Quantitative Analysis of Embedded Software Using Game-Theoretic Learning

Sanjit A. Seshia
Alexander Rakhlin

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-130
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-130.html

September 22, 2009

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE **22 SEP 2009**	2. REPORT TYPE	3. DATES COVERED **00-00-2009 to 00-00-2009**

4. TITLE AND SUBTITLE
Quantitative Analysis of Embedded Software Using Game-Theoretic Learning

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
University of California at Berkeley,Department of Electrical Engineering and Computer Sciences,Berkeley,CA,94720

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR’S ACRONYM(S)

11. SPONSOR/MONITOR’S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The analysis of quantitative properties, such as timing and power, is central to the design of reliable real-time embedded software and systems. However, the verification of such properties on a program is made difficult by their heavy dependence on the program?s environment, such as the processor it runs on. Modeling the environment by hand can be tedious, error-prone, and time consuming. In this paper, we present a new, game-theoretic approach to analyzing quantitative properties that is based on performing systematic measurements to automatically learn a model of the environment. We model the estimation problem as a game between our algorithm (player) and the environment of the program (adversary) where the player seeks to accurately predict program properties while the adversary sets environment parameters to thwart the player. We present both theoretical and experimental evidence for the utility of our game-theoretic approach. On the theoretical side, we show that we can predict the program property for all execution paths with probability greater than 1&#8722;d by only making a number of measurements that is polynomial in ln(1/d) and the program size. Experimental results for execution time analysis demonstrate that our approach is efficient, effective, and highly portable.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**28**	

Quantitative Analysis of Embedded Software Using Game-Theoretic Learning

Sanjit A. Seshia
University of California, Berkeley
sseshia@eecs.berkeley.edu

Alexander Rakhlin
University of Pennsylvania
rakhlin@wharton.upenn.edu

Abstract

The analysis of quantitative properties, such as timing and power, is central to the design of reliable real-time embedded software and systems. However, the verification of such properties on a program is made difficult by their heavy dependence on the program's environment, such as the processor it runs on. Modeling the environment by hand can be tedious, error-prone, and time consuming. In this paper, we present a new, game-theoretic approach to analyzing quantitative properties that is based on performing systematic measurements to automatically learn a model of the environment. We model the estimation problem as a game between our algorithm (player) and the environment of the program (adversary), where the player seeks to accurately predict program properties while the adversary sets environment parameters to thwart the player. We present both theoretical and experimental evidence for the utility of our game-theoretic approach. On the theoretical side, we show that we can predict the program property for *all* execution paths with probability greater than $1-\delta$ by only making a number of measurements that is polynomial in $\ln(1/\delta)$ and the program size. Experimental results for execution time analysis demonstrate that our approach is efficient, effective, and highly portable.

1 Introduction

The main distinguishing characteristic of embedded computer systems is their tight integration with the physical world. Consequently, the behavior of software controllers of such *cyber-physical* systems has a major effect on physical properties of such systems. These properties are quantitative, including constraints on resources, such as timing and power, and specifications involving physical parameters, such as position and velocity. The verification of such physical properties of embedded software systems requires modeling not only the software program but also the relevant aspects of the program's environment. However, only limited progress has been made on these verification problems. One of the biggest obstacles is to create an adequately accurate model of a complex environment.

Consider, for example, the problem of estimating the execution time of a software task. This problem plays a central role in the design of real-time embedded systems, to provide timing guarantees and for use in scheduling algorithms. In spite of significant research on this topic over the last 20 years (e.g. [14, 20]), this problem remains far from solved. The complexity arises from the two dimensions of the problem: the *path problem*, which is to find the worst-case path through the task, and the *state problem*, which seeks to find the worst-case environment state to run the task from. The problem is particularly challenging because these two dimensions interact closely: the choice of path affects the state and vice-versa. Significant progress has been made on this problem, especially in the computation of bounds on loops in tasks, in modeling the dependencies amongst program fragments using (linear) constraints, and modeling some aspects of processor behavior. However, as pointed out in recent papers by Lee [12] and Kirner and Puschner [11], it is becoming increasingly difficult to precisely model the complexities of the underlying hardware platform (e.g., out-of-order processors with deep pipelines, branch prediction, caches, parallelism) as well as the software environment. This results in timing estimates that are either too pessimistic (due to conservative platform modeling) or too optimistic (due to unmodeled features of the platform). Industry practice typically involves making random, unguided measurements to obtain timing estimates. As Kirner and Puschner [11] write, a major challenge for measurement-based techniques is the automatic and systematic generation of test data.

In this paper, we present a new *game-theoretic* approach to verifying physical properties of embedded software by running *systematic tests* of the software in its target environment, and *learning an environment model*. The following salient features of our approach distinguish it from previous approaches in the literature:

- *Game-theoretic formulation:* We model the problem of estimating a physical quantity (such as time) as a multi-round game between our estimation algorithm (player) and the environment of the program (adversary). The physical quantity is modeled as the length of the particular execution path the program takes. In the game, the player seeks to estimate the length of any path through the program while the adversary sets environment parameters to thwart the player. Each round of the game constitutes one test. Over many rounds, our algorithm learns enough about the environment to be able to accurately predict path lengths with high probability. In particular, we show how our algorithm can be used to predict the longest path and thus predict properties such as worst-case execution time (WCET).

- *Learning an environment model:* A key component of our approach is the use of statistical learning to generate an environment model that is used to estimate the physical quantity of interest. The environment is viewed as an adversary that selects weights on edges of the program's control flow graph in a manner that can depend on the choice of the path being tested. This path-dependency is modeled as a perturbation of weights that can be introduced by the adversary. Our algorithm seeks to estimate path lengths in spite of such adversarial setting of weights. The algorithm is robust not only to adversarial choices made by the environment, but also to errors in measurement.

- *Systematic and efficient testing:* Another central idea is to perform *systematic measurements* of the physical quantity, by sampling only so-called *basis paths* of the program. The intuition is that the length of any program path can be approximated as a linear combination of the observed lengths of the basis paths. We use satisfiability modulo theories (SMT) solvers and integer programing to generate feasible basis paths and to generate test inputs to drive a program's execution down a basis path.

Although our focus in this paper is on software analysis, we believe that the above concepts are also useful for the analysis of physical properties of embedded systems in general.

We present both theoretical and experimental results demonstrating the utility of our approach. On the theoretical side, we prove that if we run a number of tests that is polynomial in the input size and $\ln \frac{1}{\delta}$, our algorithm can accurately estimate the length of any path in the program with probability $1 - \delta$ (formal statement in Section 4). In particular, we can use this result to estimate the length of the longest path – for timing, this amounts to estimating the worst-case execution time (WCET). More generally, we show that our algorithm can estimate the length of all program paths (i.e. the "timing profile" of the program) and, for any ε, it can also be used to find paths of length within ε of the longest.

We demonstrate our approach for the problem of execution time analysis of embedded software. Our approach is implemented in a tool called GAMETIME. We present experimental results comparing GAMETIME to existing state-of-the-art WCET estimation tools that are based on combining static analysis and integer programming. Results indicate that our approach can generate *even bigger* execution-time estimates than these techniques, without incurring the difficulties involved in modeling complex processor behavior. Since our approach is measurement-based, it is easy to apply to varied and complex platforms. Moreover, as noted above, our approach can be used not just for worst-case analysis, but also to predict ε-longest paths and for predicting execution times of arbitrary program paths.

For concreteness, we focus the rest of the paper on execution time analysis. However, the theoretical formulation and results in Section 4 are potentially applicable for estimating *any physical quantity* of embedded software; we have therefore sought to present our theoretical results in a general manner as relating to the lengths of paths in a graph.

The outline of the paper is as follows. We begin with a survey of related work in Section 2, mainly focussed on execution time analysis. The basic formulation and an overview of our approach is given in Section 3. The algorithm and main theorems are given in Section 4, and experimental results in Section 5. We conclude in Section 6.

A preliminary version of this work appeared in [22]. This technical report expands on both theoretical and experimental results, describing the theoretical model in far greater detail.

2 Background and Related Work

We briefly review literature on estimating physical parameters of software and relevant results from learning theory.

2.1 Estimating Execution Time and Other Physical Quantities

There is a vast literature on estimation execution time, especially WCET analysis, comprehensively surveyed by Li and Malik [14] and Wilhelm et al. [27, 20]. For lack of space, we only include here a brief discussion of current approaches and do not cover all tools. References to current techniques can be found in a recent survey [20].

There are two parts to current WCET estimation methods: *program path analysis* (also called *control flow analysis*) and *processor behavior analysis*. In program path analysis, the tool tries to find the program path that exhibits worst-case execution time. In processor behavior analysis (PBA), one models the details of the platform that the program will execute on, so as to be able to predict environment behavior such as cache misses and branch mis-predictions. PBA is an extremely time-consuming process, with several man-months required to create a reliable timing model of even a simple processor design.

Current tools are broadly classified into those based on *static analysis* (e.g., aiT, Bounds-T, SWEET, Chronos) and those that are *measurement-based* (e.g., RapiTime, SymTA/P, Vienna M./P.). Static tools rely on abstract interpretation and dataflow analysis to compute facts at program points that identify dependencies between code fragments and generate loop bounds. Even static techniques use measurement for estimating the time for small program fragments, and measurement-based techniques rely on techniques such as model checking to guide path exploration. Static techniques also perform implicit path enumeration (termed "IPET"), usually based on integer linear programming. The state-of-the-art measurement-based techniques [26] are based on generating test data by a combination of program partitioning, random and heuristic test generation, and exhaustive path enumeration by model checking.

Our technique is *measurement-based*; hence, it suffers no over-estimation and is easy to port to a new platform. It is distinct from existing measurement-based techniques due to the novel game-theoretic formulation, basis path-based test generation, and the use of online learning to infer an environment model. Our approach does rely on some static techniques, in deriving loop bounds and using symbolic execution and satisfiability solvers to compute inputs to drive the program down a specific path of interest. In particular, note that our approach completely avoids the difficulties of processor behavior analysis, instead directly executing the program on its target platform. Moreover our approach applies not just to WCET estimation, but also to estimating the entire execution time profile of a program.

While there have been several papers about quantitative verification of formal models of systems (e.g. [5]), these typically assume that the quantitative parameters of primitive elements (such as execution time of software tasks) are given as input. There is relatively little work on directly verifying non-timing properties on software, with the exception of estimating the power used by software-controlled embedded systems [24].

