Timing Analysis of Interrupt-Driven Programs under Context Bounds

Jonathan Kotker Dorsa Sadigh Sanjit A. Seshia
EECS Department, UC Berkeley
jamhoot@eecs.berkeley.edu dsadigh@berkeley.edu sseshia@eecs.berkeley.edu

***Abstract*— Timing analysis is a key step in the design of dependable real-time embedded systems, yet existing analysis tools do not work well for interrupt-driven code, which is commonly used in embedded systems. In this paper, we present a technique for timing analysis of interrupt-driven software. We show that for systems that use priority pre-emptive scheduling, if there is a finite arrival time between interrupts, one can use bounds on the number of context switches to perform timing analysis. Our work builds upon prior work on timing analysis for sequential programs. We present empirical evidence to show that we can accurately predict the execution time along any path of an interrupt-driven program on a standard micro-controller.**

I. Introduction

Timing is central to the correctness of real-time embedded systems. Timing properties 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. Even so, over the past two decades, there has been steady progress in the field of timing analysis for *purely sequential software* (see [1], [2]). Most of the progress has been on the classic problem of estimating the worst-case execution time (WCET) of a terminating software task. Such an estimate can be used as conservative checks on real-time constraints as well as for use in scheduling algorithms. While determining a bound on the WCET has many uses, it is not the only problem of interest. As tools typically overestimate the WCET, when the WCET exceeds the timing bound, one cannot be sure whether the program can really miss its deadline. One would also like to find a test case demonstrating that the program can miss its deadline. Recent methods [2] have sought to address this problem for sequential programs.

In practice, though, embedded software is not purely sequential. In many real-world applications, the control software comprises several tasks that execute concurrently. Programming with interrupts is an extremely common form of concurrency that the control software uses to obtain sensor data from its physical environment. Apart from a main function, the control software has one or more interrupt-service routines (ISRs). An ISR is invoked when its corresponding interrupt is raised, e.g., when a new sensor sample is available. For such an interrupt-driven program, there is a need to ensure that the task meets its deadline *even* in the presence of interrupts. However, the state-of-the-art of timing analysis for interrupt-driven software is extremely poor. For instance, in NASA's recent report on "unintended acceleration" in certain Toyota automobiles [3], several limitations of state-of-the-art timing analysis tools are noted, including the lack of support for interrupts.

The reason for this lack of progress on timing analysis of interrupt-driven software is not hard to guess. It is the exponential explosion in the number of interleavings of various software tasks (such as the main function and the ISRs for various interrupts). This path explosion especially impacts timing analysis, since timing is a *highly path-sensitive property* — the execution time of a basic block of a program can depend a great deal on the path it lies on. This is in contrast with verifying invariants (such as assertion violations), where one is concerned with checking if a particular "error" location is reachable without regard to how it is reached. Moreover, interrupts also impact processor operation, e.g., by flushing the CPU pipeline. Most current state-of-the-art WCET analysis techniques are based on using abstract interpretation to create an abstract timing model of the processor [1]. Even for sequential programs, the creation of an abstract timing model is an extremely tedious manual process. With interrupt-driven programs, the process is even harder due to the need to model the impact of interrupts on hardware and also due to the severe imprecision abstract interpretation suffers due to the large number of joins required on reconvergent interleaved paths.

Even with these challenges, good embedded software design often follows rules that can ease the problem. First, in many systems, there is a strict priority assignment between various tasks in the system, and the task scheduler follows *priority pre-emptive scheduling* — a task runs to completion unless a higher-priority task preempts it. Second, there is usually a *finite lower bound on inter-arrival time* between interrupts, dictated, for example, by the rate at which a sensor generates samples. This inter-arrival time bound imposes a restriction on how frequently a task can be interrupted. Finally, careful coding practices involve the use of "atomic sections" by disabling interrupts in selected parts of the program.

In this paper, we present a novel approach to the problem of timing analysis of interrupt-driven software that takes advantage of the above design rules. In particular, we make the following contributions:

- We show how a lower bound on inter-arrival time of

interrupts in turn imposes an upper bound on the number of "context switches" between the interrupted task and the ISR. This enables the use of *context-bounded* analysis, similar to the work pioneered by Qadeer et al. [4], [5]. The use of atomic sections and priority pre-emptive scheduling further reduces the number of interleavings that need to be considered.
- Even with these reductions, the number of interleaved paths can still be exponential in the context bound, and very large in practice. Obtaining measurements for a large number of paths can be very tedious and expensive. We show that we can leverage work for sequential program timing analysis to mitigate this problem. In particular, we adopt the idea of using the execution time of *basis paths* to predict the times of other program paths [2], [6]. The number of basis paths is guaranteed to be polynomial in the size of the program.
- We demonstrate our approach with experiments on a real embedded platform, the Luminary Micro LM3S8962 board with an ARM Cortex M-3 processor [7], interfaced to sensors on the iRobot Create mobile robot [8]. We show that we can accurately predict not only the WCET of various programs, but also the execution times of arbitrary program paths. When a particular deadline is violated, our approach can generate a test case exhibiting how this occurs.

To our knowledge, our approach is the first timing analysis technique for interrupt-driven software that can not only generate worst-case execution time estimates, but also can generate accurate predictions for the actual timing (not just bounds) along arbitrary program paths. Importantly, our approach is extremely *portable*: in contrast with traditional WCET techniques that rely on tedious manual modeling of the platform, our approach only requires automated systematic generation of measurements on the target platform, from which we make accurate predictions of program timing on paths that have not been tested.

The rest of the paper is organized as follows. We introduce the problem, along with basic terminology, definitions, and related work in Section II. The core of our approach is presented in Section III. Section IV presents an experimental evaluation. We conclude in Section V with directions for future work.

II. Background and Related Work

We define terminology and the problems considered in this paper in Section II-A, and compare with related work in Section II-B.

