Analyzing Data-Dependent Timing and Timing Repeatability with GameTime

Zachariah Wasson

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2014-132
http://www.eecs.berkeley.edu/Pubs/TechRpts/2014/EECS-2014-132.html

May 30, 2014

Copyright © 2014, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Analyzing Data-Dependent Timing and Timing Repeatability with GameTime

by Zachariah Lord Wasson II

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Sanjit Seshia
Research Advisor

Date

* * * * * *

Edward Lee
Second Reader

Date

Abstract

Analyzing Data-Dependent Timing and Timing Repeatability with GameTime

by

Zachariah Lord Wasson II

Master of Science in Electrical Engineering and Computer Science

University of California, Berkeley

Professor Sanjit Seshia, Chair

Execution time analysis is central to the design and verification of real-time embedded systems. One approach to this problem is GameTime, a toolkit for timing analysis, which predicts the timing of various program paths using platform measurements of a special subset of paths. This thesis explores extensions to GameTime to handle two common sources of data-dependent timing behavior: instructions with variable timing and load/store dependencies. We present a technique for automatically learning a model of the data dependencies and encoding this model into the code under analysis for processing by GameTime. Using these extensions, we show that GameTime more accurately predicts the timing for a variety of benchmarks.

Unfortunately, the complexity of modern architectures and platforms has made it very difficult to obtain accurate and efficient timing estimates. To deal with this, there have been recent proposals to re-architect platforms to make execution time of instructions more repeatable. There is however no systematic formalization of what timing repeatability means. In this thesis, we also propose formal models of timing repeatability. We give an algorithmic approach to evaluate parameters of these formal models. Using GameTime along with the data-dependent extensions discussed in this thesis, we objectively evaluate the timing repeatability of a representative sample of platforms with respect to a program of interest.

Contents

Contents i

List of Figures ii

List of Tables iii

1 Introduction 1
1.1 Related Work . . . 3

2 Background 5
2.1 GameTime . . . 5
2.2 Illustrative Example . . . 6
2.3 Basis Paths . . . 8

3 Data Dependency 10
3.1 Problem Formulation . . . 10
3.2 Implementation . . . 12

4 Timing Repeatability 16
4.1 Models of Timing Repeatability . . . 16
4.2 Implementation . . . 20

5 Experimental Results 23
5.1 Platform Details . . . 24
5.2 Benchmarks . . . 24
5.3 Data Dependency Results . . . 25
5.4 Timing Repeatability Results . . . 32

6 Conclusion and Future Work 38

Bibliography 39

List of Figures

2.1 Overview of the GameTime approach 6
2.2 Both programs compute the value of `base`$^{\texttt{exponent}}$ modulo `p`. 7
2.3 CFG and Basis Paths for Code in Figure 2.2(b) 7

3.1 Toy example illustrating load/store dependency 11
3.2 Comparison of control-flow graphs with and without consideration for load/store dependencies 12
3.3 Overview of the automatic generation of cases for data-dependent operations . . 14
3.4 Annotated version of the toy example in Figure 3.1 that explicitly splits on recently used variable indices 15

5.1 Exact representation of divide cases for ARM Cortex-M3 26
5.2 Various approximations of divide using DTL ($\log_2$ scale) 28
5.2 Various approximations of divide using DTL ($\log_2$ scale) 29
5.3 Multiply results using DTL approach 30
5.4 Modexp prediction accuracy with and without `divide_dtl_random`; the red dots represents the absolute error between the true measurements and normal predictions, while the blue dots represents the absolute error between the true measurements and predictions using `divide_dtl_random` 31
5.5 SimIt-ARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after an array has been sorted 35
5.6 SimIt-ARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after two matrices have been multiplied 36
5.7 PTARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after an array has been sorted 36
5.8 PTARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after two matrices have been multiplied 37

List of Tables

5.1 Characteristics of benchmarks where *LOC* is lines of code, n is the number of nodes, m is the number of edges, *Total* is the total number of feasible paths in the CFG, and b is the number of basis paths 25
5.2 $\pi_{\max}^{\text{norm}}$ results for the PTARM and ARM Cortex-M4 platforms (the $\pi_{\max}^{\text{norm}}$ results are scaled by 1000 for readability) 33

Acknowledgments

There have been several people that helped make this thesis possible. First, and foremost, I would like to thank my advisor, Professor Sanjit Seshia, for guiding and supporting me throughout my graduate career. He has been an amazing mentor since I started at Berkeley, making a significant effort to understand my interests and helping to shape my career plans. In addition, I would like to thank my other reader, Professor Edward Lee, for taking the time to provide feedback and comments. I would also like to acknowledge Jonathan Kotker for his knowledge of the internals of GameTime and his expertise in timing analysis, which have been invaluable resources. I am grateful to the people at Toyota Technical Center and Chrona for working with me on various experiments using GameTime. My fiancé, Jaclyn Leverett, has also been an incredible support; she is always willing to listen when I need talk out an idea, and she provides valuable insights and solutions to the problems I encounter. Finally, I would like to thank my father, Zach, my mother, Linda, and my sister, Katie, for always encouraging me to pursue my interests. This work would not have been possible without the generous funding from CASIO, TerraSwarm, and NSF awards #0931843 and #0644436.