Adversarial analysis has been employed for problems such as system-level dynamic power management [10], but to our knowledge, the adversarial model and analysis used in this paper is the first for timing estimation and for estimating quantitative parameters of software.

2.2 Learning Theory

Results of this paper build on the *game-theoretic prediction* literature in learning theory. This field has witnessed an increasing interest in sequential (or *online*) learning, whereby an agent discovers the world by repeatedly acting and receiving feedback. Of particular interest is the problem of learning in the presence of an adversary with a *complete absence of statistical assumptions* on the nature of the observed data.

The problem of sequentially choosing paths to minimize the *regret* (the difference between cumulative lengths of the paths chosen by our algorithm and the total length of the longest path after T rounds) is known as an instance of *bandit online linear optimization*. The "bandit" part of the name is due to the connection with the *multi-armed bandit* problem, where only the payoff of the chosen "arm" (path) is revealed. The basic "bandit" problem was put forth by Robbins [21] in 1952 and has been well-understood since then. The recent progress comes from the realization that well-performing algorithms can be found (a) for large decision spaces, such as paths in a graph, and (b) under adversarial conditions rather than the stochastic formulation of Robbins. We are the first to bring these results to bear on the problem of quantitative analysis of embedded software.

We refer the reader to a recent book [4] for a comprehensive treatment of sequential prediction. Some relevant results can be found in [17, 9, 1].

2.3 Miscellaneous

Our algorithm uses the concept of *basis paths* of a program, which has been explored before in the software engineering community to compute the *cyclomatic complexity* of a program [16]; however, our theoretical results rely on extracting a special basis called a *barycentric spanner* [1]. Our approach heavily relies on advances in SMT solving for input test generation; these techniques are surveyed in a recent book chapter [2].

3 Theoretical Formulation and Overview

We are concerned with estimating a physical property of a software task (program) executing in its target platform (environment). The physical quantity of interest is in general a function of three things: the program code, parameters of its environment, and the inputs to the program. More concisely, we can express the physical quantity q as the following function

$$q = f_P(x, w)$$

where x denotes the inputs to the program (such as data read from memory or received over the network), w denotes the environment parameters (such as the contents of the cache or network delays), and f_P denotes the program-specific function that maps x and w to a value of the physical quantity.

In general, x and w vary over time, and so does q. However, the function f_P is typically constant over time. We will make the variation with time explicit by adding a subscript:

$$q_t = f_P(x_t, w_t)$$

Some sample physical properties of interest are as follows:

- *Global worst-case estimation:* In this case, we want to estimate the largest value of the quantity q for all values of x and w:
$$\max_{x,w} \; f_P(x,w) \tag{1}$$
- *Worst-case estimation over a time horizon* τ*:* This is a similar problem as above, except that the worst case is to be computed over a finite time horizon τ, formally specified as follows:
$$\max_{t=1..\tau} \max_{x_t,w_t} \; f_P(x_t,w_t) \tag{2}$$
- *Average-case estimation over a time horizon against a worst-case environment:* In this case, we want to estimate, for a time horizon τ and for any sequence of environment parameters $w_1, w_2, \ldots, w_\tau$, the following quantity:
$$\max_{x_t} \frac{1}{\tau} \sum_{t=1}^{\tau} f_P(x_t,w_t) \tag{3}$$
- *Can the system consume R resources at any point over a time horizon of* τ*:* The question we ask here is whether q_t exceeds R for any choice of t, x_t, and w_t. For example, a concrete instance of this problem is to ask whether a software task can take more than R seconds to execute.

For concreteness, in the remainder of this section, we will focus on a single quantity, *execution time*, and on a single representative problem, namely, the *worst-case execution time* (WCET) estimation problem. However, our theoretical formulation and algorithms carry over to estimating any physical quantity and to problems other than worst-case analysis.

The WCET estimation problem can be defined as follows:

> Given a terminating software task S and a platform M on which S executes, estimate the longest time S takes to terminate on M.

Moreover, we will focus on WCET estimation over a finite time horizon τ. If we let τ go to ∞, this problem reduces to the true WCET estimation problem. For brevity, we will simply refer to finite-horizon WCET estimation as WCET estimation; however, our experimental results compare against techniques for the true WCET estimation problem.

The main ideas in our theoretical formulation are elaborated below.

Game-theoretic formulation: We model the WCET estimation problem as a game between the WCET estimation tool $\mathcal{T}$ and the environment $\mathcal{E}$ of S.

The game proceeds over multiple rounds, $t = 1,2,3,\ldots$. In each round, $\mathcal{T}$ picks the inputs x to S. These inputs determine the path taken through the program. $\mathcal{E}$ picks, in a potentially adversarial fashion, environment parameters w. This choice by $\mathcal{E}$ *can depend* on the inputs selected by $\mathcal{T}$.

At the end of each round t, $\mathcal{T}$ receives as feedback the execution time l_t of S for its chosen path under the parameters chosen by $\mathcal{E}$. Note that we assume that $\mathcal{T}$ only receives the overall execution time of the task, not a more fine-grained measurement of (say) each basic block in the task along the chosen path. This enables us to minimize any skew from instrumentation inserted to measure time. Based on the feedback l_t, $\mathcal{T}$ can modify its input-selection strategy.

After some number of rounds τ, we stop: $\mathcal{T}$ must output its prediction of the longest execution time of S that could have been exhibited during rounds $t = 1, 2, \ldots, \tau$. $\mathcal{T}$ wins the game if its prediction is correct; otherwise, $\mathcal{E}$ wins. *The goal of $\mathcal{T}$ is thus to select a sequence of inputs so that it can accumulate enough data to identify, with high probability, the longest execution time of S during $t = 1, 2, \ldots, \tau$.*

Note that this longest execution time need not be due to inputs that have been already tried out by $\mathcal{T}$.

By permitting $\mathcal{E}$ to select environment parameters based on $\mathcal{T}$'s choice of path, we can model path-dependent timing as well as perturbation in execution time of a single path due to variation in environmental conditions or measurement error. The more predictable the timing behavior of the platform, the smaller this perturbation will be. For theoretical analysis, we model the perturbation as a random variable whose mean is bounded by a parameter $\mu_{\max}$. If a platform has predictable timing, such as the PRET processor proposed by Edwards and Lee [6], it would mean that $\mu_{\max}$ is small. (The $\mu_{\max}$ parameter will play a role in determining the rate of convergence of our proposed algorithm.)

Formulation as a graph problem: An additional aspect of our model is that the game operates on the control-flow graph G_S of the task S, with loops unrolled to a pre-determined safe upper bound.

In this setting, the game described above works as follows. At any round t, the player $\mathcal{T}$ selects a path x_t through the graph G_S from a designated *source node* (entry point of the function) to a designated *sink node* (exit point/return statement of the function). This is performed by generating input values for S to drive execution down path x_t, using standard constraint-based test generation techniques using SMT solvers. $\mathcal{E}$ selects lengths for all source-sink paths in G_S, where this selection can depend on the choice of x_t. However, $\mathcal{E}$ only reveals the length l_t of the chosen path x_t.

The goal of $\mathcal{T}$ is thus to *select paths* so that within a time horizon τ it can accumulate enough data to identify, with high probability, the longest path in G_S during rounds $t = 1, 2, \ldots, \tau$.

Next, we formalize the above problem definition.

3.1 Theoretical Formulation

Consider a directed acyclic graph $G = (V, E)$ derived from the control-flow graph of the task with all loops unrolled. We will assume that there is a single source node u and single sink node v in G; if not, then dummy source and sink nodes can be added.

Let $\mathcal{P}$ denote the set of all paths in G from source u to sink v. We can associate each of the paths with a binary vector with $m = |E|$ components, depending on whether the edge is present or not. In other words, each source-sink path is a vector x in $\{0,1\}^m$, where the ith entry of the vector for a path x corresponds to edge i of G, and is 1 if edge i is in x and 0 otherwise. The set $\mathcal{P}$ is thus a subset of $\{0,1\}^m$.

The path prediction interaction is modeled as a repeated game between our algorithm ($\mathcal{T}$) and the program environment ($\mathcal{E}$). On each round t, $\mathcal{T}$ chooses a path $x_t \in \mathcal{P}$ between u and v. Concurrent with this choice, the adversary $\mathcal{E}$ picks a table of non-negative path lengths given by the function $\mathcal{L}_t : \mathcal{P} \to \mathbb{R}^{\geq 0}$. Then, the total length l_t of the chosen path x_t is revealed, where $l_t = \mathcal{L}_t(x_t)$. The game proceeds for some number of rounds $t = 1, 2, \ldots, \tau$.

At the end of round τ, the goal of $\mathcal{T}$ is to accurately estimate the worst-case execution time due to environment states in rounds $t = 1, 2, \ldots, \tau$. This can be expressed as the following quantity:

$$\mathcal{L}_{\max} = \max_{x \in \mathcal{P}} \max_{t=1,2,\ldots,\tau} \mathcal{L}_t(x) \tag{4}$$

Moreover, we would also like $\mathcal{T}$ to identify the worst-case path given by

$$x^* = \text{argmax}_{x \in \mathcal{P}} \max_{t=1,2,\ldots,\tau} \mathcal{L}_t(x) \tag{5}$$

We make a few remarks on the above theoretical model.

First, we stress that, in the above formulation, the goal is to find the WCET *due to environment states in rounds* $t = 1, 2, \ldots, \tau$. In order to find the true WCET, for all possible environment states, we need to assume that the worst-case state occurs at some time between $t = 1$ and $t = \tau$. We contend that this formulation is useful in spite of this assumption because it serves to decouple the path dimension of the WCET estimation problem from the state dimension. In our experience, for many applications, the worst-case environment state does appear at some time during testing — the problem is that testing may not pick the worst-case path at that same time. With our formulation, the goal is to accurately estimate the WCET even if we do not sample the worst-case path when the worst-case state occurred.

Second, the definition of our estimation target $\mathcal{L}_{\max}$ assumes that the timing of a program depends only on the control flow through that program. In general, the timing can also depend on characteristics of input data that do not influence control flow. We believe that the basic framework we describe here also applies to the case of data-dependent timing, and leave an exploration of this aspect to future work.