A. Problem Definition

Real-time embedded programs are reactive programs that execute repeatedly within a top-level "$\mathbf{while}(1)$ loop". We are concerned with the tasks invoked within this loop, which are required to be terminating programs. For this paper, we are concerned with programs structured as a single "main" task along with one or more interrupt-service routines (ISRs) which are written typically as other tasks (think of C functions). Typically, the boot-up sequence of the system involves registering the ISRs as handlers for the various interrupts that the system must respond to.

We present a simple imperative language to model these interrupt-driven programs. Figure 1 shows the program syntax. An interrupt-driven program P is composed of N tasks, each of which is a sequential program. Each task T has an associated *priority level* p, which is a positive integer. We will assume that each task has a unique priority level, and a larger priority level indicates higher priority. A task of priority p_i can interrupt a task with priority p_j if $p_i > p_j$. Once a higher-priority task has interrupted a lower-priority task, it runs to completion unless it is interrupted by a task with still higher priority. This scheduling scheme is known as *priority pre-emptive scheduling*, and is widely implemented in embedded platforms.

$$
\begin{array}{rl}
S ::= & v := e \mid \mathbf{skip} \mid \mathbf{if}\, e\, \mathbf{then}\, S_1\, \mathbf{else}\, S_2 \\
 & \mid S_1; S_2 \mid \mathbf{while}\, e\, \mathbf{do}\, \langle B \rangle\, S \\
 & \mid \mathbf{atomic}\,\{\, S \,\} \mid \mathbf{timed_while}\, \tau\, \mathbf{do}\, S \\
T ::= & \langle S, p \rangle \\
P ::= & T_1 \parallel T_2 \parallel \ldots \parallel T_N
\end{array}
$$

Fig. 1: **Syntax for Interrupt-Driven Programs.** v and e denote an l-value and an expression in any standard imperative programming language such as C. The $\mathbf{skip}$ statement is a no-op. Every $\mathbf{while}$ loop has an associated loop bound B. T denotes a sequential task with an associated priority p, and P denotes a program composed of n tasks.

The code for a task T follows standard syntax of an imperative language such as C, with a few small exceptions. Assignments have the form $v := e$ where v is an l-value and e is any expression in C including procedure calls. For simplicity, we disallow recursive procedure calls; in any case, it is highly desirable in real-time embedded software to impose finite bounds on recursion depth. The syntax of Fig. 1 includes **if** statements as a way of modeling all conditional constructs, including **switch** statements. We will use **switch** statements where required for brevity. The main exceptions to standard program notation are with regard to **while** loops and the presence of a special **atomic** program construct, as described below:

1) Each **while** loop must have a statically-known upper bound B on the number of loop iterations. We assume each loop is annotated with such a bound. We will use the standard **for**-loop notation where it is more convenient to do so.
2) There is a special timed-while loop construct **timed_while** which has an associated deadline τ. This loop runs for exactly τ cycles and terminates thereafter. This construct models timed loops common in embedded code that waits for an event for a specific amount of time, with termination guaranteed by the expiration of a hardware timer.

3) We include a special **atomic** construct which models a piece of code S that runs uninterrupted. This construct is typically implemented by disabling interrupts before running S and re-enabling interrupts after S completes execution. Using such atomic code sections within sequential code is considered good programming practice to ensure that certain operations are completed atomically irrespective of the presence of interrupts.

We assume that interrupts cannot occur infinitely often during the execution of P and that there is a finite lower bound on the inter-arrival times of interrupts. We believe this is a reasonable assumption that holds in practice for real-time embedded systems.

Given the above model, we are concerned with answering the following three types of timing analysis questions. For each question, the inputs include an interrupt-driven program P and the platform it executes on. The platform is the complete hardware and software environment of P, including the compiler, processor, and memory architecture.

- **P1: Threshold Property Checking.**
  Does P always complete within τ cycles? If not, provide a test case (counterexample).
- **P2: Worst-Case Execution Time Prediction.**
  Predict the worst-case execution time of P and generate corresponding test case.
- **P3: Predicting Timing along All Paths.**
  Predict the execution time (not a bound) of program P along *all* paths, where a path involves following a specific interleaving of tasks and particular paths within tasks.

One can observe that problem **P3** is more general than **P1** and **P2** in that if one can solve **P3**, one can answer questions **P1** and **P2** as well. Therefore, in Section III, we focus on addressing problem **P3**. We demonstrate our results for all three problems in Section IV.

Our technique relies on the notion of context-bounded analysis [4], [5]. Following the definition introduced by Qadeer and Rehof [5], a *context* is an uninterrupted sequence of actions by a single task. A bound of K on the number of contexts implies a bound of $K-1$ on the total number of context switches between tasks.

B. Related Work

As noted above, Qadeer et al. introduced the idea of verifying multithreaded software by using context bounds [4], [5]. However, their work focuses on traditional propositional temporal properties. Our paper is the first to apply the idea of context-bounded analysis to the problem of timing analysis.

Brylow and Palsberg [9] consider the topic of deadline analysis in interrupt-driven programs — checking whether every interrupt is serviced before its deadline. They assume that worst-case execution times are already determined for certain program fragments and use this in their analysis. In contrast, we are concerned with predicting execution time properties of the entire interrupt-driven program, and can generate the WCET estimates required in their analysis.

The WCET analysis community has mainly focused on analyzing sequential programs without interrupts. A recent industrial experience report [10] states the difficulty of estimating the WCET of an interrupt service routine in welding control software, writing: "It was difficult to detect if other interrupts had disturbed the measurement of the current interrupt." While there has been work on testing non-timing "functional" properties of interrupt-driven software (e.g., [11]), there is no systematic work for verifying timing properties of such programs. The work on *schedulability analysis* — in which one analyzes if a task can meet its deadline in spite of pre-emption by other tasks — is related; however, that work treats tasks as atomic objects (see, e.g., [12]), whereas we perform a detailed program analysis of tasks, considering interleaved program paths and interaction of tasks through shared variables. To the best of our knowledge, our technique is the first systematic approach for performing WCET analysis (and other timing analysis) on interrupt-driven programs.

