Estimating Timing Profiles for Simulation of Embedded Systems

Andrew H. Chan
Extended Abstract
EECS 290N
May 15, 2009
andrewhc@eecs.berkeley.edu

Abstract

Software simulation of embedded systems is generally less expensive and offers more flexibility than direct testing on hardware. In embedded systems, it is necessary to consider the concurrency and timing properties of the target system that are not captured by the sequential execution of its code. To account for the timing properties of the target system, the simulator must be able to estimate the timing behavior of the system under simulation. This paper proposes a method to efficiently extract a timing profile from the target system that allows the simulator to accurately estimate the timing characteristics of the embedded system.

1. Introduction

Software simulation of embedded systems provides several advantages over direct hardware testing. First, it is usually less expensive to run many trials in simulation rather than running tests directly on the hardware, especially if the intent is to perform many thousands of trials. Second, it offers more flexibility because different testing environments can be constructed and applied more efficiently in software. However, in embedded systems, it is necessary not only to account for the functional characteristics of the target system but also the concurrency and timing properties not captured by the sequential execution of its code.

Running a *functional* software model of the target system on a general-purpose machine will not generally yield the same timing behavior as the actual hardware because the machine running the simulation has completely different hardware specifications from the system it simulates. This paper proposes a method to extract and encode the timing profile of a target system, given access to the system, such that a pure software simulation can use the timing profile to accurately estimate the target system's timing characteristics.

2. Related Work

In the *Access Point Event Simulation of Legacy Embedded Software Systems* (APES-LESS) project [7], Resmerita presents an approach for efficient simulation of software models mapped to platform abstractions. Any line of the embedded source code containing an I/O access is considered an *access point*, where process control is returned to the APES engine to compute the next set of environment variables or to analyze the current simulation state. The APES simulation engine uses the discrete-event domain in the Ptolemy II framework [2], where the time stamps of the access point events are computed from the estimated execution times between consecutive access points. Considering that many target platform tasks may be executing concurrently, it is imperative that the time stamps be accurate to effectively model the concurrent behavior of the target system.

Seshia in [9] describes a method to estimate the worst-case execution time of a target platform by modeling the estimation problem as a game between the estimation algorithm and the environment. The estimation algorithm attempts to find the longest path through the program while the environment sets environmental parameters to thwart it. The technique relies on computing a set of basis paths through the *control flow graph* (CFG) of the program that is a good "representative" set of all the other paths in the CFG. The algorithm then proceeds to sample these paths, and uses the limited timing data it obtains to estimate the worst-case execution time and path. Many of the techniques in this paper are borrowed from [9], and the relevant aspects are described in the next section.

3. Methodology

3.1. Timing Model

In this section, an overview of the timing model is presented. We first define the control flow graph, and then de-

scribe its connection with timing estimation.

The *control flow graph* (CFG) of a program is a representation of the program's code as a weighted directed graph, where each node represents a statement in the code, and its outgoing directed edges represent the statements that immediately follow it. In most cases, a statement simply leads to the next. However, a conditional statement, such as an `if` or `switch` statement, may have more than one outgoing edge. The edge taken by the *control flow* of a particular *execution* of the program depends on the result of the conditional statement.

Consider the following simple program:

```
void func(int x, int y) {
    if(x > y) {
        x = 3;
    }
    else {
        y = x;
    }
    return;
}
```

Figure 1 shows the CFG of the above program, with the nodes labeled by their corresponding statements and the edges labeled by the conditions required to traverse them.

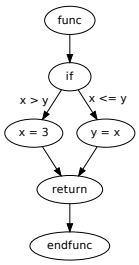

Figure 1. Control flow graph of a simple program.

In this model, we assume that all loops are unrolled so that the CFG is acyclic, and an execution of the program corresponds to a path from the start node (source) to the end node (sink). A *basic block* is a sequence of statements with no conditionals and where only the first node of the basic block may have more than one incoming edge. In other words, not only does a basic block execute sequentially from start to finish, but the execution of a program can never jump to the middle of a basic block from outside the block. Using this definition of a basic block, the CFG can be simplified by compressing all the statements in a basic block to a single node.

