a gaussian distribution.

In the following we will analyse the low load and high load situations on the SGI Origin 2800/128 `embla.ntnu.no`.

3 Run-time estimation

Our argumentation in the previous section and the plots in Figures 2 and 3 indicate that executions times do not follow a gaussian distribution. Rather we observe that the data seem congested along a minimal execution time, but with a distribution into CPU run-times the double of this.

A good estimator for CPU time comparisons should provide stable estimates for any random sample of k test runs, where k should be as small as possible. Thus given a sample d of k observations, organised such that $d_1 \leq \ldots \leq d_k$, we will apply the following *estimation techniques*:

- *average* $A(d)$ of d, as usual, defined by $A(d) = \frac{d_1+\ldots+d_k}{k}$,
- *median* $M(d)$ of d, as usual, defined by $M(d) = \frac{d_{\lfloor (k+1)/2 \rfloor}+d_{\lceil (k+1)/2 \rceil}}{2}$, which equals $d_{(k+1)/2}$ if k is odd,
- *lower quartile* $Q(d)$ of d defined by $Q(d) = \frac{d_{\lfloor (k+1)/4 \rfloor}+d_{\lceil (k+1)/4 \rceil}}{2}$, which equals $d_{(k+1)/4}$ if 4 divides $k+1$, and
- *minimum* $\min(d)$ of d defined by $\min(d) = d_1$.

For low values of k some of these measurements will coincide. Specifically, they will all be the same when $k = 1$, and for $k = 3$ the lower quartile and the minimum will be identical.

We did these calculations for k taking on all odd numbers in the range 1 to 19. Our 600 measurements D were first filtered into a low load set L of 87 observations and a high load set H of 338 observations. The observations in L and H are kept in chronological order. Then for each k the observations within each load set were rotated into running groups of k consecutive observations. Thus for low load and $k = 11$ the first set of data would be L_1 to L_{11}, the second of L_2 to L_{12}, the third of L_3 to L_{13}, and so forth up to the 87th group which consists of L_{87} to L_{10}. This reuse of observations for different groups is not considered problematic since each

		k=1		k=3		k=5		k=7		k=9	
average	L	344	16%	344	9.5%	344	7.0%	344	5.6%	344	5.0%
$A(d)$	H	387	19%	387	11%	387	8.9%	387	7.8%	387	6.9%
median	L	344	16%	332	10%	328	9.0%	321	6.1%	319	4.6%
$M(d)$	H	387	19%	377	15%	374	13%	373	11%	372	11%
quartile	L	344	16%	315	4.4%	313	1.2%	313	0.88%	313	0.67%
$Q(d)$	H	387	19%	332	9.9%	332	7.6%	329	7.6%	328	6.6%
minimum	L	344	16%	315	4.4%	312	0.64%	312	0.14%	312	0.095%
$\min(d)$	H	387	19%	332	9.9%	321	6.0%	317	4.3%	315	2.7%

		k=11		k=13		k=15		k=17		k=19	
average	L	344	4.5%	344	4.0%	344	3.5%	344	3.2%	344	2.9%
$A(d)$	H	387	6.4%	387	5.9%	387	5.6%	387	5.3%	387	5.1%
median	L	318	2.9%	318	3.0%	317	2.9%	316	1.4%	316	1.2%
$M(d)$	H	370	9.5%	370	8.4%	369	8.0%	369	7.8%	369	7.6%
quartile	L	312	0.56%	312	0.36%	312	0.19%	312	0.096%	312	0.071%
$Q(d)$	H	326	6.6%	326	5.8%	325	5.7%	325	5.3%	324	5.3%
minimum	L	312	0.089%	312	0.083%	312	0.080%	312	0.076%	311	0.072%
$\min(d)$	H	314	2.0%	313	0.67%	313	0.52%	313	0.44%	312	0.41%

Table 1: Average of CPU run-time estimates (in seconds) and relative standard deviation for each group of observations.

observation is independent of the others, and each group is handled independently of the others. The result is presented in Table 1. Each row contains the average (in seconds) and relative standard deviation, also called coefficient of variation, (in percent) for low load (87 groups) and high loads (338 groups) of observations, 1 row for each estimation technique. The columns are given for different k.

When setting up Table 1 we have assumed that, for each k and estimation technique, the resulting CPU run-time values will vary according to a gaussian distribution. Since we have 87 and 338 groups to compute over (low load and high load, respectively), we may meaningfully interpret the average (given in seconds) and the standard deviation (given in percent of the average) of these estimates. Looking at the numbers we see that the behaviour for low load and high load situations are distinct, but that the trend for each load group is the same: the standard deviation falls much more rapidly for increasing k for low load situations. Discussing each estimation technique, i.e., each row in the column we see that:

average: The average values remain unchanged for increasing k. This is because taking the average of equally sized groups of averages is equivalent to taking the average of all observations. The standard deviation remains the highest of all estimation techniques, and is acceptable (less than or equal to 5%) only for $k \geq 9$ in low load situations.

median: The average median drops slightly with increasing k, but soon stabilises for low load situations, $k \geq 7$. It reaches a plateau sooner, but at a higher value, for high load situations. Standard deviation remains fairly high throughout, but is acceptable for $k \geq 7$ in low load situations.

quartile: The average best quartile drops with increasing k. It soon stabilises for low load situations, $k \geq 3$, but needs larger samples for high load situations,

	k=1	k=3	k=5	k=7	k=9	k=11	k=13	k=15	k=17	k=19
load ≤ 80	16%	4.4%	0.64%	0.14%	0.095%	0.089%	0.083%	0.080%	0.076%	0.072%
$90 \leq$ load ≤ 125	19%	9.9%	6.0%	4.3%	2.7%	2.0%	0.67%	0.52%	0.44%	0.41%
any load	19%	10%	5.8%	3.5%	2.2%	1.7%	0.95%	0.86%	0.80%	0.75%

Table 2: Standard deviation for CPU run-time estimator minimum versus sample size.