Kidd et al. [13] present an approach to transform a concurrent real-time program with priority pre-emptive scheduling to a sequential program so that any state reachable in the original concurrent program can be reached by performing reachability analysis of the sequential program. This is close to our work in that we could conceivably use their reduction; however, additional assumptions will be needed on inter-arrival time of interrupts, as in our paper. Other methods for more compactly transforming context-bounded concurrent programs to sequential programs are also available [14], [15]; however, with priority pre-emptive scheduling the benefit of these transformations is somewhat limited. Our contribution is to show how the ideas of context-bounding and basis paths can be combined to perform accurate timing analysis of interrupt-driven software.

III. Approach

Consider an interrupt-driven program $P = T_0 \| T_1 \| \ldots \| T_N$, where i denotes the priority level of T_i. We will consider T_0 to be the main function, and all other tasks to be ISRs. Thus, there are n interrupts, which we denote by $\iota_1, \iota_2, \ldots, \iota_n$. As part of the problem description, we are also supplied a lower bound α on the time between interrupts – the "inter-arrival" time of interrupts. Finally, the platform of interest is also specified.

The high-level idea of our approach is to reduce the problem of timing analysis of interrupt-driven programs to timing analysis of sequential programs, by deriving a context bound that is adequate to explore all interleaved paths of P. The approach operates in the following five steps.

1) Use the finite inter-arrival times of interrupts to derive a context bound CB for P that is adequate to explore all interleaved paths of P.
2) Use CB to generate a single sequential program P_{seq} that is path-equivalent to P for the context bound CB.

3) Invoke GameTime [2], a timing analysis technique for sequential programs, on P_{seq}. The key idea in GameTime is to extract a subset of program paths that forms a basis (in the standard linear algebra sense) for the set of all program paths. We term these paths as *basis paths*. GameTime also uses an SMT solver [16] to generate test cases that drive program execution down these basis paths. The key difference with previous applications of GameTime is that the generated basis paths are *interleaved* paths in P, involving context switches between the main function and ISRs.
4) Execute test cases for basis paths on the platform with an interrupt-generation test harness, and measure the execution time of basis paths.
5) Use measured times to infer a platform model, using the GameTime learning algorithm. The inferred model is used to predict execution times of other paths, and answer Problems **P1**, **P2**, and **P3**.

For ease of presentation, we will describe the process somewhat out of order. We will start first with the third item, our technique for timing analysis of sequential programs, then describe the remaining steps.

A. Timing Analysis of Sequential Programs using Basis Paths

While there are several tools for estimating worst-case execution time of sequential programs [1], the only tool we are aware of which can address Problems **P1** and **P3** is GameTime [2], [6]. Our approach therefore builds upon GameTime.

In this section, we give a brief overview of the relevant aspects of GameTime. Most important is the notion of basis paths which helps us deal with the large number of interleaved program paths.

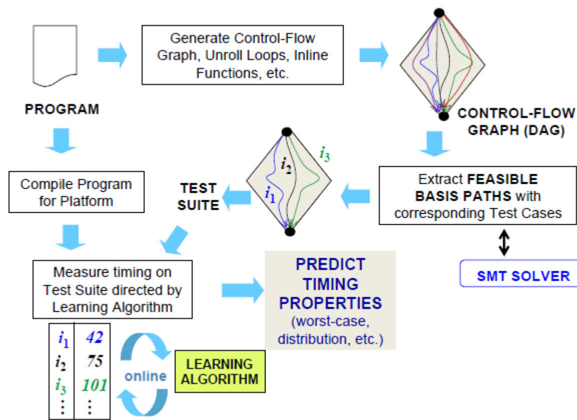


Fig. 2: GameTime overview [17]

Figure 2 depicts the operation of GameTime. The process begins (see top-left corner) with the generation of the control-flow graph (CFG) of the program, in which 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 without loss of generality. 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. Symbolic execution is used to generate an satisfiability modulo theories (SMT) formula for each candidate basis path. An SMT solver is invoked to ensure that the basis paths are feasible; it generates test cases to drive execution down those paths.

The original program (*not* the unrolled, inlined version) is compiled for the target platform, and executed on these test cases. In the basic GameTime algorithm (described in [2], [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; details of theoretical results can be found in the previous papers on GameTime [2], [6]. We provide here a less formal and more intuitive description of the theoretical guarantees for the problems of interest **P1** - **P3**:

P3: Given any δ, GameTime can predict the execution time of any program path to within a tolerance of ϵ with probability $1 - \delta$ by running a number of tests that is polynomial in the program size, in $\ln(\frac{1}{\delta})$, and a parameter $\mu_{\max}$ (described below).
The tolerance ϵ is $O(b\mu_{\max})$, where b is the number of basis paths, and $\mu_{\max}$ is an upper bound on the mean perturbation to program path timing due to path-specific variations to basic block time. Essentially, ϵ depends on how much the time of a basic block can vary based on the path it lies on: the greater the mean variation, the larger the value of ϵ.

P2: For WCET estimation, GameTime provides a similar high-probability guarantee on finding the *path* along which the WCET is exhibited. Once this path is identified, one can simply execute this on the target platform and measure the corresponding execution time. Thus, if GameTime correctly finds the worst-case path, it accurately computes the WCET.
More specifically, given the mean perturbation bound $\mu_{\max}$, if the worst-case path timing is larger than the timing of any other path by a margin ρ (which is also $O(b\mu_{\max})$), then GameTime is guaranteed to find the worst-case path with probability $1 - \delta$ by running a number of tests that is polynomial in the program size, in $\ln(\frac{1}{\delta})$, and $\mu_{\max}$.

It is easy to see how Problem **P1** also receives a similar high-probability theoretical guarantee. However, if the underlying assumption on the margin ρ does not hold, GameTime might

not correctly predict the WCET. In general, it is possible for GameTime to generate an estimate that under- or over-approximates the timing of a program path. In practice, though, we have found the estimates to be accurate (within a few percent relative error) and the worst-case path has been always correctly predicted, even on architectures that include caches, complex pipelines, and branch prediction [6].