Chapter 1

Introduction

Execution time analysis is central to the design and verification of real-time embedded systems. System designers are often interested in finding the worst-case execution time (WCET) of a program or a timing bound on a particular path of the program. This information can be used to check the schedulability of the system or optimize a specific path. One of the biggest challenges is the increasing complexity of the underlying hardware platform (e.g., out-of-order execution, deep pipelines, branch prediction, caches, parallelism, power management functions). For programs that have a large execution path space, the challenge of execution time analysis is further compounded. As pointed out by several researchers (e.g., [9, 8]), the interaction of a large space of program paths with the complexity of underlying platform results in timing estimates that are either too pessimistic (due to conservative modeling) or too optimistic (due to unmodeled features of the platform).

One approach to this problem is GameTime, a toolkit for timing analysis. In [21, 22], Seshia and Rakhlin establish the theory behind GameTime. A major advantage of the GameTime approach over other timing analysis methods is that it uses systematic testing and game-theoretic prediction to estimate timing on any platform with no modification. One of the key assumptions is that the timing of a program depends only on its *execution context*, which is comprised of two elements: the starting platform state and the starting program state. While this assumption holds for many real-world programs, there are programs that exhibit data-dependent behavior. The current GameTime algorithm can only reason in a data-dependent manner about variables that determine which path a program will take; however there are many other forms of data-dependency that GameTime cannot handle. In this thesis, we present the various ways that data-dependency may manifest itself in a program along with extensions to GameTime to handle these cases. These extensions provide the user a way to specify more detailed information about a given platform without requiring a complete and complex model of the entire platform.

While timing analysis tools aid us in determining the WCET for a given program, they do not solve the inherit issues that make timing analysis so difficult. One solution to this problem is to rethink the design of embedded processors so as to make their timing more *repeatable* and *predictable* (see, e.g., [5]). Although there has been considerable research

in this area, the notion of repeatable timing has yet to be formalized. One rather strict interpretation is to define a platform to have repeatable timing when the execution time of an instruction is completely independent of the state of the platform and the arguments to that instruction. In other words, the instruction takes the same execution time, no matter when and how it is executed. However, this strict notion of "perfect repeatability" may be rather hard to achieve. Indeed, we find empirically that even some of the most promising architectures proposed for achieving repeatable timing do not achieve perfect repeatability.

We present an approach to quantify and evaluate timing repeatability from a "programmer's perspective." By this, we mean that we provide the author of a program a way to evaluate the timing repeatability of a platform for that specific program. Informally, we say that a platform is *timing-repeatable* for a program if the execution time of any fragment of that program on that platform depends minimally on its *execution context* as defined above (the starting platform state and the starting program state). Our approach operates by timing the entire program on a polynomial number of systematically-generated test cases, rather than having to perform tedious and tricky measurements of individual instruction timing. Specifically, our approach to evaluate timing repeatability is based on the notion of *basis paths*, a subset of program paths that spans the space of all program paths [21, 22]. The approach we use here is inspired by the GameTime approach, particularly in its use of basis paths, and utilizes our data-dependent extensions to accurately measure timing repeatability.

Traditional WCET analysis focuses on finding safe (conservative) upper bounds on the true WCET (and similarly, lower bounds on BCET). Finding conservative bounds can be an effective approach for ensuring safe real-time behavior. However, it does not estimate timing repeatability. Estimating timing repeatability requires measuring the *variation* in program timing with respect to changes in platform and program state. The difference between a lower bound on BCET and an upper bound on WCET is a rather coarse measure of such variation, and it is also sensitive to the approximations made by specific tools. In contrast, we propose a tool-independent quantitative definition of timing repeatability, and an approach for estimating it that is easy to port to any platform and program.

Chapter 2 describes the basic algorithm behind GameTime and provides a detailed look at *basis paths*. In Chapter 3, we introduce the topic of data-dependency. In particular, we outline a method for automatically generating cases for an arbitrary data-dependent operation. We also discuss methods for extending GameTime to handle data-dependency. This is based on joint work with Sanjit Seshia. We formalize timing repeatability and provide algorithms for computing two metrics to measure the repeatability of a platform in Chapter 4. This is based on joint work with Jonathan Kotker and Sanjit Seshia. Chapter 5 contains our evaluation of the algorithms discussed in the previous two chapters with various benchmarks on multiple real-world platforms. Finally, in Chapter 6, we summarize this work and discuss future directions.

In summary, this thesis makes the following novel contributions.

