Measuring and Analyzing Real-Time Performance

KEVIN B. KENNY, Northrop Research & Technology Center
KWEI-JAY LIN, University of Illinois at Urbana-Champaign

You must know time and resource requirements to design a real-time system. Flex determines time requirements by measuring actual execution. It can incorporate programmer knowledge about expected timings.

It is difficult to design and build hard real-time systems: systems in which computations must be completed at predefined times. Hard real-time systems must meet critical requirements for dependability and safety, producing correct results by a deadline.

One key to satisfying the deadline requirement is determining the timing behavior of the system's tasks. Programmers must be able to determine the maximum execution time of any given task.

Many performance-analysis systems¹ use analytical methods to determine a task's maximum execution time. Such systems usually use a model of the target architecture to calculate the expected execution time of a given source-code segment.

Some use detailed models that take into account low-level architectural features such as pipeline instructions or cache replacements; others use simple models of high-level instructions. These methods usually cannot model system overhead such as the time spent accessing secondary storage or using system servers. They also cannot handle unbounded loops or recursive program structures.

Flex, an experimental real-time language we are developing for the Concord project,¹ instead uses an empirical approach that first measures the actual timing behavior and then uses the measurement results to determine the parameters of a programmer-supplied timing model. This timing model gives the system the programmer's understanding of the program's timing behavior in terms of its asymptotic time complexity. The measurement system determines the exact val-

uses of model parameters using sophisticated statistical methods to derive the program's timing characteristics precisely.

Flex is better than performance analyzers that examine only code because it can cope with more kinds of program structures and it does not depend on an underlying hardware model. No other system we know of uses the programmer's knowledge about timing behavior and offers a quantitative measure of statistical confidence in the validity of the performance model.

**Requirements**

Systems that use a hardware model to determine the maximum time of tasks in real-time systems² have fundamental limitations.

First, the general problem of calculating runtimes for arbitrary programs is undecidable, so these systems limit the primitives available to the programmer. For example, they force the programmer to specify a bound on every loop and they forbid recursion and recursive data structures like linked lists in favor of statically allocated tables. These restrictions can both complicate programming and cause gross runtime inefficiencies — exactly what you don't want in real-time systems.

Even if the restrictions were acceptable, there would still be the problem of validating the hardware model. Many analysis systems assume a fixed time for each instruction type, ignoring the uncertainties in time caused by things like caches and paging systems.

Therefore, despite their attempts at utmost reliability, these systems generate results that are approximate at best. There is no good way to quantify how well the time they've computed reflects the actual time a task needs.

**Empirical approach.** Flex measures the program in actual execution and compares the measurement results to a model of the program's behavior so you are confident the program will continue to behave as measured.

The programmer's knowledge about an algorithm's performance is of value here. While programmers are notoriously bad at estimating runtimes, frequently they have a very good idea about performance, at least in the asymptotic sense. What their mental models lack is a good perspective of the time required for the program's individual operations — exactly what measurements provide.

A system that can measure the time a task requires under various conditions and integrate these measured times into a programmer-supplied parametric model will be able to:

- analyze program structures like unbounded loops and recursive control structures,
- provide accurate timing information, even on hardware whose timing behavior is difficult to model and analyze, and
- validate the timing model statistically for goodness of fit.

**Constraint block.** The Flex compiler is such a system.² Flex primitives let the programmer specify and enforce the timing behavior of a code block. Its fundamental construct for specifying and enforcing performance behavior is the constraint block.

A constraint block comprises a Boolean expression (the constraint) that must remain true whenever the block executes, the code block to which the constraint applies, and an optional exception handler that executes when the compiler detects a constraint failure.

For example, (duration ≤ 1500) specifies that the computations, whenever they begin, must take no more than 1,500 time units to complete, and (start ≥ t₁; finish ≤ t₂; temperature < 110) specifies that the computations must start at a time no earlier than t₁, and end at a time no later than t₂. During computation, the contents of the constrained variable temperature must be less than 110 at all times. Presumably, this variable will be maintained by a process that polls a sensor.

**Flex System**

Figure 1 shows the structure of the Flex compiler's timing-measurement system. The programmer adds directives to a Flex or C program to direct the system to measure the timing behavior of certain code blocks. The compiler generates object code with instructions inserted to determine the elapsed time of these blocks and record them, together with data on the operating conditions, to a measurement file. The compiler also writes a description of the parametric model to another file.

