In *Computing Science and Statistics* **30** (1998), pp. 309–314.

Predicting Algorithm Performance

Joshua Landrum Janis Hardwick Quentin F. Stout
University of Michigan, Ann Arbor, Michigan 48109 USA

Abstract

We examine the problem of predicting the time that an algorithm will take to solve a problem of specified size on a specified computer, and of predicting the uncertainty in the estimate. The predictions are based on an adaptive sampling approach, constrained by a fixed time budget for all experiments. Complications arise due to the poorly understood nature of system variation, because small inputs may be poor predictors of the performance for large inputs due to changes in cache behavior, and because time is both a constraint and the variable being estimated.

Keywords and phrases: experimental algorithms, computer performance prediction, regression, model checking, design of experiments, adaptive sampling

1 Introduction

At times it is desirable to know how long a computer program will need to run to solve a problem with a given input size on a specific machine. By size, we mean such things as the dimensions of a matrix or the length of an array, etc. While in some cases there may be asymptotic guidelines for the growth rate of running time as a function of input size, such rates provide neither input- nor machine-specific answers.

Consider the problem of factoring large integers, one that arises in cryptographic research. As the size of the integer grows, factoring it becomes increasingly time consuming; and thus, it's important to be able to predict how long it will take to factor very large integers. For the factoring problem, researchers are able to combine analytic models of growth with empirical estimates derived from solving smaller problems to make good predictions about larger ones. However, for many other problems analytic guidelines are not available or may not apply to the target problem size. In such cases, empirical methods may provide the best basis for formulating predictions.

In this paper, we focus on a special case of the general problem just described. We assume that there is a fixed *budget* of machine time during which all exploratory runs must be carried out. Incorporating a budget not only helps us define the prediction problem, but it also addresses serious practical concerns. Even given the constraint of a budget, an investigator's knee-jerk reaction may be to run a program at the largest size believed to fit within the budget. The intuition behind such an experiment would be based on assumptions that the time is a known deterministic power of size. As we will see, these assumptions may be far from realistic.

What we propose here is that statistical techniques be employed for the problem of algorithm prediction. For a specific machine and problem setting, identifying and modeling relevant sources of variation is critical to the task of making accurate predictions. Variation, for example, can be introduced by the data, by the algorithm, by the hardware and software system running the algorithm, and by other users on the system. It is also imperative to incorporate model checking into the prediction problem. The standard approach of using order notation analysis to model growth provides only asymptotic rates which, without specific constants, are too vague to apply in practice.

In general, previous work in the algorithm performance and prediction literature has tended to ignore both model checking and sources of variability. Further, we know of no previous work that incorporates a time budget into the prediction process.

Certainly, there has been research on predicting machine performance using measures such as instruction counts. However, while these techniques are useful for improving code performance, especially when the instruction mix is known (see [2]), carrying out detailed counts for specific inputs completely ignores system variability. Furthermore, most timing methods require access to source code which we do not necessarily assume to be available to the end users making the predictions.

Thus, while we take a "black box" approach to the problem of predicting running time, we believe that doing so addresses a problem of significant importance in computer science and general computing. Although tackling the problem in this way makes it more difficult to handle, the results will apply in far more general situations.

To summarize, we address the following problem: Given a particular machine, a target problem size, and a fixed time budget during which all experiments must

take place, we wish to

(a.) Predict the running time for the target size, and

(b.) Assess the accuracy of the prediction.

In the next section, we provide a basic model for the problem and in Section 3 we describe a simple adaptive approach to solving it. In Section 4 we discuss sources of timing variation, and in Section 5, the effects of cache memory on time. In Section 6 we propose an improved design algorithm, and we conclude with a short discussion of ongoing work in Section 7.

2 Model and Notation

To simplify (and shorten) explication, we work with a simple model in which the **time**, t, required to run a problem depends only on the **size**, s, of the program input. In some cases, time is a deterministic function of size; and for our purposes here, this is assumed, although it is not a requirement for the approach taken. Specifically, we assume the relationship between time and size to be one commonly arising in the computer science literature. Let

$$t_i = \beta_0 + \beta_1 s_i^r + \varepsilon_i \quad \text{for} \quad i = 1, 2, \ldots \tag{1}$$

where β_0, β_1 and possibly r are unknown parameters.