- A formalization of the various types of data-dependency along with extensions for

GameTime to handle these features (Chapter 3);

- A formalization of timing repeatability, both from the programmer's perspective and the architect's perspective along with algorithmic techniques for evaluating timing repeatability, particularly for a programmer's perspective (Chapter 4); and
- An experimental evaluation of data-dependency and timing repeatability for various programs on an assortment of processor platforms (ARM Cortex-M3, ARM Cortex-M4, and PTARM) (Chapter 5).

1.1 Related Work

Timing analysis has become a topic of great interest for many researchers over the last few decades, spurred forward due to the growing prevalence of cyber-physical systems in every aspect of our lives. Li and Malik [10] and Wilhelm et al [17] provide comprehensive surveys for the current tools and techniques in WCET analysis. Many current approaches to the problem, such as [26], involve abstraction based techniques where the underlying hardware is modeled manually. A major drawback to these techniques is that each target platform must be carefully modeled in order to generate reliable upper bounds, whereas GameTime requires no modification to run on any platform as discussed above. In [21], Seshia and Rakhlin compare many of these techniques to the GameTime approach.

There have been various works related to building *repeatable* and *predictable* architectures. Whitham and Audsley [25] have proposed MCGREP, a predictable and reconfigurable CPU architecture that is entirely microprogrammed. Andalam et al. present an architecture called ARPRET [1], which executes PRET-C, an extension to C which provides synchronous concurrency and high-level timing constructs. The MERASA project [24] is developing multi-core architectures alongside the tools and techniques for timing analysis to ensure predictability of the entire processor. The Heracles project [7] also focuses on timing-predictable multi-core architectures, and the Verilog designs along with a cycle-accurate simulation environment are publicly available. PTARM [12], an ARM-based precision-timed architecture, achieves repeatable timing without sacrificing performance through hardware techniques that are both predictable and repeatable. In the T-CREST project, the hardware/software architecture [19] and the code-generation [15] are considered together for time-predictable embedded systems.

There has also been prior work on formalizing *predictable timing*. As noted by Liu et al. [12] and Schoeberl [18], predictability, although related, is different from repeatability: the former is concerned with whether a timing analysis tool can accurately predict program timing, whereas the latter is concerned with whether the timing exhibited by a platform is repeatable. Proposals for quantifying predictable timing include various relations between the WCET, the best-case execution time (BCET), and bounds for WCET and BCET found by a timing analysis tool: for example, the difference between the true WCET and the bound found by a tool has been proposed as one metric. See [18] for other metrics. Such metrics

provide insight into predictions about timing made by a *specific timing analysis technique* for a given program and platform. Timing repeatability, in contrast, is a property of the program and platform alone.

Chapter 2

Background

In this chapter, we briefly cover the typical assumptions of timing analysis. We then provide an overview of GameTime followed by a detailed description of basis paths, a key component of the GameTime algorithm.

2.1 GameTime

We limit ourselves to sequential programs `P` where loops have statically-derivable or known finite loop bounds and function calls have finite recursion depths. Thus `P` can be unrolled to an equivalent program `Q` where every execution path in the (possibly cyclic) control-flow graph (CFG) of `P` is mapped one-to-one to a path in the acyclic control-flow graph of `Q`. We further assume that the program runs uninterrupted.

The basic flow of GameTime is shown in Figure 2.1. The input to GameTime is a C source file. This program is first passed through a preprocessing stage where loops are unrolled and functions are inlined. A control-flow graph is then constructed from the modified source file. Since all loops are unrolled and the program is assumed to have finite recursion, the CFG is a directed acyclic graph. In addition, dummy start and end nodes are added if the CFG does not have a single entry and exit point.

Once the CFG has been extracted, GameTime generates a subset of paths, known as basis paths, which will be discussed in Section 2.3. Each candidate basis path is checked for feasibility by formulating a satisfiability modulo theories (SMT) formula that encodes the semantics of the path; a satisfying assignment yields a test case that drives execution down that path. If no satisfying inputs are found, the candidate path is discarded and the search continues.

After a suitable set of basis paths has been generated, the program is compiled for a given platform, and the execution time for each basis path is measured on the platform itself. The learning algorithm uses these end-to-end execution time measurements to infer edge weights on the CFG, which it can then use to estimate timing values for other paths, including the WCET path.