**Language support.** To support timing measurement and analysis, Flex uses a new directive, #pragma measure. This directive both directs the compiler to generate code to measure the time or resources consumed by a code block and provides the parametric model for analyzing the measurement. The syntax for this directive is:

```
#pragma measure <resource defining parameters in expression [safety]]
```

Case, which can have two values (mean and worst), identifies whether the model is expected to fit to the mean resource usage or to the worst case. Generally, the programmer will use the mean if he expects the block's timing behavior to be deter-

![Figure 1. Flex's timing-measurement system.](image-url)
ministic (any variation will be caused by measurement error) and the worst case if he expects it to be random or unpredictable.

Resource, which in Flex can have two values (duration and count) identifies which resource to measure. Flex considers duration to be the amount of real time the block consumes and count to be the number of trips made through the block’s top-level control structure. We have not experimented with other resources, but it would be easy to model the amount of memory a program uses, the communications bandwidth it consumes, the number of processors it uses, and so on.

Parameters are the model parameters, and the expression models expected resource behavior. Variables in the expression not identified as parameters are free variables, recorded by the measurement logic so the timing analyzer can use their values to determine the best fit to the observed performance data.

The optional safety clause specifies a safety factor — a number by which the system multiplies the computed time to determine the time to allow for a computation. The safety factor lets the programmer tell the system to always provide, for example, at least 10 percent more time than the worst case seen so far.

Figure 2a shows a #pragma measure directive. Here the programmer has coded an insertion sort. He expects the mean duration of the sort to be a quadratic function of n, the length of the list. The programmer has specified that the system provide a safety factor of two — the insertion sort should always be allowed to take double the mean time that it has been observed to take for any given list length. This safety factor reflects the programmer’s knowledge that insertion sort, in the worst case, requires twice the time it does on average.

Data collection. The #pragma measure directive causes the compiler to insert probe points in the generated object code. (In a production system, the measurements will likely be done by an in-circuit emulator or other hardware monitor.)

For each execution of the block, the program records the location of the #pragma measure, the contents of all the free variables, and the value of the performance statistic being measured. The code generated for the program in Figure 2a will, for example, look something like the code in Figure 2b. (The code to write the variables to the data file uses in-memory buffering so it doesn’t slow the program and spoil the measurement.)

The directive also generates a new function, a measurement function, which lets the program examine the system’s best estimate of the time it will take. This function simply evaluates and returns the expression in the parametric model, obtaining the model parameters from an earlier run of the timing analyzer. Figure 2c shows the measurement function for the program in Figure 2a.

Timing analysis. After the program has run, possibly several times, and measurement data has been collected, the program runs a timing-analysis program. This program determines the best fit of the parameters to the observed runtime and produces a report that describes this fit and gives a confidence level for the model. The timing-analysis program also updates the file containing the model description with the observed values for the parameters.

Curve fitting. The process of determining model parameters given the observed time and resource data is a curve-fitting problem; statistically, we describe it as a chi-square minimization problem. For each observation i in a #pragma measure statement, the system records y_i, the observed quantity (the amount of time or

```c

```
other resource consumed). It also records the contents of the free variables \( v_i \), \( v_{m_i} \) in the expression.

The system then reduces the data by replacing groups of observations that have the same free-variable values with a single record of the mean value of the observed quantity \( y_i \) and the standard deviation of \( y_i \). The problem then is to determine a set of values for the variables \( x = (x_1, \ldots, x_n) \) that minimizes the chi-square statistic

\[
\chi^2(x) = \sum_{1 \leq i \leq m} \left( \frac{f(x, y_i) - y_i}{\sigma_i} \right)^2
\]

There are many techniques to do this (Charles Lawson and Richard Hanson have written an excellent survey of them\(^4\)). Unfortunately, many of them are unsuitable for the difficult task of estimating parameters for \#pragma measure.

First, techniques that treat the curve-fitting problem as a linear-regression problem can be discarded because many models are not linear with respect to the model parameters.

Nonlinear problems can have a complex topology. If we imagine a plot of chi-square as a function of the model parameters, the path from an initial estimate to the minimum may proceed down a valley that turns one way and another. Such a complex topology can make methods such as Gauss-Newton iteration fail to converge. It can also slow convergence in methods like conjugate-gradient and downhill-simplex in multidimensions.

The biggest danger is that the parameter-estimation process may be underdetermined: The observed data may not distinguish among parameters. One possible source of this ambiguity is if the programmer accidentally creates a model for some complex behavior that has parameters whose values are completely determined by some combination of other parameters. For example, if he specifies a model like \( f(n) = A \cdot \exp(Bn + C) \), \( A \) and \( C \) are completely dependent on one another.