We explain the basis path generation process using a simple sequential program that performs modular exponentiation, given in Figure 3(a). Modular exponentiation is a necessary primitive for implementing public-key encryption and decryption. In this operation, a base b is raised to an exponent e modulo a large prime number. In this particular benchmark, we use the *square-and-multiply* method to perform the modular exponentiation, based on the observation that

$$b^e = \begin{cases} (b^2)^{e/2} = (b^{e/2})^2, & e \text{ is even}, \\ (b^2)^{(e-1)/2} \cdot b = (b^{(e-1)/2})^2 \cdot b, & e \text{ is odd}. \end{cases} \quad (1)$$

The unrolled version of the code of Figure 3(a) for a 2-bit exponent is given in Figure 3(b).

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

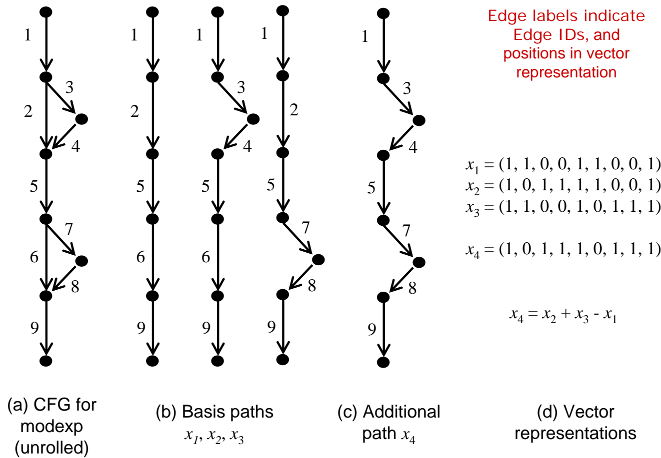


Fig. 4: **CFG and Basis Paths for Code in Fig. 3(b)**

Figure 4(a) denotes the control-flow graph for the code in Figure 3(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. 4(a), each edge is labeled with its index in the vector representation of the path. Edges 2 and 3 respectively 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. 4(b) shows the basis paths for the graph of Fig. 4(a). Here x_1, x_2, and x_3 are the paths corresponding to `exponent` taking values 00, 10, and 01 respectively. Fig. 4(c) shows the fourth path x_4, expressible as the linear combination $x_2+x_3-x_1$ (see Fig. 4(d)).

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.

B. Using Context Bounds to Generate a Sequential Program

Let us assume for this section that we are given a fixed context bound CB. We will explain in Section III-C how a finite inter-arrival time of interrupts can be used to generate a finite context bound.

Given a context bound CB and an interrupt-driven program $P = T_0 \| T_1 \| \ldots \| T_N$, we generate a sequential program P_{seq} that is path-equivalent to P up to context bound CB. Recall that T_j has higher priority than T_i if $j > i$, and that the main function is T_0. The procedure iteratively replaces each T_j, starting with $j = N$, with a replacement sequential program T'_j, such that every interleaved path starting in T_j and possibly involving higher-priority tasks is a program path in T'_j. Thus, T'_0 is the desired sequential program P_{seq}.

The sequential programs T'_j update a set of dummy shared variables that track the number of context switches and the program locations at which context switches occur. We describe below how we obtain T'_j from T_j.

Without loss of generality, suppose that T_j is a sequence of k atomic statements:

$$T_j \triangleq S_1; S_2; S_3; \ldots S_k$$

Thus, for each higher-priority task T_i, $i > j$, there are $k+1$ possible locations where it may be invoked, plus the possibility that it may not interrupt T_j at all. We encode the possible switching points as well as the choice of tasks at

```
modexp(base, exponent) {
 result = 1;
 for(i=EXP_BITS; i>0; i--) {
 // EXP_BITS = 2
  if ((exponent & 1) == 1) {
   result = (result * base) % p;
  }
  exponent >>= 1;
  base = (base * base) % p;
 }
 return result;
}
```

(a) Original code `P`

```
modexp_unrolled(base, exponent) {
 result = 1;
 if ((exponent & 1) == 1) {
  result = (result * base) % p;
 }
 exponent >>= 1;
 base = (base * base) % p;
 // unrolling below
 if ((exponent & 1) == 1) {
  result = (result * base) % p;
 }
 exponent >>= 1;
 base = (base * base) % p;
 return result;
}
```

(b) Unrolled code `Q`

Fig. 3: **Modular exponentation.** Both programs compute the value of `base`$^{\texttt{exponent}}$ modulo `p`.

those switching points using a nondeterministic choice symbol "$*$", which is replaced by a fresh Boolean variable when generating an SMT formula by symbolic path execution. Also, each invocation of a higher-priority task increments a global variable C that tracks the number of context switches. C is initialized to 0 when P begins execution, and a higher-priority task can interrupt a lower-priority task only if $C < CB$.

The sequential program T'_j has the format

$$R_1; S_1; R_2; S_2; R_2; \ldots R_k; S_k; R_{k+1}$$

where each R_l, $l = 1, 2, \ldots, k+1$, is the following piece of code:

$$
\begin{array}{l}
\textbf{for } i = 1..CB \textbf{ do} \\
\quad \textbf{if } C < CB \textbf{ then} \\
\quad\quad \textbf{switch}(*) \ \{ \\
\quad\quad\quad \textbf{case } j+1: \ C := C+1; T'_{j+1}; \textbf{break} \\
\quad\quad\quad \textbf{case } j+2: \ C := C+1; T'_{j+2}; \textbf{break} \\
\quad\quad\quad\quad \ldots \\
\quad\quad\quad \textbf{case } N: \ C := C+1; T'_N; \textbf{break} \\
\quad\quad\quad \textbf{default}: \ \textbf{skip} \\
\quad\quad \}
\end{array}
$$

In the above code snippet, the outer **for** loop encodes the fact that there can be at most CB invocations of a higher-priority task between atomic statements. The nondeterministic choice "$*$" encodes the choice of an arbitrary higher-priority task or no ISR invocation (in the event the "**default**" case is chosen).