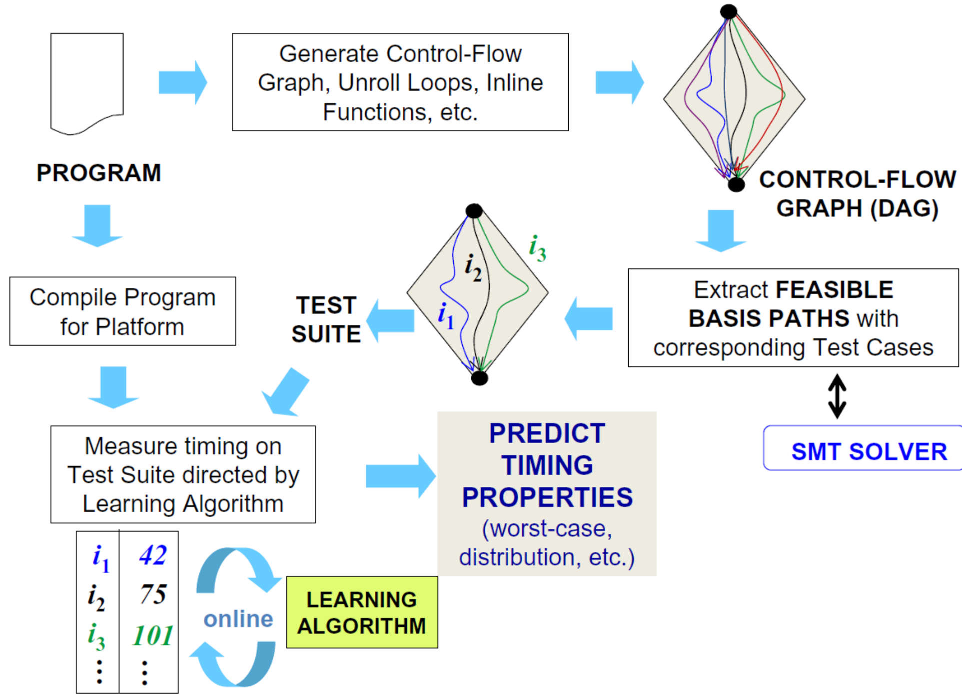


Figure 2.1: Overview of the GameTime approach

2.2 Illustrative Example

Our example is the modular exponentiation code given in Figure 2.2, drawn from [20]. 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 p. In this particular benchmark, we use the *square-and-multiply* method to perform the modular exponentiation, based on the observation that

$$b^e = \begin{cases} (b^2)^{e/2} = (b^{e/2})^2, & e \text{ is even,} \\ (b^2)^{(e-1)/2} \cdot b = (b^{(e-1)/2})^2 \cdot b, & e \text{ is odd.} \end{cases} \tag{2.1}$$

The unrolled version of the code of Figure 2.2(a) for a 2-bit exponent is given in Figure 2.2(b).

In the CFG extracted from a program, nodes correspond to program counter locations, and edges correspond to basic blocks or branches. Note that this is slightly different than the standard representation, in which basic blocks are nodes, but both are equivalent.

Figure 2.3(a) denotes the control-flow graph for the code in Figure 2.2(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 if and only if the ith edge is on the path (and 0 otherwise). For example, in the graph of Figure 2.3(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 the `exponent` $= 0$) and 'then' branches of

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

(a) Original code `P`

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

(b) Unrolled code `Q`

Figure 2.2: Both programs compute the value of $\texttt{base}^{\texttt{exponent}}$ modulo `p`.

the condition statements at lines 3 and 9 respectively in the code, while edge 5 corresponds to the basic block comprising of lines 6 and 7. We denote by $\mathcal{P}$ the subset of $\{0,1\}^m$ that correspond to valid program paths. Note that this set can be exponentially large in m.

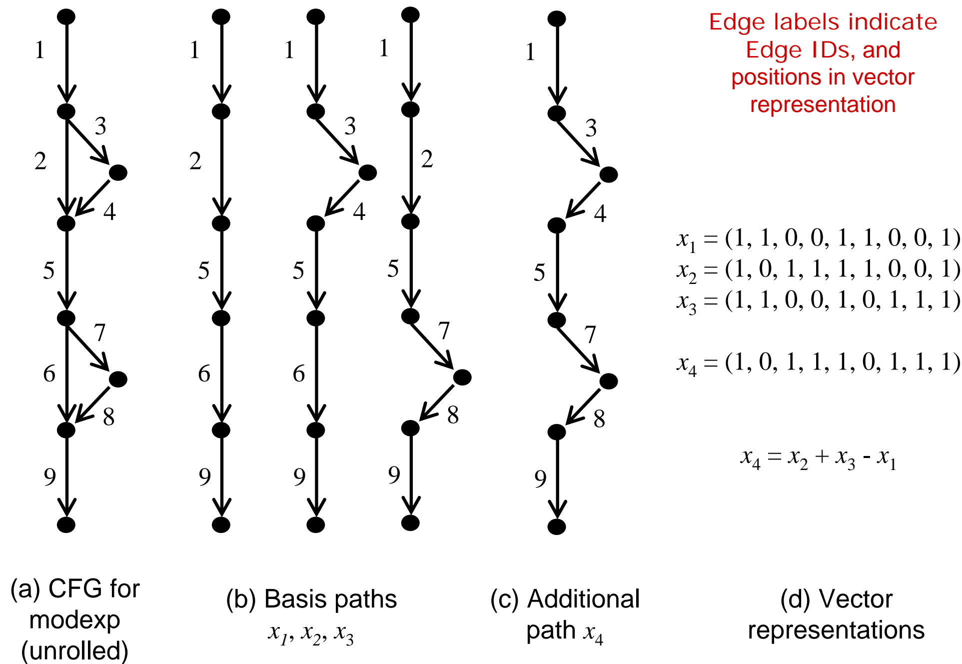


Figure 2.3: CFG and Basis Paths for Code in Figure 2.2(b)