Another, less controllable, ambiguity source is when the observed data fail to distinguish between two behaviors. For example, if the system observes only a small range of \( n \), a model with \( O(n^2) \) and \( O(n \log n) \) terms may give a very good fit to the observed data with either a large \( n^2 \) coefficient and a small \( n \log n \) coefficient or a large \( n \log n \) coefficient and a small \( n^2 \) coefficient. As the number of parameters increases, so does the likelihood that one parameter will be nearly dependent on others.

This last problem implies that the parameter-estimation process may be fundamentally ill-conditioned: Determining the best fit may involve delicately canceling large parameters. (A solution of \( f(n) = 10^3 n^2 - 10^3 n \log n \) may give results close to the observed data points but is not likely to extrapolate well or demonstrate numerical stability.) We therefore need a method that remains numerically robust in the face of an underdetermined problem.

After trying a number of methods, we decided to incorporate a modified Levenberg-Marquardt algorithm\(^5\) in our timing-analysis system. At one point the original algorithm calculates an approximate inverse of a matrix, which will be rank-deficient if the problem is underdetermined. In our analyzer, the inversion is done using the singular-value decomposition with careful editing of the singular values\(^6\) to ensure adequate convergence and avoid delicate parameter cancellation.

**Host-system support.** For timing measurements to be useful, the machine's clock must be accurate and it must measure a time unit that is much shorter than the duration of the code blocks being measured. This is a problem on our development system, a Sun 3/50, whose clock has a very coarse resolution of 20 ms. This coarse resolution has prevented us from measuring fine-grained functions.

Reading the clock must be inexpensive — it must take much less time than the function to be measured. Likewise, writing a measurement record must be inexpensive.

Many researchers have worked on the problem of how to make monitoring non-intrusive, but they have found no solution that is perfect for all environments. Completely nonintrusive monitoring can only be done with hardware probes that identify program events either by listening to the bus or using a mirrored processor.

Finally, since time is recorded after the event, we must guarantee that there is no...
interruption between the event, reading the clock, and writing the record. Many Unix systems fail to meet this constraint, because any system call (including gettimeofday), which reads the clock, causes the program to return to the ready queue, which may let another program be run.

**USING THE SYSTEM**

We used Flex to study the performance of certain objects in the Gnu C++ class library. The class we chose to study was the AVLSet class, which represents ordered sets of items as AVL trees. This class uses recursive functions internally, so it is not amenable to program analysis. Because individual tree operations were too fine-grained for our machine’s 20-ms clock, we measured operation groups.

**Timing measurement.** In our study, each program cycle:

- preloaded the tree with \( n \) random elements,
- inserted 1,000 more elements into the tree and measured the time to do so,
- deleted the 1,000 elements from the tree and measured the time to do so, and
- performed 1,000 successful searches for elements in the tree and measured the time to do so.

Each AVL tree operation is expected to take \( O(\log n) \) time, so we describe the time for each of the three measured steps with the model

\[
\text{#pragma measure mean duration defining } A, B, C \text{ in } A \log(n) + Bn + C
\]

(We added the linear term to show how the system copes with extraneous terms in the model.)

Figure 3a-c shows the results of the measurement for each of the three functions performed by the study program: insertion, deletion, and search. The curves show the fitted function

\[
t = A \log n + Bn + C
\]

The functions appear visually to be an excellent fit, and this appearance is borne out by the chi-square statistics, which show a confidence level of better than 99 percent.

Table 1 shows the model parameters as determined by the system. The very small linear term shows that the analysis rejected the hypothesis that there is a linear component to the timing behavior.

**Measurement report.** Figure 4 shows the measurement report for the AVL insert operation. The first line shows the source file name and line number where the \#pragma measure directive being analyzed appears. This line is followed by a line giving the parameters determined to give the best fit.

The next line states the reason for terminating the iteration that fits the curve. In this case, the iteration terminated because it reached a flat point—no direction of further improvement could be found. This situation usually indicates that a minimum was reached successfully.

Other possible reasons for terminating the iteration are:

- An iteration step did not change the model parameters by a significant amount.
- An iteration step did not improve the fit significantly.
- A built-in limit on the number of iterations was exceeded. This situation

---

**TABLE 1**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Log n coefficient</th>
<th>( n ) coefficient</th>
<th>Constant term</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert</td>
<td>0.0159</td>
<td>( 6.20 \times 10^{-7} )</td>
<td>0.2357</td>
</tr>
<tr>
<td>Search</td>
<td>0.0088</td>
<td>(-1.26 \times 10^{-7} )</td>
<td>-0.0001</td>
</tr>
<tr>
<td>Delete</td>
<td>0.0209</td>
<td>( 6.06 \times 10^{-7} )</td>
<td>0.3012</td>
</tr>
</tbody>
</table>
usually indicates that the curve-fitting procedure is lost in a valley of complex topology, and it means you can't trust the results of the fit. Sometimes restarting the analysis can find a better fit.