Overall, we believe that decoupling the path problem from the state problem in a manner that can be applied easily to any platform is in itself a significant challenge. This paper mainly focuses on solving this problem. In future work, we plan to address the limitations of the model identified above.

The third and final remark we make is about the "size" of the theoretical model. Since a DAG can have exponentially-many paths in the number of nodes and edges, the domain of the function $\mathcal{L}_t$ is potentially exponential, and can change at each round t. In the worst case, the strategy sets of both $\mathcal{T}$ and $\mathcal{E}$ in this model are exponential-sized, and it is impossible to exactly learn $\mathcal{L}_t$ for every t without sampling all paths. Hence, we need to approximate the above model with another model that, while being more compact, retains enough accuracy to generate useful results in practice.

Below, we present a more compact model, which our algorithm is then based upon. We will present this model in two steps.

3.1.1 Modeling with Weights and Perturbation

We model the selection of the table of lengths $\mathcal{L}_t$ by the environment $\mathcal{E}$ as a two-step procedure.

(i) First, $\mathcal{E}$ chooses a vector of non-negative *edge weights*, $w_t \in \mathbb{R}^m$, for G. These weights represent *path-independent* delays of basic blocks in the program.

(ii) Then, after observing the path x_t selected by $\mathcal{T}$, $\mathcal{E}$ picks a distribution from which it draws a perturbation vector $\pi_t(x_t)$. The functional notation indicates that the distribution is a function of x_t.

The vector $\pi_t(x_t)$ models the path-specific changes that $\mathcal{E}$ applies to its original choice w_t. We will abbreviate $\pi_t(x_t)$ by $\pi^x{}_t$. In cases where we wish to denote $\pi_t(x')$ for x' that could be different from x_t, we will explicitly write $\pi^x{}_t(x')$ or $\pi_t(x')$.

The only restriction we place on $\pi_t(x)$, for any x, is that $\|\pi_t(x)\|_1 \leq N$, for some finite N. The parameter N is arbitrary, but places the constraint that the perturbation of any path length cannot be unbounded.

Thus, the overall path length observed by $\mathcal{T}$ is

$$l_t = x_t \cdot (w_t + {\pi^x}_t) = {x_t}^{\mathsf{T}}(w_t + {\pi^x}_t)$$

Now let us consider how this model relates to the original formulation we started with.

First, note that, in the original model, $\mathcal{E}$ picks the function $\mathcal{L}_t$ that defines the lengths of all paths. To relate to that model, here we can assume, without loss of generality, that $\mathcal{E}$ draws a-priori the perturbation vectors ${\pi^x}_t(x)$ for all $x \in \mathcal{P}$, but only ${\pi^x}_t(x_t)$ plays a role in determining l_t.

Second, equating the observed lengths, we see that

$$\mathcal{L}_t(x_t) = x_t^{\mathsf{T}}(w_t + {\pi^x}_t)$$

The main constraint on this equation is the requirement that $\|{\pi^x}_t\|_1 \leq N$, which implies that $|x_t^{\mathsf{T}}{\pi^x}_t| \leq N$. In effect, by using this model we require that $\mathcal{E}$ pick $\mathcal{L}_t$ by first selecting path-independent weights w_t and then, for each source-sink path, modifying its length by a perturbation of at most $\pm N$. Note, however, that the model places absolutely no restrictions on the value of w_t or how it changes with t (from round to round).

The goal for $\mathcal{T}$ in this model is to estimate the following quantity

$$\mathcal{L}_{\text{max}} = \max_{x \in \mathcal{P}} \max_{t=1,2,\ldots,\tau} x^{\mathsf{T}}(w_t + {\pi^x}_t) \tag{6}$$

Moreover, we would also like $\mathcal{T}$ to identify the worst-case path given by

$$x^* = \text{argmax}_{x \in \mathcal{P}} \max_{t=1,2,\ldots,\tau} x^{\mathsf{T}}(w_t + {\pi^x}_t) \tag{7}$$

3.1.2 Simplified Model without Perturbation

To more easily introduce the key concepts in our algorithm, we will initially assume that the perturbation vectors at all time points are identically 0, viz., ${\pi^x}_t(x) = 0$ for all t and x.

Clearly, this is an unrealistic idealization in practise, since in this model the length of an edge is independent of the path it lies on. We stress that our main theoretical results are for the more realistic model defined in Section 3.1.1.

We next give an overview of our approach in the context of a small example.

3.2 Overview of Our Approach

We describe the working of our approach using a small program from an actual real-time embedded system, the Paparazzi unmanned aerial vehicle (UAV) [18]. Figure 1 shows the C source code for the `altitude_control_task` in the Paparazzi code, which is publicly available open source.

Starting with the source code for a task, and all the libraries and other definitions it relies on, we run the task through a C pre-processor and the CIL front-end [8] and extract the control-flow graph (CFG). In this graph, each node corresponds to the start of a basic block and edges are labeled with the basic block code or conditional statements that govern control flow. Figure 2 shows the CFG for the code shown in Figure 1. Note that we assume that code terminates, and bounds are known on all loops. Thus, we start with code with all loops (if any) unrolled, and the CFG is thus a directed acyclic graph (DAG). We also pre-process the CFG so that it has exactly one source and one sink. Each execution through the program is a source-to-sink path in the CFG.

An exhaustive approach to program path analysis will need to enumerate all paths in this DAG. However, it is well-known that a DAG can have exponentially many paths (in the number of vertices/edges). Thus, a brute-force enumeration of paths is not efficient.

```
#define PPRZ_MODE_AUTO2 2
#define PPRZ_MODE_HOME 3
#define VERTICAL_MODE_AUTO_ALT 3
#define CLIMB_MAX 1.0
...
void altitude_control_task(void) {
 if (pprz_mode == PPRZ_MODE_AUTO2
     || pprz_mode == PPRZ_MODE_HOME) {
  if (vertical_mode == VERTICAL_MODE_AUTO_ALT) {
  /* inlined below: function  altitude_pid_run(); */
    float err = estimator_z - desired_altitude;
    desired_climb = pre_climb + altitude_pgain * err;
    if (desired_climb < -CLIMB_MAX)
      desired_climb = -CLIMB_MAX;
    if (desired_climb > CLIMB_MAX)
     desired_climb = CLIMB_MAX;
}}}
```

Figure 1: **Source code for** `altitude_control_task`

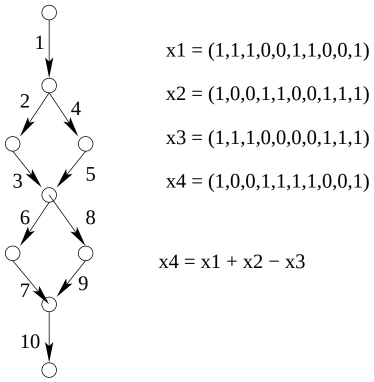
1
2
4
3
5
6
8
7
9
10
x1 = (1,1,1,0,0,1,1,0,0,1)
x2 = (1,0,0,1,1,0,0,1,1,1)
x3 = (1,1,1,0,0,0,0,1,1,1)
x4 = (1,0,0,1,1,1,1,0,0,1)
x4 = x1 + x2 − x3

Figure 3: **Illustration of Basis Paths.** An edge label indicates the position for that edge in the vector representation of a path.

Our approach is to sample a set of *basis paths*. Recall that each source-sink path can be viewed as a vector in $\{0,1\}^m$, where m is the number of edges in the unrolled CFG. The set of all valid source-sink paths thus forms a subset $\mathcal{P}$ of $\{0,1\}^m$. We compute the basis for $\mathcal{P}$ in which each element of the basis is also a source-sink path.

Figure 3 illustrates the ideas using a simple "2-diamond" example of a CFG. In this example, paths x_1, x_2 and x_3 form a basis and x_4 can be expressed as the linear combination $x_1 + x_2 - x_3$.

Our algorithm, described in detail in Section 4, randomly samples basis paths of the CFG and drives program execution down those paths by generating tests using constraint solving. From the observed lengths of those paths, we estimate edge weights on the entire graph. This estimate, accumulated over several rounds of the game, is then used to predict the longest source-sink path in the CFG. Theoretical guarantees on performance are proved in Section 4 and experimental evidence for its utility is given in Section 5.

4 Algorithm and Theoretical Results

Recall that, in the model introduced in the previous section, the path prediction interaction is modeled as a repeated game between our algorithm (Player) and the program environment (Adversary). On each round t, we choose a source-sink path $x_t \in \mathcal{P} \subseteq \{0,1\}^m$. The adversary chooses the lengths of paths in the graph. We assume that this choice is made by the following two stage process: first, the adversary chooses the worst-

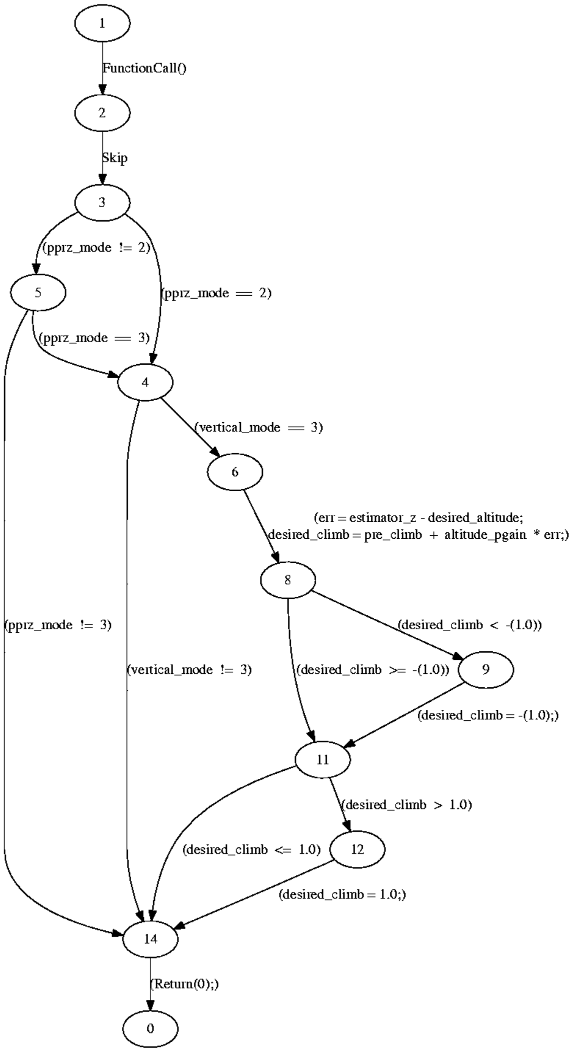

Figure 2: **Control-flow graph for** `altitude_control_task`

case *weights*, $w_t \in \mathbb{R}^m$, on the edges of G independently of our choice x_t, and then skews these weights by adding a random perturbation $\pi^x{}_t$, whose distribution depends on x_t. (We will also refer to edge weights and path lengths as "delays", to make concrete the link to timing analysis.)