2.3 Basis Paths

A central notion used in GameTime and in our algorithm for evaluating timing repeatability is that of a *basis path*. As noted above, each program path can be modeled as a 0-1 vector; thus, we can think of computing a basis of this path-space in the sense that any path in the graph can be written as a linear combination of the paths in the basis.

Figure 2.3(b) shows the basis paths for the graph of Figure 2.3(a). Here x_1, x_2, and x_3 are the paths corresponding to `exponent` taking values 00, 10, and 01 respectively. Figure 2.3(c) shows the fourth path x_4, expressible as the linear combination $x_2+x_3-x_1$ (see Figure 2.3(d)).

The number of feasible basis paths b is bounded by $m - n + 2$ (where n is the number of CFG nodes). In general, the number of basis paths b is less than m. 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.

Algorithm 1 Finding a set of Basis Paths

1: $(\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m) \leftarrow (\mathbf{e}_1, \ldots, \mathbf{e}_m)$.
2: **for** $i = 1$ to m **do** {{Compute a basis of $\mathcal{P}$}}
3: $\quad \boldsymbol{b}_i \leftarrow \arg\max_{\boldsymbol{x} \in \mathcal{P}} |\det(B_{\boldsymbol{x},i})|$
4: $\quad$ Use symbolic execution and SMT solving to determine if $\boldsymbol{b}_i$ is feasible
5: $\quad$ **if** $\boldsymbol{b}_i$ is infeasible **then**
6: $\quad\quad$ Use the unsatisfiable core to rule out the subset of edges on the path that are infeasible
7: $\quad$ **else**
8: $\quad\quad$ Include basis path $\boldsymbol{b}_i$
9: $\quad$ **end if**
10: **end for**

Algorithm 1 shows the algorithm used to compute basis paths, as adapted from [21]. A set of b paths $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_b\} \subseteq \mathcal{P}$ is called a *basis* if any path $\boldsymbol{x} \in \mathcal{P}$ can be written as a linear combination of basis paths: $\boldsymbol{x} = \sum_{i=1}^{b} \alpha_i \boldsymbol{b}_i$. The running time of Algorithm 1 is polynomial if all program paths are known a priori to be feasible; however, in general, this is not known, and the general problem is NP-hard. Nevertheless, given the major advances in SAT/SMT solving over the last decade [13], the computation is very efficient in practice. Also, it is important to note that Algorithm 1 does not enumerate all paths explicitly; similar to the implicit path enumeration (IPET) method used by WCET tools, it symbolically represents the space of all paths and generates a basis from this symbolic representation.

In Algorithm 1, $B = [\boldsymbol{b}_1, \ldots, \boldsymbol{b}_m]^\intercal$ and $B_{\boldsymbol{x},i} = [\boldsymbol{b}_1, \ldots, \boldsymbol{b}_{i-1}, \boldsymbol{x}, \boldsymbol{b}_{i+1}, \ldots, \boldsymbol{b}_m]^\intercal$. B is initialized so that its ith row is e_i, the standard basis vector with 1 in the ith position and 0s elsewhere. The output of the algorithm is the final value of B. The ith iteration of the for-loop in lines 2-10 attempts to replace the ith element of the standard basis with a path that is linearly independent of the previous $i - 1$ paths identified so far and with all remaining

standard basis vectors. Line 3 of the algorithm corresponds to maximizing a linear function over the set of $\mathcal{P}$, and can be solved using LONGEST-PATH.

As noted above, each *basis path* is tested for feasibility by formulating an SMT formula that encodes the semantics of that path; thus, a satisfying assignment yields a test case that drives execution down that path. Test cases generated in this manner are used in the learning algorithm of GameTime as well as for evaluating timing repeatability in our experiments.

Chapter 3

Data Dependency

In this chapter, we discuss two key types of data dependency that appear regularly–data dependent instructions and load/store dependencies.

3.1 Problem Formulation

Data Dependent Instructions

Some architectures take a variable amount of time for certain instructions depending on the arguments provided to the instruction. We refer to these instructions as data dependent, since the timing depends on the input data. Examples of these data dependent instructions include multiply and divide on ARM Cortex-M3 microprocessors [3].

