GameTime: A Toolkit for Timing Analysis of Software

Sanjit A. Seshia and Jonathan Kotker

EECS Department, UC Berkeley
{sseshia,jamhoot}@eecs.berkeley.edu

Abstract. Timing analysis is a key step in the design of dependable real-time embedded systems. In this paper, we present GameTime, a toolkit for execution time analysis of software. GameTime is based on a combination of game-theoretic online learning and systematic testing using satisfiability modulo theories (SMT) solvers. In contrast with many existing tools for timing analysis, GameTime can be used for a range of tasks, including estimating worst-case execution time, predicting the distribution of execution times of a task, and finding timing-related bugs in programs. We describe key implementation details of GameTime and illustrate its usage through examples.

1 Introduction

Timing properties of embedded systems are determined by the behavior of both the control software and the platform the software executes on. The verification of such properties is made difficult by their heavy dependence on characteristics of the platform, including details of the processor and memory hierarchy.

Several kinds of timing analysis problems arise in practice. First, for hard real-time systems, a classic problem is to estimate the worst-case execution time (WCET) of a terminating software task. Such an estimate is relevant for verifying if deadlines or timing constraints are met as well as for use in scheduling strategies. Second, for soft real-time systems, it can be useful to estimate the distribution of execution times exhibitable by a task. Third, it can be very useful to find a test case on which the program exhibits anomalous timing behavior; e.g., a test case causing a task to miss its deadline. Finally, in "software-in-the-loop" simulation, the software implementation of a controller is simulated along with a model of the continuous plant it controls, with the simulations connected using execution time estimates. For scalability, such simulation must be performed on a workstation, not on the target embedded platform. Consequently, during the workstation-based simulation, it is necessary to predict the timing of the program along a particular execution path on the target platform.

All of the problems mentioned in the preceding paragraph are instances of *predicting* a particular execution time property of a terminating software task. In this paper, we present GAMETIME, a toolkit for timing analysis of software. In contrast with existing tools for timing analysis (see, e.g., [4]), GAMETIME can predict not only extreme-case behavior, but also certain execution time statistics (e.g., the distribution) as well as a program's timing along particular execution paths. Additionally, it is measurement-based, making it easy to port to new platforms. The GAMETIME approach, along with an exposition of theoretical and experimental results, including comparisons with other

P.A. Abdulla and K.R.M. Leino (Eds.): TACAS 2011, LNCS 6605, pp. 388–392, 2011.

methods, is described in existing papers [5,6]. The goal of this paper is to describe the overall tool flow along with aspects of the implementation not described in detail in those papers. We also illustrate, with a running example, how GAMETIME can be used to make various execution time predictions.

2 Running Example

We consider programs P where loops have statically-known finite loop bounds and function calls have known finite recursion depths. Thus P can be unrolled to an equivalent program Q where every execution path in the (possibly cyclic) control-flow graph of P is mapped 1-1 to a path in the acyclic control-flow graph of Q. Our running example is the modular exponentiation code given in Figure 1(a). Modular exponentiation is a necessary primitive for implementing public-key encryption and decryption. The unrolled version of this code for a 2-bit exponent is given in Figure 1(b).

```
1  modexp(base, exponent) {
2   result = 1;
3   for(i=EXP_BITS; i>0; i--) {
4   // EXP_BITS = 2
5    if ((exponent & 1) == 1) {
6     result = (result * base) % p;
7    }
8    exponent >>= 1;
9    base = (base * base) % p;
10  }
11  return result;
12 }
```

(a) Original code P

```
1  modexp_unrolled(base, exponent) {
2   result = 1;
3   if ((exponent & 1) == 1) {
4    result = (result * base) % p;
5   }
6   exponent >>= 1;
7   base = (base * base) % p;
8   // unrolling below
9   if ((exponent & 1) == 1) {
10   result = (result * base) % p;
11  }
12  exponent >>= 1;
13  base = (base * base) % p;
14  return result;
15 }
```

(b) Unrolled code Q

Fig. 1. Modular exponentation. Both programs compute the value of $\texttt{base}^{\texttt{exponent}}$ modulo p

3 The GAMETIME Approach

We begin with a brief overview of the approach taken by GAMETIME and a description of one of the core components, the generation of basis paths of a program (Sec. 3.1). Sec. 3.2 gives a sample experimental result on the running example described in Sec. 2.