In the simplified model, which we consider first, we suppose that the perturbation is zero; thus, we observe the overall path length $l_t = x_t^\mathsf{T} w_t$. In the general model, only $l_t = x_t^\mathsf{T}(w_t + \pi^x{}_t)$ is observed. No other information is provided to us; not only do we not know the lengths of the paths not chosen, we do not even know the contributions of particular edges on the chosen path. It is important to emphasize that in the general model we assume that the adversary is adaptive in that w_t and $\pi^x{}_t$ can depend on the past history of choices by the player and the adversary.

Suppose that there is a single fixed path x^* which is the longest one on each round. One possible objective is to find x^*. In the following, we exhibit an efficient randomized algorithm which allows us to find it correctly with high probability. In fact, our results are more general: if no single longest path exists, we can provably find a batch of longest paths. We describe later how our theoretical approach paves the way for analyzing worst-case execution time given a range of assumptions at hand.

Before diving into the details of the algorithm, let us sketch how it works:

- First, compute a representative set of basis paths, called a *barycentric spanner* (see section 4.1)
- For a specified number of iterations τ, do the following:
  ⋆ pick a path from the representative set
  ⋆ observe its length
  ⋆ construct an estimate of edge weights on the whole graph from the observed length
- Find the longest path or a set of longest paths based on the average of the estimates over τ iterations.

It might seem mysterious that we can re-construct edge weights (delays, for the case of timing analysis) on the whole graph based a single number, which is the total length of the path we chose. To achieve this, our method exploits the power of randomization and a careful choice of a representative set of paths. The latter choice is discussed next.

4.1 Focusing on a Barycentric Spanner

It is well-known in the game-theoretic study of path prediction that any deterministic strategy against an adaptive adversary will fail [4]. Therefore, the algorithm we present below is randomized. As we only observe the entire length of the path we choose, we are bound to select from the set of paths *covering* the whole graph or else we risk missing a highly time-consuming edge. However, simply covering the graph is not enough – note that such coverage corresponds to "statement coverage" in the program, without covering all ways of getting to a statement. Indeed, a key feature of the algorithm is the ability to exploit correlations between paths to guarantee that we find the longest. Hence, we need a *barycentric spanner* (introduced by [1]), a set of up to m paths with two valuable properties: any path in the graph can be written as a linear combination of the paths in the spanner, and the coefficients in this linear combination are bounded in absolute value. The first requirement says that the spanner is a good representation for the exponentially-large set of possible paths; the second says that lengths of some of the paths in the spanner will be of the same order of magnitude as the length of the longest path. These properties enable us to repeatedly sample from the barycentric spanner and reconstruct delays on the whole graph. We then employ concentration inequalities[1] to prove that these reconstructions, on average, converge to the true delays of the paths. Once we have a good statistical estimate of the true weights on all the edges, it only remains to run a longest-path algorithm for weighted directed acyclic graphs (LONGEST-PATH), subject to path feasibility constraints.

The existence of a barycentric spanner has been shown in Awerbuch and Kleinberg [1]. In particular, the authors provide the following procedure to find a 2-barycentric spanner set (where coefficients are bounded in absolute value by 2) $\{b_1,\ldots,b_m\} \in \mathcal{P}$ (see also [17]).

In Algorithm 1, $B = (b_1,\ldots,b_m)$ and $B_{-i} = (b_1,\ldots,b_{i-1},b_{i+1},\ldots,b_m)$. The output of the algorithm is the final value of B, a 2-barycentric spanner; i.e., any path $x \in \mathcal{P}$ can be written as $x = \sum_{i=1}^{m} \alpha_i b_i$ with $|\alpha_i| \leq 2$. The ith iteration of the for-loop in lines 2-4 repeatedly replaces the ith element of the standard basis with a path that is orthogonal to the previous $i-1$ paths identified so far and with all remaining standard basis vectors and also spans the path-space $\mathcal{P}$. Line 3 of the algorithm corresponds to maximizing a linear function over the set $\mathcal{P}$, and can be solved using LONGEST-PATH.[2] At the end of the for-loop, we are left with a basis of $\mathcal{P}$ that is not necessarily a 2-barycentric spanner. Lines 5-7 of the algorithm refine this basis into a 2-barycentric spanner using the same LONGEST-PATH optimization oracle that is used in the for-loop.

[1] Concentration inequalities are sharp probabilistic guarantees on the deviation of a function of random variables from its mean.

[2] In practise, to compute feasible basis paths one must add constraints that rule out infeasible paths, as is standard in integer programming formulations for timing analysis [14]; in this case, the longest-path computation is solved as an integer linear program.

Algorithm 1 Finding a 2-Barycentric Spanner

```
$(b_1, \ldots, b_m) \leftarrow (\mathbf{e}_1, \ldots, \mathbf{e}_m)$.
for $i = 1$ to $m$ do {{Compute a basis of $\mathcal{P}$}}
    $b_i \leftarrow \arg\max_{x \in \mathcal{P}} |\det(x, B_{-i})|$
end for
while $\exists x \in \mathcal{P},\ i \in \{1, \ldots, m\}$ satisfying
$|\det(x, B_{-i})| > 2|\det(b_i, B_{-i})|$ do {{Transform $B$ into a 2-barycentric spanner}}
    $b_i \leftarrow x$
end while
```

Algorithm 2 GAMETIME with simplified environment model

```
Input $\tau \in \mathbb{N}$
Compute a 2-barycentric spanner $\{b_1, \ldots, b_b\}$
for $t = 1$ to $\tau$ do
    Environment chooses $w_t$.
    We choose $i_t \in \{1, \ldots, b\}$ uniformly at random.
    We predict the path $x_t = b_{i_t}$ and observe the path length $\ell_t = b_{i_t}^{\mathsf{T}} w_t$
    Estimate $\tilde{v}_t \in \mathbb{R}^b$ as $\tilde{v}_t = b\ell_t \cdot \mathbf{e}_{i_t}$, where $\{\mathbf{e}_i\}$ denotes the standard basis.
    Compute estimated weights $\tilde{w}_t = B^+ \tilde{v}_t$
end for
Use the obtained sequence $\tilde{w}_1 \ldots \tilde{w}_\tau$ to find a longest path(s). For example, for Theorem 4.2, we compute
$x_\tau^* := \arg\max_{x \in \mathcal{P}} x^{\mathsf{T}} \sum_{t=1}^{\tau} \tilde{w}_t$.
```

One can intuitively view the determinant computation as computing the volume of the corresponding polytope. Maximizing the determinant amounts to spreading the vertices of the polytope as far as possible in order to obtain a "diverse" set of basis paths.

It is shown [1] that the running time of Algorithm 1 is only quadratic in m. Gyorgy et al. [9] extend the above procedure to the case where the set of paths spans only a b-dimensional subspace of $\mathbb{R}^m$ (where $b \leq m$), a scenario which is more realistic for our setting. Slightly abusing notation, let B be the $b \times m$ matrix with b_i's as rows. We define the Moore-Penrose pseudo-inverse of B as $B^+ = B^{\mathsf{T}}(BB^{\mathsf{T}})^{-1}$. It holds that $BB^+ = I_b$. For theoretical analysis, let M be any upper bound on the length of any basis path.

Since we have assumed an adaptive adversary that produces w_t based on our previous choices $x_1 \ldots x_{t-1}$ as well as the random factors ${\pi^x}_1 \ldots {\pi^x}_{t-1}$, we should take care in dealing with expectations. To this end, let us denote the conditional expectation $\mathbb{E}_t[A] = \mathbb{E}[A | i_1, \ldots, i_{t-1}, {\pi^x}_1, \ldots, {\pi^x}_{t-1}]$, keeping in mind that randomness at time t in the general model stems from our random choice i_t of the basis path *and* the adversary's random choice ${\pi^x}_t$ given i_t. In the simplified model, all randomness is due to our choice of the basis path, and this makes the analysis more transparent. We stress that the adversary can vary the distribution of ${\pi^x}_t$ according to the path chosen by the Player.

4.2 Analysis under the Simplified Model

We now analyze the effectiveness of GAMETIME under the simplified model presented in Section 3. We begin by proving some key properties of the algorithm.

Preliminaries

The following Lemma is key to proving that Algorithm 2 performs well. It quantifies the deviations of our estimates of the delays on the whole graph, $\tilde{w}_t$, from the true delays w_t, which we cannot observe.

Lemma 4.1 *With probability at least* $1-\delta$, *for all* $x \in \mathcal{P}$,

$$\left| \frac{1}{\tau} \sum_{t=1}^{\tau} (\tilde{w}_t - w_t)^{\mathsf{T}} x \right| \le \tau^{-1/2} c \sqrt{2b + 2\ln(2\delta^{-1})}, \tag{8}$$

where $c = 4bM$.

Proof: We will show that $\mathbb{E}_t \tilde{w}_t x = w_t x$ for any $x \in \mathcal{P}$, i.e. the estimates are unbiased[3] *on the subspace spanned by* $\{b_1, \ldots, b_b\}$. By working directly in the subspace, we obtain the required probabilistic statement and will have the dimensionality of the subspace b, not m, entering the bounds.