Oftentimes the cycle counts are provided in the technical reference manuals of these microarchitectures; however the actual cases are not always explicitly stated. We must infer the various conditions that result in different cycle counts in this case. Thus we may treat a data dependent instruction as the following function we would like to learn,

$$f(x_1, x_2, \ldots, x_n) = c$$

where x_i is the ith input argument and c is the cycle count. Furthermore, we can assume we have a method to measure cycle counts from the actual microarchitecture, since this is a requirement for utilizing GameTime for a given platform. Therefore, we can generate as many test cases as needed for $f(\boldsymbol{x})$, and use that as a training set for learning the function.

Since GameTime currently operates at the C source code level, we can expand the definition of 'instruction' to include any operation with timing variability due to its inputs that is not explicitly stated in the code itself, e.g., via if/then/else constructs. For instance, on architectures without a hardware divide instruction, the mod operator becomes an assembly routine. Upon inspection, we would see that this assembly routine features multiple branches based on the inputs and intermediate results, leading to data-dependent timing. However, since the data-dependency is only apparent at the assembly level, GameTime lacks the fa-

```
1  uint8_t compare(uint8_t A[], uint8_t x, uint8_t y) {
2      uint8_t ret_val;
3      uint8_t ax = A[x];
4      uint8_t ay = A[y];
5      if (ax == ay) {
6          ret_val = 1;
7      } else {
8          ret_val = 0;
9      }
10     return ret_val;
11 }
```

Figure 3.1: Toy example illustrating load/store dependency

cilities to reason about these cases. We will use 'instruction' and 'operation' interchangeably when discussing data-dependent instructions due to this equivalence.

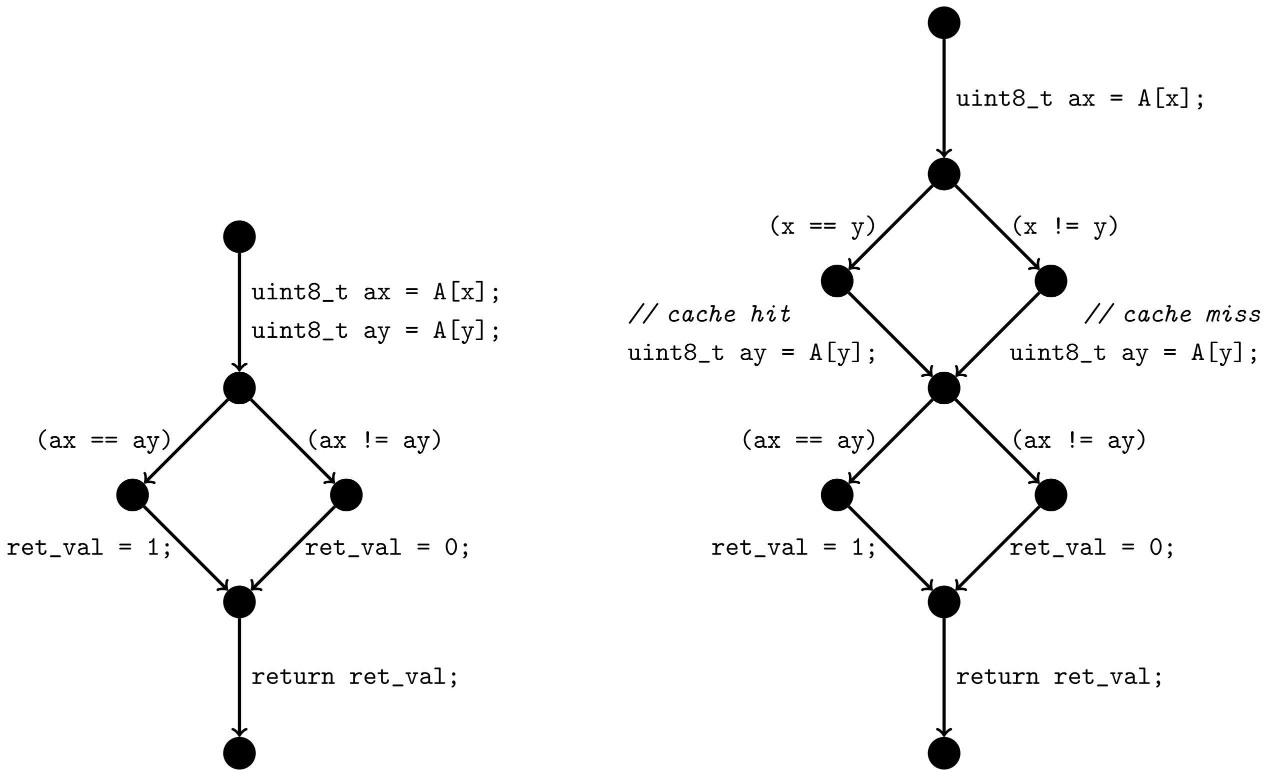
Load/Store Dependencies

When an architecture incorporates a cache or similar mechanism, data dependency can be introduced in variable indexed arrays, since the timing for reading an element from an array depends on whether the element is in the cache or not. This type of data dependency is illustrated in Figure 3.1. This function takes an array as input along with two indices and compares the values at those indices for equality.