It is easy to see that each intermediate statement R_l in T'_N reduces to **skip** and hence T'_N is path-equivalent to T_N. Building on this base case, we can easily obtain the following theorem by induction on N.

Theorem 1: For all $j = 0, 1, 2, \ldots, N$, the set of program paths of T'_j equals the set of all interleaved paths of $T_j \| T_{j+1} \| T_{j+2} \| \ldots \| T_N$ with at most $CB-1$ context switches.

In particular, the set of program paths of $T'_0 = P_{\text{seq}}$ is equal to the set of all interleaved paths of P with at most $CB - 1$ context switches (i.e., a context bound of CB).

C. From Inter-Arrival Times to Context Bounds

Let α be the lower bound on the inter-arrival time of interrupts on the platform of interest. We argue how α can be used to generate a context bound CB that is sufficient to include all executions of the interrupt-driven program P.

We start by hypothesizing that $CB = 1$. With this context bound, we generate a sequential program as described in Sec. III-B and compute the WCET T_W. If T_W is less than α, we know that P will complete execution before a second interrupt is raised. Thus, we can terminate with $CB = 1$.

However, if $T_W \geq \alpha$, it is possible that the main function of P is interrupted twice before terminating. Thus, we set $CB = 2$, regenerate the corresponding sequential program, and recompute the WCET T_W. This time, we compare T_W with 2α. If $T_W < 2\alpha$, we can terminate with $CB = 2$. Otherwise, we increase CB by one and repeat the procedure. In general, when $CB = k$, we compare T_W with $k\alpha$, terminating when $T_W < k\alpha$, and otherwise incrementing CB to $k+1$ and iterating.

If the time taken by an ISR (in the presence of higher-priority interrupts) is less than the minimum inter-arrival time of interrupts, this procedure is guaranteed to terminate with a finite context bound. To see this, note that on each iteration, T_W will grow by a smaller factor than α. This is typically the case for real-time embedded software: the rule of thumb is that ISRs must terminate very quickly in order to guarantee that every interrupt is serviced. The execution time of ISRs are typically a small fraction of the minimum inter-arrival time α.

D. Generating Measurements for Basis Paths and Predictions

The sequential program P_{seq} is fed as input to GameTime, which generates basis paths for this program along with the corresponding test cases. Each test case includes an assignment to program variables as well as to the nondeterministic choice variables that indicate where tasks are interrupted and by which higher-priority tasks.

We then execute the test cases within a harness that triggers interrupts at the right locations as indicated by the test case.

This harness is specific to each platform, involving the use of a few inline assembly instructions at each interrupt point (location). While this involves a slight modification to the original code, given the small number of inline assembly instructions, we believe any skew to program timing is miniscule.

Measurements can be obtained using one of a range of execution time measurement techniques – again these are platform-specific. Perhaps the most non-intrusive (but rather expensive) method is the use of a logic analyzer. Somewhat simpler is the use of on-chip cycle counters or on-board timers. These are applicable provided the code fragment is small enough that the timer register does not overflow. We use the latter approach as it is applicable for our benchmarks. Any alternative accurate measurement technique can be used instead. It is important to note that getting accurate measurements on the embedded platform can be a time-consuming process, involving repeated re-compilation and logging of measurements — therefore, it is desirable to limit the number of measurements taken. (We will see in the next section how the notion of basis paths helps us to limit the number of measurements taken, while retaining prediction accuracy.) Further platform-specific details about measurement are given in Section IV-B.

Once the measurements are obtained for the basis paths, we invoke GAMETIME's learning algorithm (as described in Section III-A) to provide answers to the problem of interest (**P1**, **P2**, or **P3**).

E. Efficiency Analysis

In this section, we calculate the number of basis paths that GAMETIME requires to perform its timing analysis and compare it to the total number of paths that are possible through a sequential program, to demonstrate the efficiency of the GAMETIME approach.

We assume that the control-flow graph of a task T_i $(1 \leq i \leq j)$ has m_i edges, n_i nodes, and p_i possible interrupt points, with a context bound of CB. Let $m = \max_i m_i$ and $p = \max_i p_i$. Since, in the worst case, the number of interrupt points can exceed the number of basic blocks, $m_i = O(p_i)$ and $m = O(p)$. For ease of analysis, we first consider a specific task T_i that can only be interrupted by exactly one higher priority task $T_j, j > i$. To generate the sequential task T_i' corresponding to task T_i, we make copies of the control-flow graph of T_j and attach a copy to each interrupt point in task T_i. The control-flow graph of T_i' thus has $O(m_i + p_i \cdot CB \cdot m_j)$ edges, which is $O(p^2 \cdot CB)$. As described earlier and in [2], the number of basis paths is linear in the number of edges, and so GAMETIME will infer $O(p^2 \cdot CB)$ basis paths. In contrast, since there are p_i possible interrupt points, and each interrupt point can be taken at least once and at most CB times, we have at least one unique program path through T_i for every choice of CB out of $p_i \cdot CB$ interrupt points. Thus, there are $O((pCB)^{CB})$ total paths through the control-flow graph of T_i. This simple case of two tasks is representative of the difference between the total number of paths through the control-flow graph of a sequentialized task and the number of basis paths that GAMETIME requires.