Figure 2 depicts the operation of GAMETIME. As shown in the top-left corner, the process begins with the generation of the control-flow graph (CFG) corresponding to the program, where all loops have been unrolled to the maximum loop bound, and all function calls have been inlined into the top-level function. The CFG is assumed to have a single source node (entry point) and a single sink node (exit point); if not, dummy source and sink nodes are added. The next step is a critical one, where a subset of program paths, called *basis paths* are extracted. These basis paths are those that form a basis for the set of all paths, in the standard linear algebra sense of a basis. A satisfiability modulo theories (SMT) solver is invoked to ensure that the generated basis paths are feasible. We discuss this step in more detail in Sec. 3.1.

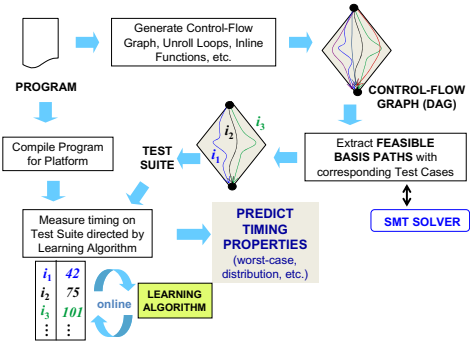

Fig. 2. GAMETIME **overview**

The basis paths are generated along with the corresponding test cases that drive execution down those paths. The program is then compiled for the target platform, and executed on these test cases. In the basic GAMETIME algorithm (described in [5,6]), the sequence of tests is randomized, with basis paths being chosen uniformly at random to be executed. The overall execution time of the program is recorded for each test case. From these end-to-end execution time measurements, GAMETIME's learning algorithm generates a weighted graph model that is used to make predictions about timing properties of interest. The predictions hold with high probability; see the previous papers on GAMETIME [5,6] for details.

In principle, GAMETIME can be set up to use any compiler front-end to generate the CFG and perform test generation along basis paths using an SMT solver; we have experimented with using both CIL [2] and the Microsoft Phoenix compiler front-end [1]. Similarly, any SMT solver for the combination of bit-vector arithmetic and arrays can be used. The core GAMETIME algorithms (involving linear algebra to generate basis paths and the learning algorithm) are implemented separately in Python.

3.1 Generating Basis Paths

In the CFG extracted from a program, nodes correspond to program counter locations, and edges correspond to basic blocks or branches.

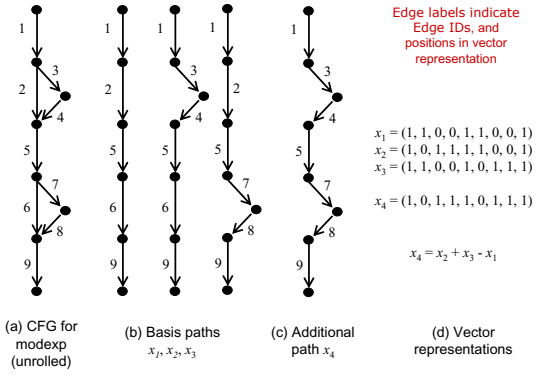

Fig. 3. CFG and Basis Paths for Code in Fig. 1(b)

Figure 3(a) denotes the control-flow graph for the code in Figure 1(b). Each source-sink path in the CFG can be represented as a 0-1 vector with m elements, where m is the number of edges. The interpretation is that the ith entry of a path vector is 1 iff the ith edge is on the path (and 0 otherwise). For example, in the graph of Fig. 3(a), each edge is labeled with its index in the vector representation of the path. For example, edge 2 and 3 correspond to the else (0th bit of `exponent` = 0) and then branches of the condition statements at lines 3 and 9

respectively in the code, while edge 5 corresponds to the basic block comprising lines 6 and 7. We denote by $\mathcal{P}$ the subset of $\{0,1\}^m$ corresponding to valid program paths. Note that this set can be exponentially large in m.