In modeling the errors, we regard "system interrupts" to be the fundamental source of variation. While a plethora of factors may contribute to whether an interrupt occurs (see Section 4), we model them together as the black box that produces time delays. During each time unit, there is a fixed probability that an interrupt will occur and add a fixed extra amount of work. Only the time charged to the algorithm, not the full time to service the interrupt, is being considered. One way to view the number of interrupts that occur during a given run is as the sum of random variables whose variance is proportional to their number. In other words, as program input size increases, more time units are expended and during each of these, there either is or is not an interrupt. We assume that the total expected running time for the program is the underlying time plus the total interrupt time which, as noted, depends on size. Thus, we take

$$E(\varepsilon_i) = 0 \quad \text{and} \quad Var(\varepsilon_i) = \sigma_i^2 = e_i(\beta_0 + \beta_1 s_i^r)^p,$$

for $i = 1, 2, \ldots$ and $Var(e_i) = \sigma^2$ and p unknown. On occasion, it will be reasonable to assume also that the errors, ε_i, are independent and normally distributed with variance σ_i^2.

1. Sample a few sizes, to get a rough approximation of the growth rate.
2. Repeat the following until the budget is exhausted:
   - Use the current observations to estimate parameters.
   - Pick as the next sample point the one within the budget range which maximizes
   $$\frac{\textit{expected variance improvement}}{\textit{projected cost of the sample}}$$
3. Predict the time the target problem size will take and provide an estimation of the variance.

Figure 1: Basic Adaptive Sampling Procedure

3 Algorithm for Adaptive Design

Since sampling costs are proportional to time which in turn is what is being modeled, we face the problem of estimating how much of the budget will be used when sampling at a certain size. This suggests *adaptive* selection of design points so that previously observed data can be utilized to generate an estimate for the cost of sampling at future sizes.

A simple, iterative, greedy or "myopic" approach to adaptive selection of input sizes is illustrated in Figure 1. While techniques that look further ahead might improve accuracy, they tend to have higher computation cost, and this cuts into the overall time budget.

To estimate the variance of the prediction (and thus the expected improvement gained by sampling at a given size), it is helpful to have the prediction as a function of the timing measurements. To address this, we first assume that r is unknown, $\mathrm{E}(\varepsilon_i) = 0$ and that the errors are independent. We then expand the error term to second order, and retain only the terms proportional to s^{2r}, where s is the target problem size. Assuming that the target size is large in comparison with the observed sizes, this technique is expected to provide a good approximation.

As we will find, however, using the simplistic algorithm in Figure 1 can be problematic. For example, the budget constraint makes it desirable to observe runtimes for small sizes which are unlikely to actually fit the model in the region of interest (see Section 5 for details). Furthermore, the growth rate, p, of the variance is unknown, and estimating it is not dealt with in the present sampling algorithm. These two issues are addressed via a more refined algorithm in Section 6.

4 Sources of Variation

The sources of variation that affect the run time of a program can be grouped into two categories: those internal to the problem and those external to it — dependent on the system but not the inherent work the problem performs.

For example, some algorithms, such as those used to locate large primes for cryptography, are probabilistic. Naturally the randomness in such algorithms produces variation in running time. This is a case in which variability is *internal* to the problem. As another example, consider the program input. In some situations, running time depends only on input size and not on the input itself. This is true with matrix multiplication where the individual elements of the matrix have no impact on running time. In other instances, time can be affected by the actual inputs. Consider searching an array. If the element being sought is the first element examined, then the search time is quick. However, if the element isn't even in the search space, more time is taken for the search — even though the array size is the same in each case.

Thus, if you know everything about a given algorithm, i.e., if you have the source code, you may be able to model internal factors such as those just mentioned. In this paper, however, we work only with problems having no internal variation. Any variability associated with the algorithm itself is considered along with the mix that we refer to as "system variation".

As noted earlier, external sources are modeled as system interrupts. Interrupts might come from either the operating system, e.g., checking to see if the job has been killed; or from swapping the process out entirely, such as occurs on a multi-user machine. Figures 2 and 3 in Section 5 appear to show anomalies resulting from a multi-user system. Since interrupts such as these are unlikely to be uniform, we anticipate that some interrupts will come in bursts and cause a good deal of short-term correlation in timing measurements. Other measured sources of variation, such as those caused by poor clock resolution or periodic system interrupts, are in fact highly regular. A problem in which both small periodic and large aperiodic system interrupts occur is examined in [1].

Memory usage can also provide timing variations. An interrupt, for example, may cause cached memory to be flushed. If this memory is used again by the process, and would not have been flushed otherwise, it must be reloaded, and hence the interrupt causes a further delay.