Figure 3.2(a) shows the control flow graph that GameTime generates from this C code. However, if we consider an architecture with some form of caching, then there could be drastically different timing depending on if certain elements are already in the cache or not. When `(x == y)` in the code snippet above, `A[x]` is loaded into the cache when it is read; this results in a cache hit when `A[y]` is read. On the other hand, a cache miss will likely occur for all cases where `(x != y)`. Thus we would like GameTime to generate a CFG that matches Figure 3.2(b) in order to reason about these inherit conditionals.

This type of data dependency can become more complex than the illustration above. For instance, we may have a cache with a larger line size than the elements in the array; thus we can hold multiple elements in single cache line. Consider the toy example above on a microarchitecture with a cache line size of 32-bits. Since the array `A` contains elements of type `uint8_t`, four elements can fit in a single cache line. We can change our initial condition from `(x == y)` to `(x <= y && y < x+4)` to handle these cases.

More formally, we consider a program `P` containing an array A with some index of interest i. We would like to consider the cases $i \in J$ and $i \notin J$, where J is some range $[j, j+l)$ associated with a recent access to $A[j]$, and l is the cache line size. Furthermore, we would like to consider all $J \in \mathcal{J}$ where $\mathcal{J}$ is the set of all ranges that have been accessed thus far in `P`.



(a) Original control-flow graph of Figure 3.1

(b) Modified control-flow graph with explicit notion of the load/store dependency

Figure 3.2: Comparison of control-flow graphs with and without consideration for load/store dependencies

It is important to note that other factors may affect the cache as well, e.g., data prefetching may cause certain variables to be loaded into the cache without us noticing. However, for the purposes of data dependency, we are only concerned with cases where the input data results in variable timing. By considering these cases, we provide an approximation of the affects of these variable indexed arrays to GameTime in order to help it provide accurate timing estimates.

3.2 Implementation

For both types of data dependency, we utilize a preprocessor based approach that adds additional cases to the original C code so that GameTime may reason about the data dependency. In this way, we require no changes to the core algorithm.

Data Dependent Instructions

We handle data dependent instructions by replacing the given operation (i.e. a divide) with an appropriate C case statement. Depending on the complexity of the cases, we may choose to use an approximation.

As discussed in Section 3.1, we often do not have the explicit cases for a given operation. Thus we must learn this function and translate it to conditionals in C code. In some instances, it is possible to infer the cases by analyzing reference manuals and inspecting various test cases. However, this can be too difficult or time consuming, so we would like a more automated approach. To this end, we utilize decision tree learning, as the output closely matches if/then/else structure in C. Although the learned model will typically not match the true cases precisely, it provides a usable approximation.

Decision Tree Learning

Decision tree learning (DTL) creates a model that predicts some target variable given some set of input variables [14]. Each node in the resultant decision tree represents a condition or test, e.g., is `x > 255`. The tree continues to branch on conditions until a leaf is encountered; this leaf represents the actual decision or target variable. One of the major advantages of DTL is that the resultant predictive model is easy to understand and interpret, which allows us to programmatically transform the decision tree into C code. However, DTL can easily overfit data, so we must carefully tweak certain parameters, such as the maximum depth of the tree or the minimum number of samples per leaf. There are also certain concepts that are difficult for decision trees to learn, as we will see in Chapter 5.

More formally, let $x_i \in \mathbb{R}^n, i = 1, \ldots, l$ be training samples associated with labels $y \in \{0, \ldots, K-1\}^l$ where K is the number of values x_i may take. DTL works by recursively partitioning this space such that samples in a given partition are grouped together. We consider candidate splits $\theta = (j, t_m)$ where j is a feature and t_m is a threshold at node m. Using this split, we partition Q, the (x, y) data at node m, into two subsets

$$Q_{left}(\theta) = (x, y) | x_j \leq t_m$$
$$Q_{right}(\theta) = Q \setminus Q_{left}$$

The impurity at m is computed as

$$G(Q, \theta) = \frac{n_{left}}{N_m} H(Q_{left}) + \frac{n_{right}}{N_m} H(Q_{right})$$

where $H(\cdot)$ is the impurity function, n_{left} and n_{right} are the number of samples in the left and right partition, respectively, and N_m is the total number of samples at m. In order to find the best split, we minimize the impurity

$$\theta^* = arg\, min_\theta G(Q, \theta)$$

We then recurse on the subsets $Q_{left}(\theta^*)$ and $Q_{right}(\theta^*)$ until any either $N_m = 1$ or one of the previously listed conditions is met, e.g., maximum depth.