A key feature of GAMETIME is the ability to exploit correlations between paths so as to be able to estimate program timing along any path by testing a relatively small subset of paths. This subset is a basis of the path-space $\mathcal{P}$, with two valuable properties: any path in the graph can be written as a linear combination of the paths in the basis, and the coefficients in this linear combination are bounded in absolute value. The first requirement says that the basis is a good representation for the exponentially-large set of possible paths; the second says that timings of some of the basis paths will be of the same order of magnitude as that of the longest path. These properties enable us to repeatedly sample timings of the basis paths to reconstruct the timings of all paths. As GAMETIME constructs each *basis path*, it ensures that it is feasible by formulating and checking an SMT formula that encodes the semantics of that path; a satisfying assignment yields a test case that drives execution down that path.

Fig. 3(b) shows the basis paths for the graph of Fig. 3(a). Here x_1, x_2, and x_3 are the paths corresponding to `exponent` taking values 00, 10, and 01 respectively. Fig. 3(c) shows the fourth path x_4, expressible as the linear combination $x_2 + x_3 - x_1$ (see Fig. 3(d)).

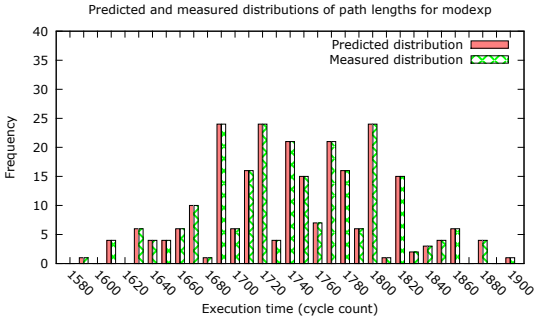

Fig. 4. Actual (striped) and Predicted (shaded) Execution Times for Code in Fig. 1.

The number of feasible basis paths b is bounded by $m - n + 2$ (where n is the number of CFG nodes). Note that our example graph has a "2-diamond" structure, with 4 feasible paths, any 3 of which make up a basis. In general, an "N-diamond" graph with 2^N feasible paths has at most $N + 1$ basis paths.

Computing tests for all basis paths can be viewed as a structural test coverage criterion. Covering all basis paths (for any basis) gives full statement and branch coverage, but not full path coverage. Also, generating tests for basis paths can be viewed as a way of exploiting the structure of the program's CFG for systematic test generation. We have found basis path coverage to be valuable for the prediction of timing properties of a program.

3.2 Sample Experimental Result

We used GAMETIME to estimate the distribution of execution times of `modexp` function for an 8-bit exponent (256 program paths) by testing only the (9) basis paths. The experiments were performed for the StrongARM-1100 processor – which implements the ARM instruction set with a 5-stage pipeline and both data and instruction

caches – using the SimIt-ARM cycle-accurate simulator [3]. Fig. 4 shows the predicted and actual distribution of execution times – we see that GAMETIME predicts the distribution perfectly. Also, GAMETIME correctly predicts the WCET (and produces the corresponding test case: `exponent=255`).

Acknowledgments. This work was supported in part by NSF grants CNS-0644436 CNS-0627734, and CNS-1035672, an Alfred P. Sloan Research Fellowship, and the Multiscale Systems Center (MuSyC), one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity. We also acknowledge the contributions of Andrew Chan, Sagar Jain, and Min Xu to the development of GAMETIME.

References

1. Phoenix software optimization and analysis framework, `https://connect.microsoft.com/Phoenix`
2. Necula, G., et al.: CIL - infrastructure for C program analysis and transformation, `http://manju.cs.berkeley.edu/cil/`
3. Qin, W., Malik, S.: Simit-ARM: A series of free instruction-set simulators and micro-architecture simulators, `http://embedded.eecs.berkeley.edu/mescal/forum/2.html`
4. Wilhelm, R., et al.: The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools. ACM Transactions on Embedded Computing Systems, TECS (2007)
5. Seshia, S.A., Rakhlin, A.: Game-theoretic timing analysis. In: Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD), pp. 575–582 (2008)
6. Seshia, S.A., Rakhlin, A.: Quantitative analysis of systems using game-theoretic learning. ACM Transactions on Embedded Computing Systems (TECS) (to appear)