Yet another source of variability comes from the model itself. To avoid overfitting, while allowing generality, only two terms in the polynomial describing the problem's "true" rate of growth are considered (the constant term is included to deal with the changeover regions discussed in Section 5). Of necessity this means the model may be a less good fit for some regions. This is true in particular, for small sizes for which lower-order terms may be significant.

Some machines may introduce some randomness at the hardware level. For example, when pushing a system to run as fast as possible, some consistency might be sacrificed. There are times when a memory read might take between 60 and 65 cycles, rather than always taking 62. Other machines use random cache replacement algorithms, to avoid pathological memory problems. While hardware variability isn't explicitly incorporated into our model, it too is assumed to contribute to the black box representing general system variation.

We have only touched on a few of the many sources of variability that affect this prediction problem. As we continue to work in this area, and as systems become increasingly complicated, we anticipate that we'll encounter quite a number of others.

5 Cache Effects

Modern computers are equipped with various types of memory. Each has a small amount of fast memory, called *cache*, coupled with a much larger and slower main memory system (*RAM*). Disk memory, sometimes called secondary storage, is even larger and slower, but will not be considered here.

Problems run at "small" design points (small sizes) fit into the cache memory while those run at "large" design points (large sizes) do not. Between the small and large sized problems, there will be a declining portion of data residing in cache as the problem size increases. Ideally the transition from fitting in cache to requiring main memory would be sharp and the region where the changeover takes place would be clearly demarcated. An illustration of such desirable behavior is given in Figure 2 where timing data for a very simplistic problem are displayed. For this problem, every 16th element of an array is accessed in turn, with this access being repeated several times. The data are the times for the process to run for increasing array sizes. The HP workstation that generated these data had a cache of 512k bytes, or 128k words (our unit of measure for size).

There are 4 regions of interest in this graph. For array sizes

1. $0 < s < 128k$ (i.e., when the entire array fits into cache), the time, t, is of the form $t = a_c s$;
2. $128k < s < 256k$, the time is of the form $t = a_t s + C$;
3. $s > 256k$ (i.e., when the array is more than twice the cache size), the time is of the form $t = a_r s$;

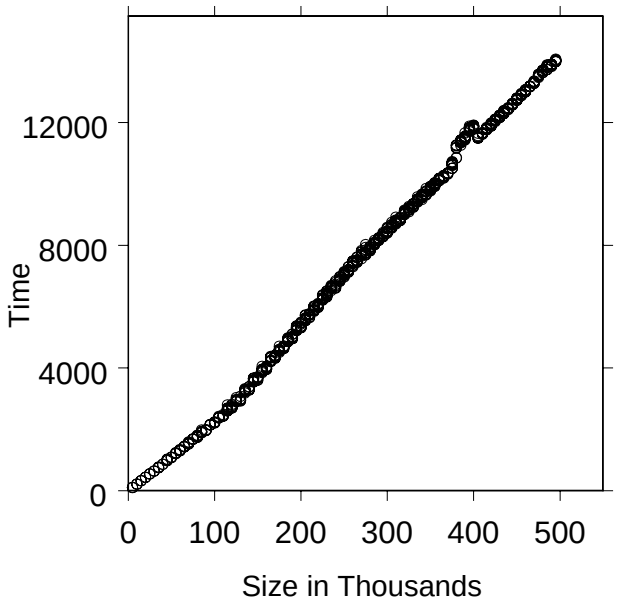

Figure 2: Sequential Array Access Times, HP processor.

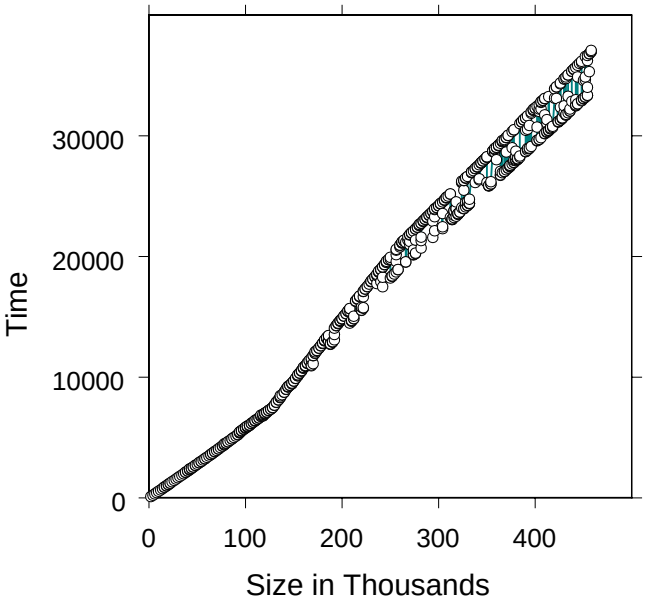

Figure 3: Sequential Array Access Times, HP run 2

4. $s \approx 380k$, the time is markedly higher than those of neighboring array sizes.