Define $v_t = Bw_t$ just as $\tilde{v}_t = B\tilde{w}_t$. Taking expectations with respect to i_t, conditioned on $i_1, \ldots, i_{t-1}$,

$$\mathbb{E}_t \tilde{v}_t = \mathbb{E}_t \left[b(b_{i_t}^{\mathsf{T}} w_t) \cdot \mathbf{e}_{i_t} \right] = \frac{1}{b} \sum_{i=1}^{b} b(b_i^{\mathsf{T}} w_t) \cdot \mathbf{e}_i = Bw_t = v_t.$$

Fix any $\alpha \in \{-2, 2\}^b$. We claim that the sequence $Z_1, \ldots, Z_\tau$, where $Z_t = \alpha^{\mathsf{T}}(\tilde{v}_t - v_t)$ is a bounded martingale difference sequence. Indeed, $\mathbb{E}_t Z_t = 0$ by the previous argument. A bound on the range of the random variables Z_t can be computed by observing

$$|\alpha^{\mathsf{T}} \tilde{v}_t| = |\alpha^{\mathsf{T}} [b(b_{i_t}^{\mathsf{T}} w_t) \mathbf{e}_{i_t}]| \le 2b |b_{i_t}^{\mathsf{T}} w_t| \le 2bM$$

and

$$|\alpha^{\mathsf{T}} v_t| \le 2bM,$$

implying

$$|Z_t| \le 4bM \doteq c.$$

An application of Azuma-Hoeffding inequality (see Appendix) for a martingale difference sequence yields, for the fixed α,

$$\Pr\left(\left| \sum_{t=1}^{\tau} Z_t \right| > c\sqrt{2\tau \ln(2(2^b)\delta^{-1})} \right) \le \delta / 2^b.$$

Having proved a statement for a fixed α, we would like to apply the union bound[4] to arrive at the corresponding statement for any $\alpha \in [-2, 2]^b$. This is implausible as the set is uncountable. However, applying a union bound over the *vertices* of the hypercube $\{-2, 2\}^b$ is enough. Indeed, if $|\sum_{t=1}^{\tau} Z_t| = |\alpha^{\mathsf{T}} \sum_{t=1}^{\tau} (\tilde{v}_t - v_t)| \le \xi$ for all vertices of $\{-2, 2\}^b$, then immediately $|\sum_{t=1}^{\tau} Z_t| \le \xi$ for any $\alpha \in [-2, 2]^b$ by linearity. Thus, by union bound,

$$\Pr\left(\forall \alpha \in [-2, 2]^b, \ \left| \sum_{t=1}^{\tau} \alpha^{\mathsf{T}} (\tilde{v}_t - v_t) \right| \le c\sqrt{2\tau b + 2\tau \ln(2\delta^{-1})} \right) \ge 1 - \delta.$$

Any path x can be written as $x^{\mathsf{T}} = \alpha^{\mathsf{T}} B$ for some $\alpha \in [-2, 2]^b$. Furthermore, $\tilde{w}_t = B^{+} \tilde{v}_t$ implies that $x^{\mathsf{T}} \tilde{w}_t = \alpha^{\mathsf{T}} B B^{+} \tilde{v}_t = \alpha^{\mathsf{T}} \tilde{v}_t$ and $x^{\mathsf{T}} w_t = \alpha^{\mathsf{T}} v_t$. We conclude that

[3] For random variables X and $\tilde{X}$, $\tilde{X}$ is said to be an unbiased estimate of X if $\mathbb{E}[X - \tilde{X}] = 0$.

[4] Also known as Boole's inequality, the union bound says that the probability that at least one of the countable set of events happens is at most the sum of the probabilities of the events, e.g. $\Pr(A \cup B) \le \Pr(A) + \Pr(B)$.

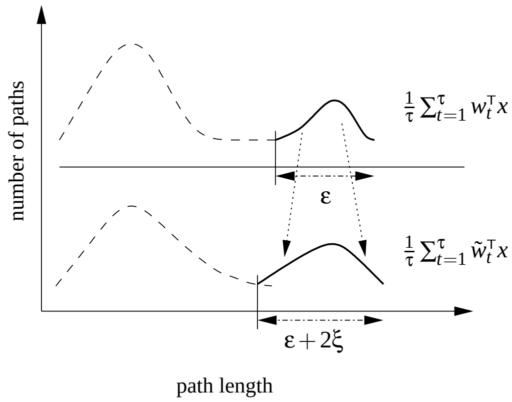
number of paths
$\frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\top x$
ε
$\frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^\top x$
ε + 2ξ
path length

Figure 4: **Illustration of the second inclusion in Lemma 4.2.** The set of ε-longest paths, the object of interest, is contained in the set of $(\varepsilon+2\xi)$-longest paths w.r.t. to the sequence $\tilde{w}_1,\ldots,\tilde{w}_\tau$. Under a margin assumption, equality between the two sets can be shown, as exhibited by Theorem 4.2.

$$\Pr\left(\forall x \in \mathcal{P},\ \left|\sum_{t=1}^{\tau}(\tilde{w}_t - w_t)^\top x\right| \le c\sqrt{2\tau b + 2\tau\ln(2\delta^{-1})}\right) \ge 1-\delta$$

and the statement follows by dividing by τ.

□

Estimating the Set of Longest Paths

With the help of Lemma 4.1, we can now analyze how the longest (or almost-longest) paths with respect to the estimated $\tilde{w}_t$'s, where path lengths are averaged over all rounds, compare to the true averaged longest paths.

Definition 4.1 *Define the set of* ε*-longest paths with respect to the actual delays*

$$\mathcal{S}_\tau^\varepsilon = \left\{ x \in \mathcal{P} : \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\top x \ge \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\top x' - \varepsilon \right\}$$

and with respect to the the estimated delays

$$\tilde{\mathcal{S}}_\tau^\varepsilon = \left\{ x \in \mathcal{P} : \frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^\top x \ge \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^\top x' - \varepsilon \right\}.$$

In particular, $\mathcal{S}_\tau^0$ *is the set of longest paths.*

The following Lemma makes our intuition precise: with enough trials τ, the set of longest paths, which we can calculate after running Algorithm 2, becomes almost identical to the true set of longest paths. We illustrate this point graphically in Figure 4: In a histogram of average path lengths, the set of longest paths (the right "bump") is somewhat smoothed when considering the path lengths under the estimated $\tilde{w}_t$'s. In other words, paths might have a slightly different average path length under the estimated and actual weights. However, we can still guarantee that this smoothing becomes negligible for large enough τ, enabling us to locate the longest paths.

Lemma 4.2 *For any* $\varepsilon > 0$ *and for* $\xi = \tau^{-1/2}4bM\sqrt{2b+2\ln(2\delta^{-1})}$,

$$\tilde{\mathcal{S}}_\tau^\varepsilon \subseteq \mathcal{S}_\tau^{\varepsilon+2\xi} \quad \textit{and} \quad \mathcal{S}_\tau^\varepsilon \subseteq \tilde{\mathcal{S}}_\tau^{\varepsilon+2\xi}$$

with probability at least $1-\delta$*.*

Proof: Let $x \in \tilde{\mathcal{S}}_\tau^\varepsilon$ and $y \in \mathcal{S}_\tau^0$. Suppose that we are in the $(1-\delta)$-probability event of Lemma 4.1. Then

$$\begin{aligned} \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal x \geq \frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^\intercal x - \xi &\geq \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^\intercal x' - \varepsilon - \xi \\ &\geq \frac{1}{\tau}\sum_{t=1}^{\tau} \tilde{w}_t^\intercal y - \varepsilon - \xi \geq \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal y - \varepsilon - 2\xi \\ &= \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal x' - \varepsilon - 2\xi, \end{aligned}$$

where the first and fourth inequalities follow by Lemma 4.1, the third inequality is by definition of maximum, and the second and fifth are by definitions of $\tilde{\mathcal{S}}_\tau^\varepsilon$ and $\mathcal{S}_\tau^0$, resp. Since the sequence of inequalities holds for any $x \in \tilde{\mathcal{S}}_\tau^\varepsilon$, we conclude that $\tilde{\mathcal{S}}_\tau^\varepsilon \subseteq \mathcal{S}_\tau^{\varepsilon+2\xi}$. The other direction of inclusion is proved analogously. $\square$

Note that $\xi \to 0$ as $\tau \to \infty$. To compute the set $\mathcal{S}_\tau^0$, we can instead compute the set $\tilde{\mathcal{S}}_\tau^{2\xi}$ that contains it. If $|\tilde{\mathcal{S}}_\tau^{2\xi}| \leq k$, for some parameter k, then we can use an algorithm that computes the k longest paths (see, e.g., [7]) to find this set.

Results under Unique Longest Path Assumption

While Lemma 4.2 is very general, we now give one interesting implication for finding a longest path under the following assumption.

Assumption 4.1 *There exists a single path* x^* *that is the longest path on any round with a certain (known) margin* ρ*:*

$$\forall x \in P,\ x \neq x^*,\ \forall t,\ (x^* - x)^\intercal w_t > \rho$$

Note that if there is a unique longest path (for any margin $\rho \geq 0$), then we can see that

$$x^* \ = arg \max_{x \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal x \ = arg \max_{x \in \mathcal{P}} \max_{t=1..\tau} w_t^\intercal x$$

Thus, under the above margin assumption, we can, in fact, recover the longest path, as shown in the next Theorem.

Theorem 4.2 *Suppose Assumption 4.1 holds with* $\rho > 0$*. We run the Algorithm 2 for* $\tau = (8bM)^2\rho^{-2}(2b + 2\ln(2\delta^{-1}))$ *iterations. Then with probability at least* $1-\delta$*, Algorithm 2 outputs*

$$x_\tau^* := \arg\max_{x \in \mathcal{P}} x^\intercal \sum_{t=1}^{\tau} \tilde{w}_t$$

and x_τ^* *is equal to* x^**.*

Proof:

Let $x^*_\tau = \arg\max_{x\in\mathcal{P}} x\sum_{t=1}^{\tau}\tilde{w}_t$. We claim that, with probability $1-\delta$ it is equal to x^*. Indeed, suppose $x^*_\tau \neq x^*$. By Lemma 4.2, $x^*_\tau \in \tilde{\mathcal{S}}^0_\tau \subseteq \mathcal{S}^{2\xi}_\tau$ with probability at least $1-\delta$. Thus,

$$\frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal x^*_\tau \geq \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal x^* - 2\xi.$$

Assumption 4.1, however, implies that

$$\frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal x^*_\tau < \frac{1}{\tau}\sum_{t=1}^{\tau} w_t^\intercal x^* - \rho$$

leading to a contradiction whenever $\rho \geq 2\xi = \tau^{-1/2}8bM\sqrt{2b+2\ln(2\delta^{-1})}$). Rearranging the terms, we arrive at $\tau \geq (8bM)^2\rho^{-2}(2b+2\ln(2\delta^{-1}))$, as assumed. We conclude that with probability at least $1-\delta$, $x^*_\tau = x^*$ and $\{x^*\} = \tilde{\mathcal{S}}^0_\tau = \mathcal{S}^{2\xi}_\tau$.