After the reason for termination, the report shows the level of confidence in the model, both as a chi-square statistic and as a confidence limit. In this case, the confidence exceeds 99.99 percent. A confidence this high indicates that the experimental error in the data points has been underestimated. In this case, the error has definitely been underestimated owing to the granularity of the clock that was used to make the measurements.

Finally, the table gives a parameter set that can help you estimate how sensitive the fit is to the model parameters. The first line gives the length scale of the problem — the relative amount that a parameter has to change to increase chi-square by a given amount.

The rest of the table gives the singular values and right singular vectors, each of which represents a linearly independent direction in which the parameters can be perturbed. So, if $p_i$ is the model's $i$th parameter, $s_i$ its $i$th scale factor, $v_i$ the $i$th component of a singular vector, and $w$ is its
corresponding singular value, then replacing each $p_j$ with $p_j + (c_0 \mu_j)/\mu$ will increase the model's chi-square by 1. Therefore, a small singular value indicates a direction in which the model is relatively insensitive to perturbations, and therefore fails to distinguish among two or more of the parameters.

**Graphical report.** The measurement report in Figure 4 is not easy to understand. It is better to give the programmer a graphical presentation of the expected behavior if the fitted parameters differ from their actual values.

David Jackson developed a technique that perturbs each singular vector and examines its behavior. Figure 5 shows the results of this kind of perturbative analysis on our tree insertion example. Figure 5a shows the result of perturbing the first singular vector from the table in Figure 4, which represents uncertainty in the overall weights of the parameters. Figure 5b shows a perturbation of the second singular vector and represents a trade-off between the linear term and the rest of the model. Figure 5c shows a trade-off primarily between the constant term and the logarithmic term. In all three graphs, there is a 95-percent confidence that the line in Figure C.

Visual inspection or a simple calculation shows that insertion sort is better than quick sort by a factor of about 3 for short lists. The quick sort gradually gains on the insertion sort, however, and is superior on lists from about 200 numbers to lists of tens of thousands of numbers, when it is better by several orders of magnitude.

```c
for (apn = 10; t & apn <= 99999; apn *= 2.1544347) {
    n = (int) apn;
    // Show the start of the trial
    cerr << "Starting trials of sorting " << n << " elements.\n";
    cerr << "The sort function chosen will be " << sort_choice (n) << "\n";
    // Do as many sort operations as can be done in 100 seconds
    constraint_block (duration <= 100.) {
        #pragma objective minimize duration
        {
            for (i = 0; i < n; ++i)
                x [i] = random (0);
            sort (x, n);
        }
        cerr << " did " << t << " trials.\n";
    }
}
```

**Figure C.** Times measured for the two sort functions in Figure A. Dots are mean times for the insertion sorts, solid line is the quadratic function that best fits the measured data, diamonds are the times for the unmeasured quick-sort function, and the broken line is the expected runtime.
actual function lies between the upper and lower broken lines.

Model confidence. Some programmers may object to having only a statistical measure of confidence as opposed to the results of a program analysis, so let's examine the principal sources of error.

First, the model may fail to describe reality. This is a potential problem with any scheme, whether it is based on measurements or on program analysis. With program-analysis schemes, there is no way to validate the underlying assumptions embedded in the analysis program's hardware model. However, the statistics provided by the measurement-based scheme quantify its goodness of fit.

To check this assertion, you can analyze the insertion-sort program when it is given an incorrect assertion that the time required is $An^2 + n + B$. The confidence level corresponding to the calculated chi-square statistic drops from 0.999 to $10^{-13}$ — a clear indication that something is wrong with the model.

A second source of error is that the observed data on which the model is based may not be realistic (in a worst-case scenario, the measurement cases may have failed to uncover the actual worst case). Flex addresses this problem by letting the programmer specify a safety factor that allows for either a worst case that is known theoretically or a margin of error. Future research may develop techniques to verify theoretically that the worst case has been found, analogous to techniques that verify that a test set has exercised all the code.

A third source of error is the possibility that the measurement process interferes with a program's timing behavior. We have found this interference to be negligible for the coarse-grained program units we have studied — a measurement typically takes at most a few hundred microseconds to observe and record operations that take from milliseconds to seconds to complete.

Of course, when measurements must be extremely fine-grained, this objection is relevant. Our analysis scheme will still be valuable, since it can accommodate any measurement technology, including a circuit-level probe or other hardware monitor when high-accuracy, fine-grained measurements are required.