The time required to execute a basic block is represented in the CFG by the weight of the outgoing edge from its corresponding node. If the node has more than one outgoing edge, such as the node for an `if` statement, then the time to execute the conditional statement depends on the outcome of its predicate. This is reasonable because an `if` statement generally requires a jump if it evaluates to true but simply falls through to the next statement if it evaluates to false. Although this simplified timing model does not completely reflect every aspect of realistic timing behavior, it serves as a basis for abstracting the timing analsis, upon which additional details can be added.

3.2. Algorithm

In this section, the main algorithm of the paper is described. The algorithm consists of running the program with specialized inputs that drive the execution down particular paths in the CFG. The timing information for each of these executions is recorded and used to interpolate the execution times for all the other paths.

The algorithm operates over a series of rounds. In each round, the algorithm provides inputs to the target system that drive the execution along a particular path. To find these inputs, the algorithm uses a satisfiability solver on the constraints that the program's conditional statements impose.

The problem can be formulated as follows. Every round, the weights on the edges change, and the algorithm is only allowed to try one path through the CFG and measure its execution time, corresponding to the total weight of the path. After T rounds, the algorithm must construct an estimate of the average edge weights over all T rounds. Note that the algorithm only receives the end-to-end time of the execution path, not the time required for each edge. This limited timing data from each trial is the obstacle that the algorithm attempts to circumvent.

3.2.1 Path Space

In order for the sampling procedure to be effective, the algorithm must choose its sampling set carefully. In addition, the number of paths that can be tested should be polynomial in the number of edges of the CFG. There are an exponential number of paths from the source to the sink, which makes testing every possible path infeasible.

Every path through the CFG can be represented by an m-bit vector, where m is the number of edges in the graph. If a path traverses an edge, the bit corresponding to the edge is set to 1. Bits corresponding to edges that are not traversed are set to 0. These m-bit vectors form a subspace of dimension $m - n + 2$, where n is the number of nodes in the graph. The dimension is $m - n + 2$ because the space of paths consists only of vectors that satisfy flow conservation at every vertex except the source and the sink. That is, a path through the graph can be viewed as a flow, where every unit of flow that comes into a node must leave the node,

removing a degree of freedom per node. Thus, the dimension is the number of edges minus one per node, except for the source and the sink, $(m - (n - 2))$.

A subspace of dimension $b = m - n + 2$ is spanned by a basis set of b vectors. However, rather than choose an arbitrary basis set, the algorithm will choose a special basis set where every path can be represented as a linear combination of basis paths with coefficients at most 2. This special basis set is called a 2-barycentric spanner. Intuitively, a barycentric spanner is a good "representative" set of all the other paths in the graph since every path requires at most 2 times any of the basis paths. It is analagous to how two perpendicular vectors span $\mathbb{R}^2$ "more effectively" than two nearly parallel vectors. Barycentric spanners were first introduced in [1] by Awerbuch and Kleinberg, in which they provide an efficient algorithm to compute a barycentric spanner over a given subspace. Using a barycentric spanner ensures that the subset of paths that are sampled effectively capture the stochastic distribution of the delays on the edges and accurately represent all other paths in the CFG.

3.2.2 Example

We now present an example of how paths can be represented by vectors, and how they can be constructed by linear combinations of others paths. Each component of the vector represents an *edge* of the graph.

Consider the CFG in Figure (2). In this CFG, each of the edges is labeled with a number. Each component in an edge vector represents the corresponding edge. For example, the first component represents edge 1, the second represents edge 2, and so on.

The vector $\mathbf{x} = (1\ 0\ 1\ 0\ 1\ 0\ 1\ 0)$ represents the path that goes from a to b to d to e to g. The vector $\mathbf{y} = (0\ 1\ 0\ 1\ 0\ 1\ 0\ 1)$ represents the path that goes from a to c to d to f to g. The vector $\mathbf{z} = (1\ 0\ 1\ 0\ 0\ 1\ 0\ 1)$ represents the path that goes from a to b to d to f to g. These vectors form a basis set for the CFG. Another path in the CFG is $\mathbf{u} = (0\ 1\ 0\ 1\ 1\ 0\ 1\ 0)$, corresponding to a, c, d, e, g. This path can be written as a linear combination of $\mathbf{x}$, $\mathbf{y}$ and $\mathbf{z}$ as follows: $\mathbf{u} = \mathbf{x} + \mathbf{y} - \mathbf{z}$. Note that in general the coefficients for the basis vectors are not restricted to 1 and -1. They could potentially be very large in more complex CFGs unless the right set of basis paths is selected.