□

The following weaker assumption also has interesting implications.

Assumption 4.2 *There exists a path $x^* \in \mathcal{P}$ such that it is the longest path on any round*

$$\forall x \in P,\ \forall t,\ (x^* - x)^\intercal w_t \geq 0$$

If, after running Algorithm 2 for enough iterations, we find all 2ξ-longest paths (the set $\tilde{\mathcal{S}}^{2\xi}_\tau$), Lemma 4.2 guarantees that, under Assumption 4.2, the longest path $x^* \in \mathcal{S}^0_\tau$ is one of them with high probability. As discussed earlier, we can use an efficient k-longest paths computation to find a set containing $\mathcal{S}^0_\tau$. We can then use this information to repeatedly test the candidate paths in this set to find the worst-case path and estimate its length.

4.3 Analysis under General Weights-Perturbation Model

We now present an analysis of GAMETIME under the general weight-perturbation model given in Sec. 3. For easy reference, we give the GAMETIME algorithm again below but with the new environment model Algorithm 3.

As before, let M be any upper bound on the length of any basis path (where the length includes the perturbation).

In the general model, the environment $\mathcal{E}$ picks a distribution with mean ${\mu^x}_t \in \mathbb{R}^m$, which depends on the algorithm's chosen path x. From this distribution, $\mathcal{E}$ draws a vector of perturbations ${\pi^x}_t \in \mathbb{R}^m$. The vector ${\pi^x}_t$ satisfies the following assumptions:

- *Bounded perturbation:*
  $||{\pi^x}_t||_1 \leq N$, where N is a parameter.
- *Bounded mean perturbation of path length:*
  For any path $x \in \mathcal{P}$, $|x^\intercal {\mu^x}_t| \leq \mu_{\max}$

Note that ${\mu^x}_t$ is a function of the chosen path, and that ${\pi^x}_t$ depends on ${\mu^x}_t$.

We now state the main lemma for our general model. In this case, we calculate $\tilde{w}_t$ as an estimate of the sum $w_t + {\pi^x}_t$.

Algorithm 3 GameTime with general environment model

```
Input $\tau \in \mathbb{N}$
Compute a 2-barycentric spanner $\{b_1, \ldots, b_b\}$
for $t = 1$ to $\tau$ do
    Environment chooses $w_t$.
    We choose $i_t \in \{1, \ldots, b\}$ uniformly at random.
    Environment chooses a distribution from which to draw ${\pi^x}_t$, where the mean ${\mu^x}_t$ and support of the distribution satisfies the assumptions given above.
    We predict the path $x_t = b_{i_t}$ and observe the path length $\ell_t = b_{i_t}^{\mathsf{T}}(w_t + {\pi^x}_t)$
    Estimate $\tilde{v}_t \in \mathbb{R}^b$ as $\tilde{v}_t = b\ell_t \cdot \mathbf{e}_{i_t}$, where $\{\mathbf{e}_i\}$ denotes the standard basis.
    Compute estimated weights $\tilde{w}_t = B^+ \tilde{v}_t$
end for
Use the obtained sequence $\tilde{w}_1 \ldots \tilde{w}_\tau$ to find a longest path(s). For example, for Theorem 4.4, we compute $x_\tau^* := \arg\max_{x \in \mathcal{P}} x^{\mathsf{T}} \sum_{t=1}^{\tau} \tilde{w}_t$.
```

Lemma 4.3 *With probability at least* $1-\delta$*, for all* $x \in \mathcal{P}$*,*

$$\left| \frac{1}{\tau} \sum_{t=1}^{\tau} (\tilde{w}_t - w_t - {\pi^x}_t)^{\mathsf{T}} x \right| \leq (2b+1)\mu_{\max}$$
$$+\tau^{-1/2} \left[c\sqrt{2b + 2\ln(4\delta^{-1})} + d\sqrt{2m + 2\ln(4\delta^{-1})} \right] \tag{9}$$

where $c = 2b(2M + \mu_{\max})$ *and* $d = N + \mu_{\max}$.

Proof: The proof is similar to that of Lemma 4.1, so we only highlight the differences here.

$$\begin{aligned}
\mathbb{E}_t \tilde{v}_t &= \mathbb{E}_{i_t} \left\{ \mathbb{E}_{{\pi^x}_t} \left[b(b_{i_t}^{\mathsf{T}} w_t + b_{i_t}^{\mathsf{T}} {\pi^x}_t(x_t)) \cdot \mathbf{e}_{i_t} | i_t \right] \right\} \\
&= \frac{1}{b} \sum_{i=1}^{b} b(b_i^{\mathsf{T}} w_t) \cdot \mathbf{e}_i + \frac{1}{b} \sum_{i=1}^{b} b(b_i^{\mathsf{T}} \mu_t^{b_i}) \mathbf{e}_i \\
&= B w_t + \mu_t^{\text{basis}}
\end{aligned}$$

where μ_t^{basis} denotes the $b \times 1$ vector of means in which the ith element is $b_i^{\mathsf{T}} \mu_t^{b_i}$ and each entry is bounded in absolute value by $\mu_{\max}$.

Fix any $\alpha \in \{-2, 2\}^b$. As before, the sequence $Z_1, \ldots, Z_\tau$, where $Z_t = \alpha^{\mathsf{T}}(\tilde{v}_t - v_t - \mu_t^{\text{basis}})$ is a bounded martingale difference sequence. A bound on the range of the random variables can be computed by observing

$$|\alpha^{\mathsf{T}} \tilde{v}_t| = |\alpha^{\mathsf{T}}[b(b_{i_t}^{\mathsf{T}}(w_t + {\pi^x}_t))\mathbf{e}_{i_t}]| \leq 2b|b_{i_t}^{\mathsf{T}}(w_t + {\pi^x}_t)| \leq 2bM$$

and

$$|\alpha^{\mathsf{T}} \mu_t^{\text{basis}}| \leq 2b\mu_{\max}, \quad |\alpha^{\mathsf{T}} v_t| \leq 2bM$$

implying

$$|Z_t| \leq 2b(2M + \mu_{\max}) \doteq c.$$

Thus, using Azuma-Hoeffding inequality, we can conclude that for any $\delta_1 > 0$, and for fixed α,

$$\Pr\left(\left|\sum_{t=1}^{\tau} Z_t\right| > c\sqrt{2\tau\ln(2(2^b)\delta_1^{-1})}\right) \le \delta_1/2^b$$

and (skipping a few intermediate steps involving the union bound as before), we finally get

$$\Pr\left(\forall x \in \mathcal{P},\ \left|\sum_{t=1}^{\tau}(\tilde{w}_t - w_t)^\intercal x\right| \le 2b\tau\mu_{\max} + c\sqrt{2\tau b + 2\tau\ln(2\delta_1^{-1})}\right) \ge 1-\delta_1. \tag{10}$$

Now consider any fixed $x \in \{0,1\}^m$. We claim that the sequence $Y_1,\ldots,Y_\tau$, where $Y_t = x^\intercal \pi^x{}_t(x) - x^\intercal \mu^x{}_t$ is also a bounded martingale difference sequence. Clearly, since $\mathbb{E}_t[\pi^x{}_t(x)] = \mu^x{}_t$, $\mathbb{E}_t[Y_t] = 0$. Further, a bound on the range of the random variables can be computed by observing

$$|x^\intercal \pi^x{}_t(x)| \le N \quad \text{and} \quad |x^\intercal \mu^x{}_t| \le \mu_{\max}.$$

Thus,

$$|Y_t| \le N + \mu_{\max} =: d.$$

An application of Azuma-Hoeffding inequality for the fixed x and for any $\delta_2 > 0$ yields,

$$\Pr\left(\left|\sum_{t=1}^{\tau} Y_t\right| > d\sqrt{2\tau\ln(2(2^m)\delta_2^{-1})}\right) \le \delta_2/2^m.$$

Taking the union bound over all $x \in \{0,1\}^m$,

$$\Pr\left(\forall x \in \{0,1\}^m,\ \left|\sum_{t=1}^{\tau} x^\intercal(\pi^x{}_t(x) - \mu^x{}_t)\right| \le d\sqrt{2\tau m + 2\tau\ln(2\delta_2^{-1})}\right) \ge 1-\delta_2\ .$$

Thus, we get

$$\Pr\left(\forall x \in \{0,1\}^m,\ \left|\sum_{t=1}^{\tau} x^\intercal \pi^x{}_t(x)\right| \le \left|\sum_{t=1}^{\tau} x^\intercal \mu^x{}_t\right| + d\sqrt{2\tau m + 2\tau\ln(2\delta_2^{-1})}\right) \ge 1-\delta_2$$

and finally

$$\Pr\left(\forall x \in \{0,1\}^m,\ \left|\sum_{t=1}^{\tau} x^\intercal \pi^x{}_t(x)\right| \le \tau\mu_{\max} + d\sqrt{2\tau m + 2\tau\ln(2\delta_2^{-1})}\right) \ge 1-\delta_2\ . \tag{11}$$

Setting $\delta_1 = \delta_2 = \frac{\delta}{2}$ in Relations 10 and 11 above and dividing them throughout by τ, we get that, for all $x \in \mathcal{P}$, each of the following two inequalities hold with probability at most $\frac{\delta}{2}$:

$$\begin{aligned}
\frac{1}{\tau}\left|\sum_{t=1}^{\tau} x^\intercal(\tilde{w}_t - w_t)\right| \quad &> \quad 2b\mu_{\max} + \tau^{-1/2}c\sqrt{2b + 2\ln(4\delta^{-1})} \\
\frac{1}{\tau}\left|\sum_{t=1}^{\tau} x^\intercal \pi^x{}_t(x)\right| \quad &> \quad \mu_{\max} + \tau^{-1/2}d\sqrt{2m + 2\ln(4\delta^{-1})}
\end{aligned}$$

From the above relations, we can conclude that, for all $x \in \mathcal{P}$, the following inequality holds with probability at least $1-\delta$

$$\begin{aligned}\frac{1}{\tau}\left|\sum_{t=1}^{\tau} x^{\top}(\tilde{w}_t - w_t - {\pi^x}_t(x))\right| \le (2b+1)\mu_{\max} + \\ \tau^{-1/2}\left(c\sqrt{2b+2\ln(4\delta^{-1})}+d\sqrt{2m+2\ln(4\delta^{-1})}\right)\end{aligned} \tag{12}$$

which yields the desired lemma.