$k \geq 11$. Standard deviation remains fairly high for high load situations, but is good for low load situations when $k \geq 3$.

minimum: The average minimum drops rapidly with increasing k. For low load situations it immediately stabilises for $k \geq 5$, and the standard deviation is very small (less than 1%) from here onwards. For high load situations similar results appear later, $k \geq 11$, when the standard deviation is less than 2%.

It seems clear that using the average of CPU run-time as estimation technique has a relative high standard deviation. The best estimation technique seems to be the minimum, where we get good results (standard deviation less than 2%) for 11 observations, and less than 1% already for 5 observations in low load situations. Note that this technique works fine also during a normal run-time regime of the machine.

Basic statistic theory uses the rule of thumb that a distance of at least two standard deviations is a safe indicator of distinctness of two sets of observations (95% confidence interval). A distance of three standard deviations gives better than 99.5% confidence. Using the data from Table 1, we may then decide how many observations are needed if we need to be certain that two programs with slightly dissimilar run-times have significantly distinct CPU run-times. This is tabulated in Table 2 for low load, high load, over all loads, and for varying group sizes. What will constitute low and high loads on other architectures must be checked with a set of test runs.

For comparison of speed, we executed 9 runs of the SeisMod benchmark used at NTNU, on the Origin 2000 `ask.ii.uib.no` at Parallab in a mixed load situation. The minimum CPU run-time in at Parallab was 505 seconds, indicating that the new Origin 2800/128 is significantly faster than its older version. The relative speed increase, 1.6 times, measured by this technique is consistent with the factor of 1.4 to 1.8 from the benchmarks reported by [Meu00] (November 1999 for the Parallab machine, November 2000 for the NTNU machine).

4 Conclusion and future work

We have performed empirical investigation of several estimation techniques of a program's run-time when a fully controlled environment is difficult or impossible to establish. The best estimator seems to be to take the minimum of a group of executions, where the standard deviation of this estimator rapidly decreases to very low values (less than 1%) for increasing sample sizes. The average estimator, which is commonly used, does not provide such stable results.

This investigation used only one program on one machine to establish the confidence intervals. Although similar results have been seen on other machines for

low load situations, confere Figures 2 and 4, the estimation technique should be checked for a range of programs on several machines. This should also include parallel programs.

It would also be interesting to investigate what kind of statistical distribution CPU run-time variations have. It seems to be a combination of a one-sided distribution for low load situations (confer Figure 2) with a more gaussian like distribution taking over for high loads (see Figure 5). An explanation of this phenomenon is also needed, a possibility being that high loads imply interferences on resource usage between programs, and that this kind of interaction may be expected to have a gaussian distribution. So far we have looked at just one factor, average computer load, as base for determining variation. Factors such as program memory requirements may be important. Further studies taking this and other factors into account should be performed.

Acknowledgements

Thanks to Ivar Heuch, professor in mathematical statistics at the University of Bergen, for useful suggestions.

References

[Don01] Jack J. Dongarra. Performance of various computers using standard linear equations software. Technical Report CS-89-85, Computer Science Department, University of Tennessee, and Mathematical Sciences Section, Oak Ridge National Laboratory, 2001.

[DRW92] J.W. Davidson, J.R. Rabung, and DB Whalley. Relating static and dynamic machine code measurements. *IEEE transactions on computers*, 41(4):444–454, 1992.

[HFJ99] Magne Haveraaen, Helmer André Friis, and Tor Arne Johansen. Formal software engineering for computational modeling. *Nordic Journal of Computing*, 6(3):241–270, 1999.

[Jai91] Raj Jain. *The art of computer system performance analysis: techniques for experimental design, measurement, simulation, and modeling.* John Wiley and Sons, inc., 1991.

[LKL95] G. Lyon, R. Kacker, and A. Linz. A scalability test for parallel code. *Software: practice and experience*, 25(12):1299–1314, 1995.

[Meu00] Hans Werner Meuer. The top500 project of the universities mannheim and tennessee. In Arndt Bode, Thomas Ludwig, and Roland Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 43–43. Springer Verlag, 2000. Also see http://www.top500.org.

[SC00] Xian-He Sun and Kirk W. Cameron. A statistical-empirical hybrid approach to hierarchical memory analysis. In Arndt Bode, Thomas Ludwig,

and Roland Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 141–148. Springer Verlag, 2000.

[SHCL01] X.H. Sun, D.M. He, K.W. Cameron, and Y. Luo. Adaptive multivariate regression for advanced memory system evaluation: application and experience. *Performance evaluation*, 45(1):1–18, 2001.