Note that, in region (1), every reference is to data residing in cache, while in region (3) references are only to data in RAM. Thus we have $a_c < a_r$. In region (2), we observe transition behavior. Here, the beginning and end of the array map to the same cache locations and hence are never in cache when referenced. The middle of the array, however, remains in cache, although, as the array size increases, the "middle" shrinks. In this case, each increment in size adds another reference to RAM and replaces a reference to cache with one to RAM. Thus $a_t = 2a_r - a_c$. In region (4), the unusually high values are assumed to be the result of multiple users accessing the system.

As a digression, it should be mentioned that upon rerunning this very same problem on the very same machine, we encountered perplexing difficulties of a different sort. The data in Figure 3 show what appears to be a bimodal distribution of running times for large sizes. Note that, while we retain from these data the information about the changeover region, we also find what appear to be two separate lines being fit by the timing data. Our best explanation for this is again the multiuser phenomenon, but even given this, we find these data hard to interpret.

Getting back to cache effects, we have also found that such effects can vary among systems. In Figure 4, for example, we again see data for the linear array problem. The only difference here is that these data were generated by an IBM machine. Note that it is difficult to discern any changeover region due to cache effects.

Yet another example of difficulties associated with cache effects is illustrated in Figure 5. These data reflect the time taken to do matrix multiplication using an implementation of the Strassen algorithm. The runs were carried out on a multiuser machine for matrix sizes ranging from 500×500 to 1500×1500. This is an algorithm for which we know the asymptotic growth rate, so one can view the problem as being as simple as the one just considered. One hundred sample observations were taken at each design point, and careful examination of the figure shows that the growth rate is not smooth. Furthermore, as it turns out, our best estimate of the growth rate for sizes beyond the changeover region does not yet conform with the asymptotic rate. Note, by the way, that the data in Figure 5 comprise far more samples than would reasonably be used for a fixed-budget problem. The goal in this case was simply to study the complexity of experimentally measuring the growth rate and variance growth rate in a problem that we knew something about.

In closing this section, we note that while many researchers have examined sources of variation such as cache behavior, none that we know of have coupled the work with prediction efforts which estimate the variation.