□

Estimating Longest Paths

From Lemma 4.3, we can derive results on estimating the ε-longest paths and the longest path in a manner similar to that employed in Section 4.2. The main difference is that now we view $\sum_{t=1}^{\tau}\tilde{w}_t$ as an estimate of $\sum_{t=1}^{\tau}(w_t + {\pi^x}_t)$ rather than of simply $\sum_{t=1}^{\tau} w_t$.

Thus, we now define the set $\mathcal{S}_\tau^\varepsilon$ as

Definition 4.3 *Define the set of ε-longest paths with respect to the actual delays*

$$\mathcal{S}_\tau^\varepsilon = \left\{ x \in \mathcal{P} : \frac{1}{\tau}\sum_{t=1}^{\tau}(w_t + {\pi^x}_t)^{\top} x \ge \max_{x' \in \mathcal{P}} \frac{1}{\tau}\sum_{t=1}^{\tau}(w_t + {\pi^x}_t(x'))^{\top} x' - \varepsilon \right\}$$

The definition of the set $\tilde{\mathcal{S}}_\tau^\varepsilon$ stays unchanged.

The lemma on approximating the sets $\mathcal{S}$ by $\tilde{\mathcal{S}}$ now becomes the following:

Lemma 4.4 *For any $\varepsilon > 0$ and for*

$$\xi = (2b+1)\mu_{\max} + \tau^{-1/2}\left[c\sqrt{2b+2\ln(4\delta^{-1})}+d\sqrt{2m+2\ln(4\delta^{-1})}\right],$$

we have

$$\tilde{\mathcal{S}}_\tau^\varepsilon \subseteq \mathcal{S}_\tau^{\varepsilon+2\xi} \quad \textit{and} \quad \mathcal{S}_\tau^\varepsilon \subseteq \tilde{\mathcal{S}}_\tau^{\varepsilon+2\xi}$$

with probability at least $1-\delta$.

Under the margin assumption (Assumption 4.1), we can recover the longest path in the general weight-perturbation model, using an identical reasoning as before.

Theorem 4.4 *Suppose Assumption 4.1 holds with $\rho > (4b+2)\mu_{\max}$. We run Algorithm 3 for $\tau = 8(\rho - (4b+2)\mu_{\max})^{-2}\left(c^2(b+\ln(4\delta^{-1})) + d^2(m+\ln(4\delta^{-1}))\right)$ iterations.*

Then with probability at least $1-\delta$, Algorithm 3 outputs

$$x_\tau^* := \arg\max_{x \in \mathcal{P}} x^{\top}\sum_{t=1}^{\tau}\tilde{w}_t$$

and x_τ^* is equal to x^*.

The proofs of Lemma 4.4 and Theorem 4.4 are virtually identical to the corresponding results in Section 4.2, so we omit them here. Also, as in that section, we can also identify the longest path under the weaker Assumption 4.2 by finding the set $\tilde{\mathcal{S}}_\tau^{(4b+2)\mu_{\max}}$ containing $\mathcal{S}_\tau^0$ and enumerating the paths in it.

5 Experimental Results

We have implemented and evaluated our approach for problems in execution time analysis. Our analysis tool, called GAMETIME, can generate an estimate of the execution time profile of the program as well as a worst-case execution time estimate. This section details our implementation and results.

5.1 Implementation

GAMETIME operates in four stages, as described below.

1. Extract CFG. GAMETIME begins by extracting the control-flow graph (CFG) of the real-time task whose WCET must be estimated. This part of GAMETIME is built on top of the CIL front end for C [8]. Our CFG parameters (numbers of nodes, edges, etc.) is thus specific to the CFG representations constructed by CIL. In general, nodes correspond to the start of basic blocks of the program and edges indicate flow of control, with edges labeled by a conditional or basic block. In our experience, this phase is usually fast, taking no more than a minute for any of our benchmarks.

2. Compute basis paths. The next step for GAMETIME is to compute the set of basis paths and the B^+ matrix. This is done essentially as discussed in Section 4, where we also ensure the feasibility of basis paths by the use of integer programming and SMT solving. This phase can be somewhat time-consuming; in our experiments, the basis computation for the largest benchmark (statemate) took about 15 minutes.

3. Generate program inputs. Given the set of basis paths for the graph, GAMETIME then has to generate inputs to the program that will drive the program's execution down that path. It does this using *constraint-based test generation*, by generating a constraint satisfaction problem characterizing each basis path, and then using a constraint solver based on Boolean satisfiability (SAT). This phase uses the UCLID decision procedure [3] to generate inputs for each path and creates one copy of the program for each path, with the different copies only differing in their initialization functions. For our experiments, this constraint-based test generation phase was very quick, taking less than a minute for each benchmark.

4. Predict estimated weight vector or longest path. Finally, Algorithm 2 is run with the set of basis paths and their corresponding programs, along with the B^+ matrix. The number of iterations in the algorithm, τ, depends on the mode of usage of the tool. In the experiments reported below, we used a deterministic cycle-accurate processor simulator, and hence τ was set equal to b, since we perform one simulation per basis path. In general, τ can be pre-computed as described in Section 4 or increased gradually while searching for convergence to a single longest path.

The run-time for this phase depends on the execution time of the program and the number of iterations of the loop in Algorithm 2; for our experiments, this run-time was under a minute for all benchmarks.

Given the estimated weights computed at each round, $\tilde{w}_1, \tilde{w}_2, \ldots, \tilde{w}_\tau$, we can compute the overall estimated weight vector $\frac{1}{\tau}\sum_{t=1}^{\tau}\tilde{w}_t$, and use this to predict the length of any path in the program. In particular, we can predict the longest path, and its corresponding length. Alternatively, the predicted longest path can be executed (or simulated) several times to calculate the desired timing estimate.

5.2 Benchmarks

Our benchmarks were drawn from those used in the *WCET Challenge 2006* [23], which were drawn from the Mälardalen benchmark suite [15] and the PapaBench suite [18]. In particular, we used benchmarks that came from real embedded software (as opposed to toy programs), had non-trivial control flow, and did not require automatic estimation of loop bounds. The latter criterion ruled out, for example, benchmarks that compute a discrete cosine transform or perform data compression, because there is usually just one path through those programs (going through several iterations of a loop), and variability in run-time usually only comes from characteristics of the data. Most benchmarks in the Mälardalen suite are of this nature.

The main characteristics of the chosen benchmarks is shown in Table 1. The first three benchmarks, altitude, stabilisation, and climb_control, are tasks in the open source PapaBench software for an unmanned aerial vehicle (UAV) [18]. The last benchmark, statemate, is code generated from a Statemate Statecharts model for an automotive window control system. Note in particular, how the number of basis paths b is far less than the total number of source-sink paths in the CFG. (We are able to efficiently count the number of paths as the CFG is a DAG.) We also indicate the number of lines of code for each task; however, note that this is an imprecise metric as it includes declarations, comment lines, and blank lines – the CFG size is a more accurate representation of size.

Name	LOC	Size of CFG		Total Num.	Num. of basis
		n	m	of paths	paths b
altitude	12	12	16	11	6
stabilisation	48	31	39	216	10
climb_control	43	40	56	657	18
statemate	916	290	471	7×10^{16}	183

Table 1: **Characteristics of Benchmarks.** "LOC" indicates number of lines of C code for the task. The Control-Flow Graph (CFG) is constructed using the CIL front end, n is the number of nodes, m is the number of edges.

5.3 Worst-Case Execution Time Analysis

We have compared GameTime with leading tools for WCET analysis. We present here a comparison with Chronos [13]. These tools are based on models crafted for particular architectures, and are designed to generate conservative (over-approximate) WCET bounds. Although GameTime is not guaranteed to generate an upper bound on the WCET, we have found that GameTime can produce larger WCET estimates than these tools. We also show that GameTime does significantly better than simply testing the programs with inputs generated uniformly at random.

5.3.1 Comparison with Chronos and Random Testing

We performed experiments to compare GameTime against Chronos [13] as well as against testing the programs on randomly-generated inputs. WCET estimates are output in terms of the number of CPU cycles taken by the task to complete in the worst-case.

Chronos is built upon SimpleScalar [25], a widely-used tool for processor simulation and performance analysis. Chronos extracts a CFG from the binary of the program (compiled for MIPS using modified SimpleScalar tools), and uses a combination of dataflow analysis, integer programming, and manually constructed processor behavior models to estimate the WCET of the task.

To compare GameTime against Chronos, we used SimpleScalar to simulate, for each task, each of the extracted basis paths. We used the same SimpleScalar processor configuration as we did for Chronos (which is Chronos' default configuration), specified below:

```
-cache:il1 il1:16:32:2:l -mem:lat 30 2 -bpred 2lev -bpred:2lev 1 128 2 1 -decode:width 1 -issue:width
1 -commit:width 1 -fetch:ifqsize 4 -ruu:size 8
```

Since SimpleScalar's execution is deterministic for a fixed processor configuration, we did not run Algorithm 2 in its entirety. Instead, we simulated each of the basis paths exactly once (factoring out the time for initialization code) and then predicted the longest path as described in Section 4. The predicted longest path was then simulated once and its execution time is reported as GameTime's WCET estimate.

The random testing was done by generating initial values for each program input variable uniformly at random from its domain. For each benchmark, we generated 500 such random initializations; note that GameTime performs significantly fewer simulations (only as many as there are basis paths, for a maximum of 183 for the statemate benchmark).

Name of Benchmark	Chronos WCET T_c	Random testing T_r	GameTime estimate T_g	$\frac{T_c - T_g}{T_g}$ (%)	Basis path times	
					Max	Min
altitude	567	175	348	62.9	343	167
stabilisation	1379	1435	1513	-8.9	1513	1271
climb_control	1254	646	952	31.7	945	167
statemate	8584	4249	4575	87.6	3735	3235

Table 2: **Comparison with Chronos and random testing.** Execution time estimates are in number of cycles reported by SimpleScalar. For random testing, the maximum cycle count over 500 runs is reported. The fifth column indicates the percentage over-estimation by Chronos over GameTime, and the last two columns indicate the maximum and minimum cycle counts for basis paths generated by GameTime.