We can generalize this to the case of multiple tasks: consider a specific task T_i that can be interrupted by any higher priority task $T_r, (i < r \leq j)$. We can generate the sequential task T_i' corresponding to task T_i as follows: We do not need to sequentialize T_j since it is the highest priority thread and thus cannot be pre-empted by any other thread. Thus, the sequential task T_j' is the same as T_j. We sequentialize the task T_{j-1} as described in the case of two tasks to create a control-flow graph with $O(m_{j-1} + p_{j-1} \cdot CB \cdot m_j)$ edges. We can then sequentialize the task T_{j-2} by noticing that either T_{j-1}' or T_j' can interrupt at each interrupt point of T_{j-2}. The task T_{j-2}' thus has $O(p_{j-2} \cdot CB \cdot (m_{j-1} + p_{j-1} \cdot CB \cdot m_j)) = O(p_{j-2} \cdot CB \cdot m_{j-1} + p_{j-2} \cdot p_{j-1} \cdot CB^2 \cdot m_j)$ edges. Proceeding inductively, we see that the size of the control-flow graph of the sequential task T_i' is $O(\sum_{r=i}^{j-1}(\prod_{\ell=i}^{r} p_\ell CB) m_{r+1}))$ edges. However, not all the paths through this CFG are feasible, due to the context bound. In fact, to determine the number of basis paths, notice that with a context bound of CB, the effective product $\prod_{\ell=i}^{r} p_\ell CB$, after eliminating paths with more than CB context switches, has at most CB terms. Thus, the number of basis paths grows as $O((pCB)^{CB} m)$. Note that this is polynomial in the size of the tasks and is independent of the number of tasks. Using more compact transformations to a sequential program (e.g., [14], [15]) it might be possible to further reduce this bound.

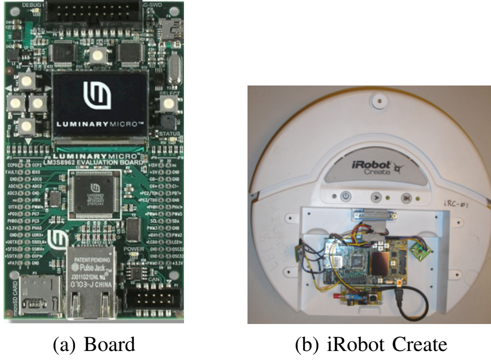
To determine the *total* number of paths through the sequential task T_i', we recognize that any of the $p_i \cdot CB$ interrupt points can be the location of one of the (at most) CB context switches. An interleaving through k tasks is a combination of CB out of $(p \cdot CB)^k$ choices of combinations of interrupt points. Thus, the total number of paths grows as $O((p \cdot CB)^{(k \cdot CB)})$. Note that this grows exponentially in the number of tasks.

IV. EXPERIMENTAL RESULTS

The goal of the experiments reported here is to demonstrate that our approach can, by measuring only a small linear subset of interleaved paths, accurately predict (i) the worst-case execution time for interrupt-driven programs (which we check by exhaustively enumerating all program paths), and (ii) the execution time along any arbitrary program path.

A. Physical Apparatus and Benchmarks

We used the Luminary Micro LM3S8962 board [7], interfaced to the iRobot Create autonomous robot platform [8] for our experiments. This microcontroller is shown in Figure 5(a) and the iRobot Create in Figure 5(b). The Luminary Micro board contains a 32-bit ARM Cortex M3 microcontroller, running at 50 MHhz. This microcontroller is interfaced to a range of peripherals: of special interest for our experiments is the UART interface to built-in iRobot sensors and the analog-to-digital interface to an ADXL-322 accelerometer. The built-in



(a) Board (b) iRobot Create

Fig. 5: Luminary Micro LM3S8962 and iRobot Create

sensors include buttons that a human can press, cliff sensors, and a bump sensor. Since the robot moves slowly, and humans cannot press buttons very quickly twice in a row, the minimum inter-arrival time of interrupts α was estimated at about 1ms for our experiments.

Our benchmarks included a toy example based on the modular exponentiation program introduced earlier, plus several real iRobot control programs. A summary of the benchmarks used is presented in Table I. The benchmarks are described in more detail below, and are also available online at `http://uclid.eecs.berkeley.edu/gametime/fmcad11/`.

Name	LOC	Size of CFG		Total Num. of paths	b	Context bound
		n	m			
modexp	60	60	70	500	12	1
iRobot-1	210	55	60	33	5	1
iRobot-2	230	141	160	3362	17	1
iRobot-3	230	97	108	1281	10	2
iRobot-4	280	213	244	33728	30	1
iRobot-5	250	179	206	65088	27	1

TABLE I: **Characteristics of Benchmarks.** "LOC" indicates number of lines of C code for the task. The Control-Flow Graph (CFG) size refers to that of the sequential program P_{seq} fed as input to GAMETIME: n is the number of nodes, m is the number of edges. The column b refers to the number of basis paths in the graph, as deduced by GAMETIME. The total number of paths indicates the total number of interleaved execution paths, not accounting for path feasibility.

B. Generating Interrupts and Measurements

To measure the timing of each basis path, we need to force an interrupt at one or more program points, depending on the context bound. There are two types of interrupts that can be forced: software and hardware interrupts. For this platform, the overhead of invoking an ISR through hardware interrupts is similar to that using software interrupts; therefore, for convenience, we decided to force software interrupts.

Software interrupts can be modeled by embedding the ARM assembly instruction `SVC` into the C code under analysis. This instruction is a supervisor call that forces a software interrupt. To use this instruction, we modify the interrupt vector table to include a custom interrupt handler. In the code under analysis, we then insert an `SVC` assembly instruction whose argument is the position of the interrupt handler in the vector table, so that on execution, the instruction goes directly to the vector table and invokes the interrupt handler for the interrupt we wish to trigger.

Obtaining Timing Measurements: The execution time of the program was measured using an on-board timer called `SysTickTimer`. This timer can be set to periodically generate an interrupt by counting down from a large starting value. The period of the timer is user-specified and is large enough that it will not finish until long after the program under analysis finishes. To get the execution time for the program under analysis, we start the timer off before the program runs and read off its value when the program finishes. We assume that the program would finish within 16,777,261 cycles (the highest possible value for the `SysTickTimer` period), which is a realistic assumption for our set of benchmarks.

C. Modular Exponentiation

Our first example is a version of the modular exponentiation example introduced in Sec. III-A. An arbitrary prime number was used for our benchmarks. For our experiments, we used a base of two, with four-bit exponents. Two of the four conditionals were moved into a mock 'interrupt handler'; the program comprising the remaining two conditions formed the "main" task. Thus, the program comprises two tasks: each with two of the conditionals. Each code fragment of the form below is considered an atomic statement.

```
if ((exponent & 1) == 1)
  result = (result * base) % p;
exponent >>= 1;

base = (base * base) % p;
```

We used GAMETIME to determine the values that would sensitize the basis paths in the control flow graph of the "main" task, and it provided 12 test cases. With a context bound of 1, there are three program points where the ISR can be invoked. Since there are 16 values of `exponent`, and three possible interrupt points, the total number of test cases is $16 * 3 = 48$.