The main steps of the algorithm will now be described. Let B be a matrix where each row is a basis vector. The ordering of the basis vectors as rows in B can be arbitrary. Call the first row of B basis vector 1, the second row basis vector 2, and so on. From the example above, the matrix would be

$$B = \begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$

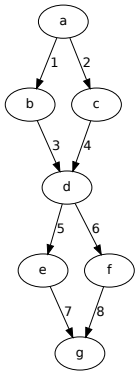

Figure 2. Control flow graph.

Notice that the number of basis paths, b, is less than m, the number of edges. Hence, B is not a square matrix (its dimensions are $b \times m$). Let B^+ denote its pseudoinverse, with dimensions $m \times b$.

The algorithm is as follows. It is a simplified version of the GAMETIME algorithm in [9].

1. Compute a 2-barycentric spanner over the subspace of paths through the CFG.
2. Construct B and compute B^+.
3. In each round from $t = 1$ to T:
   (a) Choose a basis path uniformly at random. Call it basis vector i.
   (b) Measure the time of execution c_t for the chosen basis path.
   (c) Let $\hat{\mathbf{w}}_\mathbf{t}$ equal the ith column of B^+ multiplied by bc_t (recall that b is the number of basis vectors). $\hat{\mathbf{w}}_\mathbf{t}$ is the estimated weight vector for round t.
4. Compute $\overline{\mathbf{w}} = \frac{1}{T}\sum_{t=1}^{T} \hat{\mathbf{w}}_\mathbf{t}$ (the average of the estimated weight vectors over all the rounds).

The result $\overline{\mathbf{w}}$ is an m-vector, each component containing the estimated weight for its corresponding edge.

These edge weights can now be used to construct the estimated path length of any path through the CFG, even those that were not directly sampled by the algorithm. With the appropriate assumptions, it can be shown that these estimated edge weights are close to the true edge weights with high probability. A necessary assumption is that the maximum time for any basis path in any round is bounded by M. This limits how significantly the edge weights can vary from round to round.

An intuitive explanation of how the algorithm works is as follows. Suppose in every round, the algorithm were allowed to try every basis path instead of only one. Then it

would be straightforward to calculate the estimated weights on the edges for round t. It would only require taking the inverse linear transformation to obtain edge weights from execution times. Specifically, for round t, let $\mathbf{l_t}$ be a b-vector with the time for each basis path in the corresponding component. The algorithm would then compute $B^+\mathbf{l_t} = \hat{\mathbf{w}}_\mathbf{t}$. Any error would stem from the fact that B^+ is only the pseudoinverse of B rather than the true inverse. However, this would be the best one could hope for in estimating the edge weights for round t given end-to-end timing data.

Now returning to the original situation, the algorithm is only allowed to test one basis path per round. To compensate for only sampling one basis path, it "normalizes" (multiplies) the time required for the basis path by b to make up for the fact that this basis path is only sampled once every b rounds in expectation. Being able to sample only one basis vector in every round is potentially a significant source of error. However, by showing that the sampling procedure results in an unbiased estimator of the edge weights over the T rounds, we can deduce that in expectation, the estimated edge weights won't be too far off from the actual edge weights.

3.2.3 Example

Returning to the example from above, the pseudoinverse of B is

$$B^+ = \begin{pmatrix} .125 & -.125 & .25 \\ .125 & -.375 & -.25 \\ .125 & -.125 & .25 \\ .125 & .375 & -.25 \\ .375 & .125 & -.25 \\ -.125 & .125 & .25 \\ .375 & .125 & -.25 \\ -.125 & .125 & .25 \end{pmatrix}$$

Suppose that over T rounds, the average time for basis path 1 is 12 seconds, the time for basis path 2 is 40 seconds, and the time for basis path 3 is 18 seconds. Then $\hat{\mathbf{w}}$, the estimated weights vector, would be

$$\hat{\mathbf{w}} = B^+ \begin{pmatrix} 12 \\ 40 \\ 18 \end{pmatrix} = \begin{pmatrix} 1 \\ 12 \\ 1 \\ 12 \\ 5 \\ 8 \\ 5 \\ 8 \end{pmatrix}$$

This estimated weights vector is then used to estimate the path lengths for the other paths. For example, if we wanted to estimate the path length of $\mathbf{u}$ from the previous example, we would take the dot produce of $\mathbf{u}$ and $\mathbf{w}$, which in this case is 34 seconds.