Our results are reported in Table 2. We note that the estimate of GameTime T_g is lower than the WCET T_c reported by Chronos for three out of the four benchmarks. Interestingly, $T_g > T_c$ for the stabilisation benchmark; on closer inspection, we found that this occurred mainly because the number of misses in the instruction cache was significantly underestimated by Chronos. The over-estimation by Chronos for statemate is very large, much larger than for altitude and climb_control. This appears to arise from the fact that the number of branch mis-predictions estimated by Chronos is significantly larger than that actually occurring: 106 by Chronos versus 19 mis-predictions on the longest path simulated by GameTime in SimpleScalar. In fact, the number of branches performed in a single loop of the statemate code is bounded by approximately 40.

We also note that GameTime's estimates can be significantly higher than those generated by random testing. Moreover, GameTime's predicted WCET is higher than the execution time of any of the basis paths, indicating that the basis paths taken together provide more longest path information than available from them individually.

5.4 Estimating the Full Timing Profile of a Program

One of the unique aspects of GameTime is the ability to predict the *execution time profile of a program* – the distribution of execution times over program paths – as formalized in Lemma 4.3.

To experimentally validate this ability, we performed experiments with a complex processor architecture – the StrongARM-1100 – which implements the ARM instruction set with a complex pipeline and both data and instruction caches. The SimIt-ARM cycle-accurate simulator [19] was used in these experiments.

In our experiments, we first executed each basis path generated by GameTime on the SimIt-ARM simulator and generated the averaged estimated weight vector $\tilde{w}_{avg} = \frac{1}{b}\sum_{t=1}^{b}\tilde{w}_t$. Using this estimated weight vector as the weights on edges in the CFG, we then efficiently computed the estimated length of each path x in the CFG as $x \cdot \tilde{w}_{avg}$ using dynamic programming. We also exhaustively enumerated all program paths for the small programs in our benchmark set, and simulated each of these paths to compute its execution time.

For the altitude program, the histogram of execution times generated by GAMETIME perfectly matched the true histogram generated by exhaustively enumerating program paths.

For the climb_control task, the GAMETIME histogram is a close match to the true histogram, as can be seen in Figure 5. Out of a total of 657 paths, 129 were found to be feasible; of these, GAMETIME's prediction differs from the true execution time on only 12 paths, but the prediction is never off by more than 20 cycles.

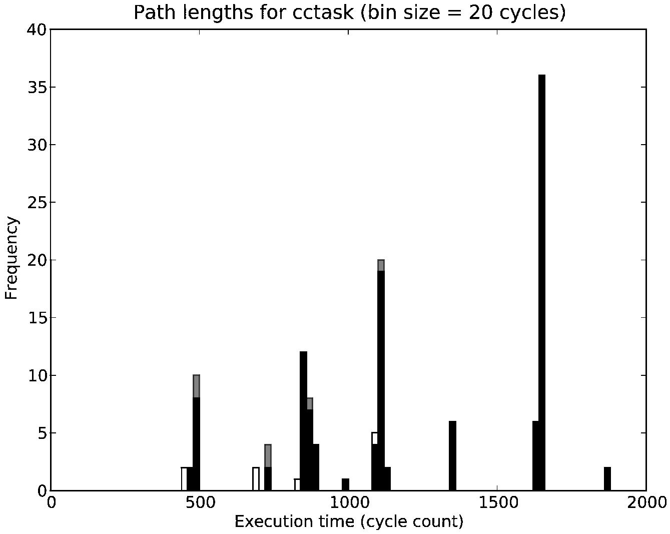

Figure 5: **Estimating the distribution of execution times with GAMETIME.** The true execution times are indicated by white bars, the predicted execution times by gray bars, and the cases where the two coincide are colored black.

In summary, we have found GAMETIME to be an adequate technique to estimate not just the WCET, but also the distribution of execution times of a program, for even complex microprocessor platforms. A key aspect of GAMETIME's effectiveness has been the generation of tests for basis paths. We have also experimented with other coverage metrics such as statement coverage, but these do not yield the same level of accuracy as do basis path coverage. Full path coverage is very difficult to achieve for programs that exhibit path explosion (e.g., statemate), while the number of basis paths remains tractable.

6 Conclusions

We have presented a new, game-theoretic approach to estimating quantitative properties of a software task, such as its execution time profile and worst-case execution time (WCET). Our tool, GAMETIME, is measurement-based, making it easy to use on many different platforms without the need for tedious processor behavior analysis. We have presented both theoretical and experimental results for the utility of the GAMETIME approach for quantitative analysis, in particular for timing estimation.

We note that our algorithm and results of Section 4 are general, in that they apply to estimating longest paths in DAGs in an unpredictable environment, not just to timing estimation for embedded software. One could apply the algorithms presented in this paper to quantitative analysis of many systems with suitable graph models. Several potential applications are worth exploring, including timing analysis of combinational circuits and distributed embedded and control systems, as well as power estimation of embedded systems.

Acknowledgments

We are grateful to Subramani Arunkumar, Richard Karp, Edward Lee, and Pravin Varaiya for valuable discussions and feedback. We thank Bharathi Seshadri, Susmit Jha, and Min Xu for their inputs. Andrew Chan helped generate the experimental data presented in Section 5.4. The first author was supported in part by NSF CAREER grant CNS-0644436 and an Alfred P. Sloan Fellowship, and the second author by DARPA grant FA8750-05-2-0249.

References

[1] Baruch Awerbuch and Robert 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] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 4, chapter 8. IOS Press, 2009.

[3] Randal E. Bryant, Daniel Kroening, Joel Ouaknine, Sanjit A. Seshia, Ofer Strichman, and Bryan Brady. Deciding bit-vector arithmetic with abstraction. In *TACAS*, volume 4424 of *LNCS*, pages 358–372, 2007.

[4] Nicolò Cesa-Bianchi and Gábor Lugosi. *Prediction, Learning, and Games*. Cambridge University Press, 2006.

[5] Arindam Chakrabarti, Krishnendu Chatterjee, Thomas A. Henzinger, Orna Kupferman, and Rupak Majumdar. Verifying quantitative properties using bound functions. In *Proc. Correct Hardware Design and Verification Methods*, pages 50–64, 2005.

[6] Stephen A. Edwards and Edward A. Lee. The case for the precision timed (PRET) machine. In *Design Automaton Conference (DAC)*, pages 264–265, 2007.

[7] David Eppstein. Finding the k shortest paths. *SIAM Journal on Computing*, 28(2):652–673, 1998.

[8] George Necula et al. CIL - infrastructure for C program analysis and transformation. `http://manju.cs.berkeley.edu/cil/`.

[9] András György, Tamás Linder, Gábor Lugosi, and György Ottucsák. The on-line shortest path problem under partial monitoring. *J. Mach. Learn. Res.*, 8:2369–2403, 2007.

[10] Sandy Irani, Gaurav Singh, Sandeep Shukla, and Rajesh Gupta. An overview of the competitive and adversarial approaches to designing dynamic power management strategies. *IEEE Trans. VLSI*, 13(12):1349–1361, Dec 2005.

[11] Raimund Kirner and Peter Puschner. Obstacles in worst-case execution time analysis. In *ISORC*, pages 333–339, 2008.

[12] Edward A. Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. Technical Report UCB/EECS-2007-72, University of California at Berkeley, May 2007.

[13] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. Technical report, National University of Singapore, 2005. `http://www.comp.nus.edu.sg/~rpembed/chronos/chronos_tool.pdf`.

[14] Yau-Tsun Steven Li and Sharad Malik. *Performance Analysis of Real-Time Embedded Software*. Kluwer Academic, 1999.

[15] Mälardalen WCET Research Group. The Mälardalen benchmark suite. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`.

[16] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[17] H. Brendan McMahan and Avrim Blum. Online geometric optimization in the bandit setting against an adaptive adversary. In *COLT'04*, pages 109–123, 2004.

[18] Fadia Nemer, Hugues Cass, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne De Michiel. Papabench: A free real-time benchmark. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[19] Wei Qin and Sharad Malik. Simit-ARM: A series of free instruction-set simulators and micro-architecture simulators. `http://embedded.eecs.berkeley.edu/mescal/forum/2.html`.

[20] 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.

[21] Herbert Robbins. Some aspects of the sequential design of experiments. *Bull. Amer. Math. Soc.*, 58(5):527–535, 1952.

[22] Sanjit A. Seshia and Alexander Rakhlin. Game-theoretic timing analysis. In *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 575–582, 2008.

[23] Lili Tan. The Worst Case Execution Time Tool Challenge 2006: Technical Report for the External Test. Technical Reports of WCET Tool Challenge 1, Uni-DUE, December 2006.

[24] Vivek Tiwari, Sharad Malik, and Andrew Wolfe. Power analysis of embedded software: a first step towards software power minimization. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 384–390, 1994.

[25] Todd Austin et al. The SimpleScalar tool set. `http://www.simplescalar.com`.

[26] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. Measurement-based timing analysis. In *Proc. 3rd Int'l Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, 2008.

[27] Reinhard Wilhelm. Determining Bounds on Execution Times. In R. Zurawski, editor, *Handbook on Embedded Systems*. CRC Press, 2005.

A Azuma-Hoeffding Inequality

The Azuma-Hoeffding inequality is a very useful concentration inequality. A version of this inequality with a slightly better constant is given as Lemma A.7 in [4].

Lemma A.1 *Let $Y_1, \ldots, Y_T$ be a martingale difference sequence. Suppose that $|Y_t| \leq c$ almost surely for all $t \in \{1, \ldots, \tau\}$. Then for any $\delta > 0$,*

$$\Pr\left(\left|\sum_{t=1}^{\tau} Y_t\right| > \sqrt{2\tau c^2 \log(2/\delta)}\right) \leq \delta$$

One-sided inequalities for $\sum_{t=1}^{\tau} Y_t$ also hold by replacing $2/\delta$ with $1/\delta$ in the logarithm. The inequality is an instance of the so-called *concentration of measure inequalities*. Roughly speaking, it says that if each random variable fluctuates within the bounds $[-c, c]$, then the sum of these variables fluctuates, with high probability, within $[-c\sqrt{\tau}, c\sqrt{\tau}]$.