With the measurements for the 12 test cases corresponding to the basis paths, GAMETIME was employed to infer a timing model using which it predicted the runtimes of each of the 48 different test cases.

In figure 6, we plot the predicted values and the measured values for each of the 48 test cases. The predicted values match the measured values with an error of less than 5%. In particular, our approach accurately predicts the worst-case execution time and produces the corresponding test case. We observe that the WCET estimate, about 290 cycles or 5.8μs, is much less than the 1ms inter-arrival time of interrupts, implying that the context bound of 1 is sufficient.

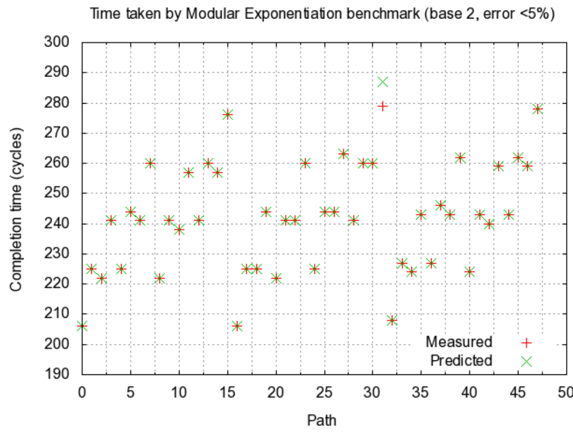


Fig. 6: **Time taken by the Modular Exponentiation benchmark**

D. iRobot Driving Code

The iRobot Create control programs we consider here involve a sample operation where the robot attempts to keep moving forward until it senses an obstacle, in which case it will try to back up, turn, and move around the obstacle. The robot can be stopped by pushing a button on its console. The accelerometer detects changes in the speed of the robot, such as when it accelerates on level ground or when it climbs a hill.

All iRobot sensors (with the exception of the accelerometer) trigger the same UART interrupt that is serviced with a single ISR. This ISR reads the values of the sensors or the status of the buttons from a UART queue, and updates local variables accordingly. The accelerometer triggers a different interrupt that is serviced by a different routine that also updates local variables with the accelerometer readings.

The code that produces the iRobot behavior described above is an infinite loop encoding a state machine. The body of this loop involves the next-state update operation based on sensor data, and this is what we used the five iRobot benchmarks shown in Table I. The "main task" in all benchmarks has a similar structure: it updates the state of the robot based on sensor readings, button presses, or accelerometer readings, if any, and the new state, if changed, modifies the velocity of the iRobot. All benchmarks also have at least two interrupt points: each conditional in the state update is considered atomic, and the velocity modification is also considered atomic. The first four benchmarks used only the sensor interrupt handler; the last used only the accelerometer interrupt handler.

The first iRobot benchmark, iRobot-1, is a highly simplified version of the behavior described above. A context bound of 1 was sufficient. The simplified state machine allows us to manually enumerate and time all of the feasible paths, and also to use the basis paths to predict the time for all paths. The measured and predicted timings for the ten feasible paths are shown in figure 7: the timings agree within one percent, and the path that was predicted to take the longest time is also the path that was measured to take the longest time. The WCET is less than 2500 cycles, which is less than 50μs, much smaller than the 1ms inter-arrival time, ensuring that the context bound of 1 is sufficient.

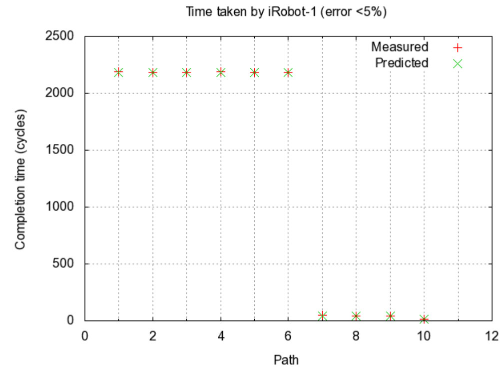


Fig. 7: **Time taken by iRobot-1 benchmark**

The second benchmark adds one more state to the state machine. For this and the remaining benchmarks, the number of interleaved program paths (as seen from Table I) is over 1000 — thus, it is not possible to time all the possible paths. Therefore, for these benchmarks, we arbitrarily selected 16 paths to be measured. The true (measured) execution times of these paths are compared with the runtimes predicted from measuring just the basis paths and running GameTime. The resulting plot for the iRobot-2 is shown in Figure 8. Again, a context bound of 1 suffices.

To experiment with a larger context bound, we assumed the minimum inter-arrival time to be $\alpha = 50\mu$s, and analyzed the third benchmark iRobot-3. With a context bound of 1, the WCET exceeded this value. However, with a context bound of 2, the WCET is 4357 cycles, or about 87μs, which is less than $2\alpha = 100\mu$s. As shown in Figure 9, the error margin between predicted and true (measured) values is almost zero.