Due to the lack of space, a formal bound on the error of the algorithm is deferred to the Appendix.

4. Experiments

The implementation of the proposed timing estimation method builds on CREST [3], which is closely based on CUTE [8]. CREST is a branch coverage testing tool, designed to verify general-purpose programs written in C. It uses CIL [5] to process and instrument the C code. CIL also extracts the CFG, which is used by CREST to generate the inputs for program execution. CREST uses Yices [4], a satisfiability solver developed by SRI, to find satisfying assignments to conditionals. None of the aforementioned programs address timing issues in their verification processes, instead focusing on bugs and reachability properties.

The linear algebra manipulations required in the timing estimation make use of the Numpy library in the Python programming language.

The experiments were done on fragments of code from the open source PapaBench [6] software for an unmanned aerial vehicle. The first is called "Altitude" and the second is called "Climb Control." The trials were performed using SimIt-Arm 2.1 [10], a cycle accurate simulator of the StrongARM architecture.

The results of the experiments are shown in Table 1.

	Altitude	Climb Control
Nodes	15	50
Edges	19	66
Basis Paths	6	18
Paths	11	657
Non-basis Paths Tested	5	94
Mean (cycles)	794	1178
Std Dev	54%	35%
Avg Est. Diff.	0.9%	2.5%
Max Est. Diff.	1.62%	12.7%

Table 1. Results of the experiments.

The experiments show that for small programs, the timing estimation method is accurate to well within the standard deviation over all the paths. The standard deviation above is the variation among all feasible paths through the program, and the mean is the average number of cycles over the paths.

Because the experiments were done on a simulator, the execution time for a given input was deterministic. However, the algorithm as specified in Section 3 can be applied when the execution times are nondeterministic. That is, given a fixed input that drives the execution down a particular path, the execution time can vary. An extension to the experiments is to prepare the processor in a random state

so that the execution time for a given path varies even on the same input. For example, the state of the cache when execution begins can change the path length from round to round even for a fixed input.

5. Conclusion

This paper presented a method to extract the timing profile from a target platform running a specified program. It differs from [9] in that it focuses on constructing a timing profile for a portion of code, rather than aiming for the worst-case execution time of an entire program. In addition, the timing profile has leeway for some error, as its goal is not to find one correct answer (e.g. the worst case execution time) but instead to provide an efficient, albeit less accurate, alternative to directly testing on hardware. Possible future improvements include augmenting the timing model to account for more timing effects.

6. Appendix

We formally bound the error of the algorithm. Let $\mathbf{w}^*$ be an m-vector denoting the true average weights of the edges over rounds 1 to T.

Theorem 6.1 *With probability at least $1-\delta$, for all paths $\mathbf{x}$ in the CFG,*

$$\left|(\overline{\mathbf{w}} - \mathbf{w}^*)^T \mathbf{x}\right| \leq \frac{2bM}{T^{\frac{1}{2}}}\sqrt{2\ln(2b/\delta)}$$

That is, with high probability, the difference between the estimated average path time and the actual average path time is bounded by an expression that approaches 0 as T, the number of rounds, goes to infinity.

Proof: Let i_t denote which basis vector was sampled in round t. Let $\mathbf{d_t}$ be a b-vector with bc_t in the i_tth component and 0 elsewhere. Let $\mathbf{l_t}$ be a b-vector whose ith component contains the execution time for basis path i in round t.

$\mathbf{d_t}$ is the algorithm's observation in round t. It has only one non-zero component. $\mathbf{l_t}$ contains all the execution times for the basis paths in round t (only one of which the algorithm was able to observe).

Then $\mathbf{e}_\tau = \sum_{t=1}^{\tau}(\mathbf{d_t} - \mathbf{l_t})$ is a vector whose ith component contains the total error of the algorithm's estimate of the execution time of basis path i over the past τ rounds.

We first need to show that every component of $\mathbf{e}_\tau$ is a bounded martingale sequence with respect to the filter $(i_1, i_2, \ldots, i_{\tau-1})$ (the random choices of the algorithm). That is, we need to show that for every component j of $\mathbf{e}_\tau$, $\mathbb{E}(e_{j,\tau} \mid i_1, i_2, \ldots, i_{\tau-1}) = e_{j,\tau-1}$ and that $|e_{j,\tau} - e_{j,\tau-1}|$ is bounded. Then we can apply Azuma's inequality to compute a concentration bound, which leads to the theorem's result.