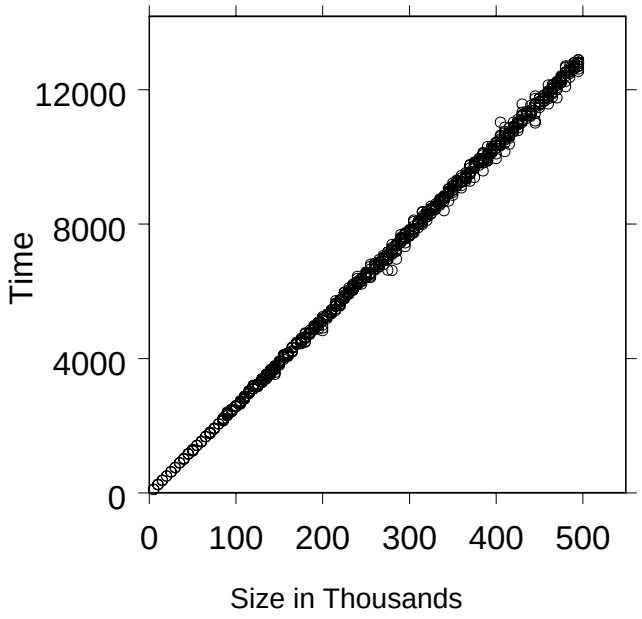

Figure 4: Sequential Array Access Times, IBM processor

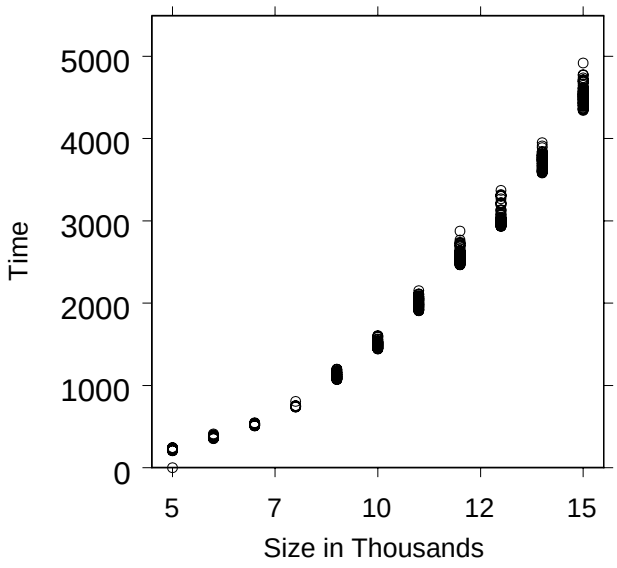

Figure 5: Strassen Multiplication, Sun proc. — Multiuser

6 Revised Algorithm

As has been discussed, the actual design problem that we face is far more complex than that addressed in Section 3. In this section, we outline a more refined algorithm that better handles some of the difficult issues that we've encountered. In particular, two problems we consider are as follows:

1. Up to a point, small size inputs will give data inconsistent with the growth rate we seek.
2. The growth rate, p, of the error variance is unknown and needs to be estimated.

With regard to issue (1.), we know that sampling at sizes that are too small "wastes" some of our budget since the timing data from these sizes occur before we have passed through the changeover region. Unfortunately, however, locating the changeover region is a statistical problem in and of itself and it is one that we cannot afford to ignore. Some small sizes must be run before the changeover so that the region can be identified. However, once this identification is believed to have been accomplished, we wish to drop times resulting from the small sizes and focus on obtaining results from the region of interest. To this end, we introduce model checking into the adaptive design procedure detailed in Figure 1.

If the estimated model appears not to fit the data well, it is assumed to be the fault of the small design points. The revised algorithm calls for the ongoing deletion of times from the smallest design points as sampling at larger sizes continues. Only the latter data are then used to predict the target time. Note that an important problem for a further revision of the algorithm is to assess when to stop deleting small sizes. There are various ways to model this, but we have yet to incorporate them.

Next, it's important to be able to detect when the growth rate of the problem doesn't follow a polynomial. The same model checking used to detect which small sizes to drop can be used to this end. Once all the samples are taken, the data are refit, and a prediction for the target is made. If the polynomial model still doesn't fit the data — or if too many points had to be thrown out to make a good fit — a flag is raised to alert the user of possible problems.

To deal with issue (2.), the model is again called into play for estimation purposes. Here the deviation of the sample points from the predicted curve serves as a proxy for the variation. Because of the statistical bias introduced by timings at small sizes, it is even more imperative to ignore such values when estimating the variance growth rate.

7 Future Work

This project is ongoing in the sense that we plan to incorporate a variety of refining factors into the model and sampling design. Here, we briefly mention some of our

concerns.

First of all, to do model checking, the points sampled must be spread out. Not surprisingly, this goal competes with that of minimizing the variance of the prediction. As a result we need to develop a weighting scheme to balance the two goals. While the greedy method for selecting sample points is still reasonable it is also ad-hoc. We hope to apply better methods to the problem of data fitting.

Next, we need to develop a better understanding of the sources of timing variation. The following are a few of the problems on which we are presently working.

- The observed variance does not fit the independence and normality assumptions;
- The variance growth rate may not be a simple power of the average time;
- The variance may depend on measurable covariates such as fraction of CPU time devoted to other users.

Finally, we would also like to include more than a one dimensional version of size. Examples of cases where this is needed are

- Graph problems which depend both on the number of nodes and the number of edges
- Jobs done on parallel machines might be run on a varying number of processors

In closing, we believe that the problem of algorithm prediction is in its infancy, and we hope that the problem presented here will stimulate interest in both the computing and statistical communities.

Acknowledgments

This research was partially supported by the National Science Foundation under grants DMS-9157715 and DMS-9504980.

References

[1] Tabe, T., Hardwick, J., and Stout, Q.F., "The communication performance of the IBM SP2", *Computing Science and Statistics* **27** (1995), pp. 347–351.

[2] Saavedra-Barrera, Rafael H. and Smith, Alan Jay, "Performance Prediction by Benchmark and Machine Analysis", *Report No. UCB/CSD 90/607 December 1990* Computer Science Division (EECS) University of California