INTEGRATING MEASUREMENT AND ANALYSIS

No matter how well the measurement system performs, there will always be a place for formal program analysis because the confidence it provides is too important to ignore. However, some program components may not be amenable to formal analysis, and a designer might be willing to accept the statistical confidence provided by the modeling system. There must be a way to integrate the two.

Expanding program analysis. Program-analysis systems work by partitioning a program into basic blocks, calculating the time each basic block requires, and using a representation of the program's control structure — combining these time requirements into time requirements for successively higher level code blocks.

It is easy to expand program analysis to take measurements into account. A code block whose running time is measured can be treated as a basic block for the purpose of timing analysis. In addition, a code block whose completion time is enforced by the constraint system can be treated as a unit. There are then four ways to determine a block's running time:

- Analyzing the basic blocks.
- Measuring the timing of the code.
- Determining the time for one iteration of a loop with one of the first two techniques and then measuring the number of iterations with the #pragma measure count facility, which measures the number of trips through a loop.
- Specifying a maximum time for the block.

Once you have the block times and loop counts from any of these techniques, it is easy to combine them into an overall time estimate.

Measuring resources. Other resources can be handled similarly, although the rules for combining them may be different. It is best to classify resources according to these combination rules. Our major resource classes are time-like and

<table>
<thead>
<tr>
<th>Construct</th>
<th>Space-like</th>
<th>Time-like</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a : b$</td>
<td>max($c_a, c_b$)</td>
<td>$c_a + c_b$</td>
</tr>
<tr>
<td>loop a end</td>
<td>$c_a$</td>
<td>$n \cdot c_a$</td>
</tr>
<tr>
<td>if($c \rightarrow a$ else b)</td>
<td>max($c_a, c_b, c_c$)</td>
<td>max($c_a$, $c_b$)</td>
</tr>
<tr>
<td>cobegin $a, b$ coend</td>
<td>$c_a + c_b$</td>
<td>max($c_a$, $c_b$)</td>
</tr>
</tbody>
</table>

$n =$ number of times loop will be executed

Figure 5. Results of perturbing (A) the first singular vector from the table in Figure 1; (B) the second singular vector; (C) trade-off between the constant term and logarithmic term. Dots are observed values, solid lines best fit, and broken lines 95-percent confidence intervals.
space-like resources.

A time-like resource is a resource that, like time, is never reused once it is consumed. A space-like resource is a physical object that, once used by a task, can be reused by another task. Space-like resources include memory, disk space, processing elements, and critical sections.

To combine the resource use of two operations that use time-like and space-like resources, the only operations you must consider are summation, taking a maximum, and multiplication by a constant factor. Table 2 shows how the two resource types are combined under common language constructs. For example, for the statement

\[
\text{if}(c) \rightarrow a \text{ else } b
\]

the worst-case time to execute is the worst-case time needed to evaluate \( c \), plus the greater of the worst-case times needed to execute \( a \) and \( b \). The amount of dynamically allocated storage is simply the greatest amount of storage required to execute \( a \) or \( b \) or evaluate \( c \).

Combining resource use this way is like Alan Shaw's method for time-like resources; the time-like column in Table 2 shows the same mile that appears in his work. The space-like column is unique to our model.

One reason we resort to program measurement is that program-analysis tools are incomplete because they do not capture the programmer's knowledge about a program's timing behavior.

Peter Puschner and Christian Koza have proposed extending a real-time language with constructs that let the programmer specify things like the maximum trips through one part of a conditional construct within a loop, and the maximum iterations conducted by a loop sequence.

These are the first steps toward integrating the programmer's knowledge into program analysis as well as measurement. These concepts might even be extended to handle other kinds of programmer knowledge like the knowledge that the total iterations of the inner loop of a pair of nested loops might be bounded.

While the space- and time-like resource categories cover most real cases, they are not the only resource type that might be handled. In the Concord project, another resource considered is precision — a computation may be able to terminate in less time if it is allowed to return results that are less precise.

Unfortunately, precision is harder to analyze than space and time. You must characterize the sensitivity of later calculations to imprecisions in the earlier results. For example, in a simple sequence \( (A; B) \), the imprecision in \( B \)'s results can come from \( B \)'s inherent imprecision, or from imprecision in the intermediate results from \( A \). More work is needed to devise a simple way to characterize the effect that various operations have on the precision of results before we can analyze precision as we can space and time.

ACKNOWLEDGMENTS

This research was sponsored by the Office of Naval Research under grant N00014-89-J-1181, by the National Aeronautics and Space Administration under grant NASA-NAG-1-673, and by the National Science Foundation under grant CCR-89-11773.

REFERENCES