It is easy to see that $e_{j,\tau} = e_{j,\tau-1} + d_{j,\tau} - l_{j,\tau}$.

Then observe that $\mathbb{E}(d_{j,\tau}) = l_{j,\tau}$, because in every round, the algorithm uniformly chooses a basis path to sample. Thus, $\mathbb{E}(e_{j,\tau} \mid i_1, i_2, \ldots, i_{\tau-1}) = e_{j,\tau-1}$ by linearity of expectation.

Next we need to bound $|e_{j,\tau} - e_{j,\tau-1}|$. Notice that the most $e_{j,\tau-1}$ can change in one round is $|d_{j,\tau} - l_{j,\tau}|$.

$d_{j,\tau} = l_{j,\tau}b$ (the execution time for basis j multiplied by b). Thus, $|d_{j,\tau} - l_{j,\tau}| \leq bM$, since we assume that the execution time for any basis path is bounded by M.

Now we apply Azuma's Inequality to arrive at

$$\Pr\left(e_{j,T} > bM\sqrt{2T\ln(2b/\delta)}\right) \leq \frac{\delta}{b}$$

This holds for any single basis path j. Taking the union bound over all b basis paths, we find an upperbound for the probability that all the basis paths satisfy the above probability.

$$\Pr\left(\forall \text{ basis paths } j, e_{j,T} > bM\sqrt{2T\ln(2b/\delta)}\right) \leq \delta$$

Having bound the error for the basis paths, we now use the properties of the barycentric spanner to bound the error on the rest of the paths.

Because any path can be written as a linear combination of the basis paths with coefficients less than or equal to 2, in the worst case, the probability that the error of any path is greater than $2bM\sqrt{2T\ln(2b/\delta)}$ is negligible.

Let $\mathbf{x} = B^T\mathbf{k}$. The total error in the estimation of path $\mathbf{x}$ over T rounds is then $|\mathbf{e_T} \cdot \mathbf{k}|$. Hence, for all paths $\mathbf{x} = B^T\mathbf{k}$,

$$\Pr\left(|\mathbf{e_T} \cdot \mathbf{k}| > 2bM\sqrt{2T\ln(2b/\delta)}\right) \leq \delta$$

Note, however, that $|\mathbf{e_T} \cdot \mathbf{k}| = T(\overline{\mathbf{w}} - \mathbf{w}^*)^T\mathbf{x}$. Thus, the theorem follows by dividing the inequality in the probability by T. ■

References

[1] B. Awerbuch and R. D. Kleinberg. Adaptive routing with end-to-end feedback: distributed learning and geometric approaches. In *STOC '04: Proceedings of the thirty-sixth annual ACM symposium on Theory of computing*, pages 45–53, New York, NY, USA, 2004. ACM.

[2] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng. Heterogeneous concurrent modeling and design in java. Technical Report Technical Memorandum UCB/ERL M04/27, University of California, July 29 2004.

[3] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. Technical Report UCB/EECS-2008-123, EECS Department, University of California, Berkeley, Sep 2008.

[4] B. Dutertre and L. de Moura. The yices smt solver. Tool paper at http://yices.csl.sri.com/tool-paper.pdf, August 2006.

[5] G. C. Necula, S. Mcpeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 213–228, London, UK, 2002. Springer-Verlag.

[6] F. Nemer, H. Cass, P. Sainrat, J. P. Bahsoun, and M. D. Michiel. Papabench: a free real-time benchmark. In F. Mueller, editor, *WCET*, volume 06902 of *Dagstuhl Seminar Proceedings*. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2006.

[7] S. Resmerita and P. Derler. The apes-less project: Access point event simulation of legacy embedded software systems. April 2009. Presented at the 8th Biennial Ptolemy Miniconference.

[8] K. Sen, D. Marinov, and G. Agha. Cute: a concolic unit testing engine for c. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.

[9] S. A. Seshia and A. Rakhlin. Game-theoretic timing analysis. In *ICCAD '08: Proceedings of the 2008 IEEE/ACM International Conference on Computer-Aided Design*, pages 575–582, Piscataway, NJ, USA, 2008. IEEE Press.

[10] Simit-arm. http://simit-arm.sourceforge.net.