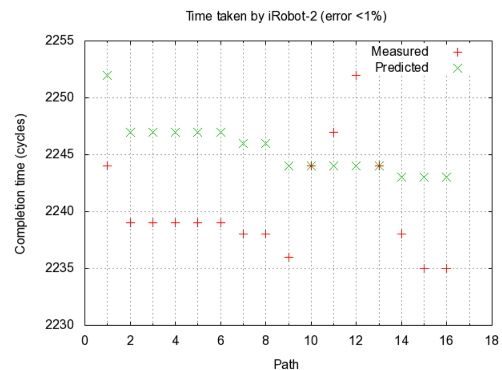


Fig. 8: **Time taken by iRobot-2 benchmark**

The fourth benchmark iRobot-4 adds more states to the state machine, while the fifth benchmark iRobot-5 keeps only those states that use the accelerometer. Nonetheless, the error margin

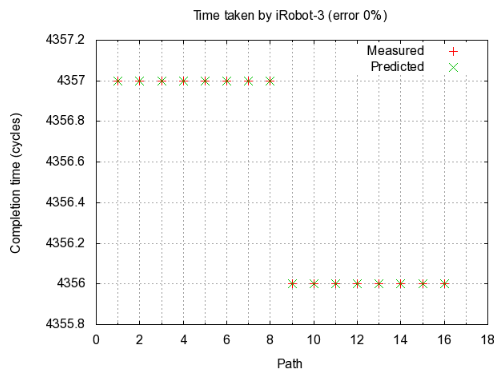


Fig. 9: **Time taken by iRobot-3 benchmark.** Note that there are only two execution times exhibited by these paths: 4357 and 4356 cycles.

in both benchmarks, for the 16 chosen paths, is less than 2 percent. In both cases, a context bound of 1 sufficed.

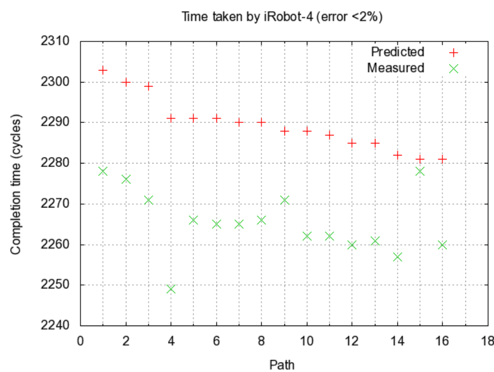


Fig. 10: **Time taken by iRobot-4 benchmark**

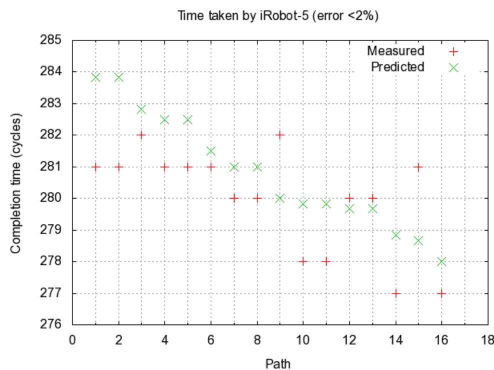


Fig. 11: **Time taken by iRobot-5 benchmark**

V. Conclusion and Future Work

In this paper, we present a new approach for timing analysis of interrupt-driven programs. The key ideas in our approach are to bound the exploration of the path space using the twin notions of context bounds and basis paths. We have demonstrated for a real embedded platform and control software that our approach can accurately predict not only the worst-case execution time, but also the execution time of arbitrary interleaved program paths without needing to exhaustively enumerate and test them. For future work, we plan to expand our experimental evaluation to include larger benchmarks with several interrupt service routines (tasks).

Acknowledgments. This work was supported in part by NSF grants CNS-0644436 CNS-0627734, and CNS-1035672, an Alfred P. Sloan Research Fellowship, the Toyota Motor Corporation, and the Hellman Family Faculty Fund. We thank the anonymous referees for their comments.

References

[1] Reinhard Wilhelm 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.

[2] S. A. Seshia and A. Rakhlin, "Game-theoretic timing analysis," in *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2008, pp. 575–582.

[3] NASA Engineering and Safety Center, "NASA report on Toyota unintended acceleration investigation, appendix a: Software," http://www.nhtsa.gov/staticfiles/nvs/pdf/NASA_FR_Appendix_A_Software.pdf.

[4] S. Qadeer and D. Wu, "KISS: keep it simple and seqeuential," in *In PLDI 04: Programming Language Design and Implementation*, 2004, pp. 14–24.

[5] S. Qadeer and J. Rehof, "Context-bounded model checking of concurrent systems," in *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2005.

[6] S. A. Seshia and A. Rakhlin, "Quantitative analysis of systems using game-theoretic learning," *ACM Transactions on Embedded Computing Systems (TECS)*, 2011, to appear.

[7] Luminary Micro, Inc., "Luminary Micro microcontroller datasheet," http://chess.eecs.berkeley.edu/eecs149/sp09/docs/Datasheet_LM3S8962.pdf.

[8] iRobot Corporation, "iRobot Create User's Manual," http://www.irobot.com/filelibrary/pdfs/hrd/create/Create\%20Manual_Final.pdf.

[9] D. Brylow and J. Palsberg, "Deadline analysis of interrupt-driven software," *IEEE Transactions on Software Engineering*, 2004.

[10] J. Gustafsson and A. Ermedahl, "Experiences from applying WCET analysis in industrial settings," in *Proc. IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, 2007, pp. 382–392.

[11] J. Regehr, "Random testing of interrupt-driven software," in *Proc. 5th ACM International Conference on Embedded Software (EMSOFT)*, 2005, pp. 290–298.

[12] M. G. Harbour, M. H. Klein, and J. P. Lehoczky, "Timing analysis for fixed-priority scheduling of hard real-time systems," *IEEE Trans. Software Engineering*, vol. 20, no. 1, pp. 13–28, 1994.

[13] N. Kidd, S. Jagannathan, and J. Vitek, "One stack to run them all - reducing concurrent analysis to sequential analysis under priority scheduling," in *17th International SPIN Workshop on Model Checking Software (SPIN)*, 2010, pp. 245–261.

[14] A. Lal and T. W. Reps, "Reducing concurrent analysis under a context bound to sequential analysis," in *CAV'08*, ser. LNCS, vol. 5123, 2008, pp. 37–51.

[15] S. K. Lahiri, S. Qadeer, and Z. Rakamarić, "Static and precise detection of concurrency errors in systems code using SMT solvers," in *CAV'09*, ser. LNCS, vol. 5643, 2009, pp. 509–524.

[16] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability*, A. Biere, H. van Maaren, and T. Walsh, Eds. IOS Press, 2009, vol. 4, ch. 8.

[17] S. A. Seshia and J. Kotker, "GameTime: A toolkit for timing analysis of software," in *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2011.