Automatic Generation of Cases

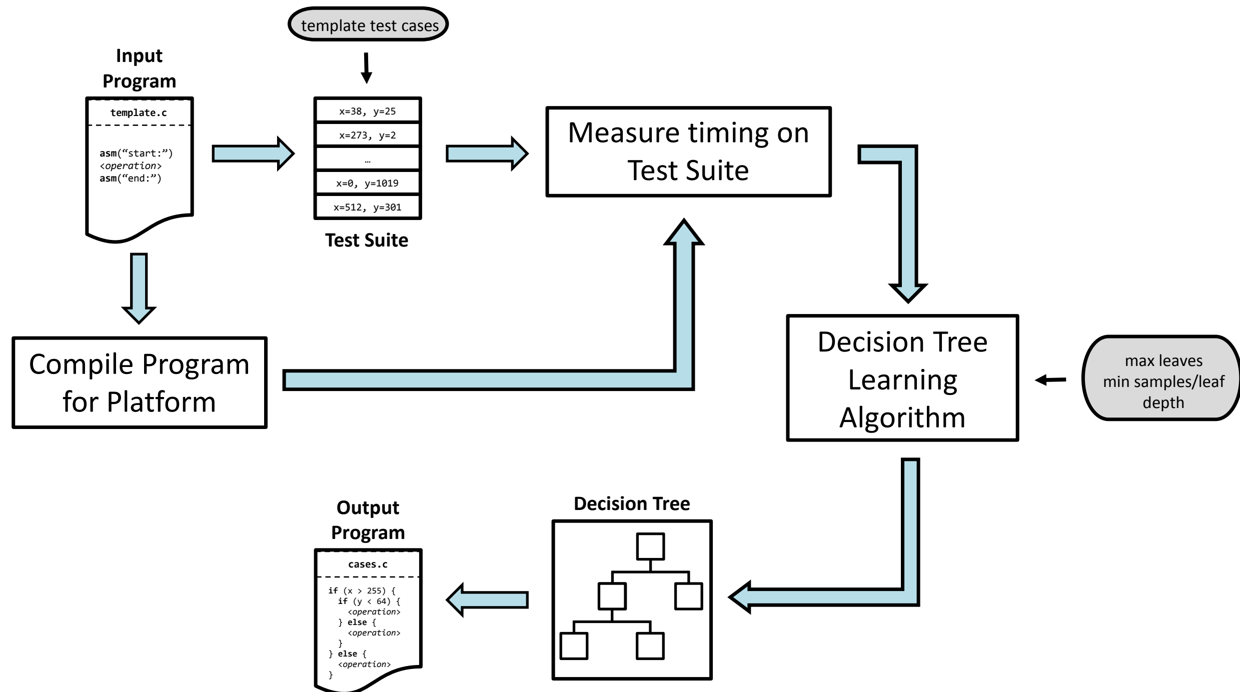


Figure 3.3: Overview of the automatic generation of cases for data-dependent operations

Figure 3.3 illustrates the method used to automatically generate a C representation of a given data dependent operation. The basic idea is to start with some operation we believe to be data-dependent (such as divide), compute the execution time across a wide range of inputs, and then use that as a training set to generate a function in C that represents the cases where the timing differs.

A template C file is provided with an annotated operation of interest (i.e. divide). The algorithm generates test cases by selecting arbitrary input arguments; alternatively, the test cases may be selected using a guided approach to balance the data if some a priori information about the instruction is available. The execution time is measured for these test cases and provided to a decision tree classifier. For our purposes, we use scikit-learn [14], an open-source machine learning library for Python, which utilizes an optimized version of the Classification and Regression Trees (CART) algorithm for decision tree learning. The depth, maximum number of leaves, minimum samples per leaf, and other parameters can be tweaked here as necessary. Finally the output decision tree is provided to a function which recurses on the nodes and generates the C code representation of the decision tree.

Load/Store Dependencies

To deal with load/store dependencies, we introduce dummy variables throughout the code to track the indexes into arrays. Based on knowledge of the cache, we can add cases based on whether a future array access is a cache hit or cache miss. Figure 3.4 shows the results of this source-to-source translation on the program from Figure 3.1.

```
1  uint8_t compare(uint8_t A[], uint8_t x, uint8_t y) {
2      uint8_t ret_val;
3      uint8_t ay;
4      uint8_t ax = A[x];
5
6      // save the index used to access A[]
7      uint8_t _x1 = x;
8      if (_x1 == y) {
9          // cache hit
10         ay = A[y];
11     } else {
12         // cache miss
13         ay = A[y];
14     }
15
16     if (ax == ay) {
17         ret_val = 1;
18     } else {
19         ret_val = 0;
20     }
21     return ret_val;
22 }
```

Figure 3.4: Annotated version of the toy example in Figure 3.1 that explicitly splits on recently used variable indices

As the number of array accesses increase, we must continue to increase the number of cases that we are required to check. If the number of cases becomes too large, we can start restricting which checks we perform. This approximation relies on the fact that our caches are finite in size and ultimately will replace some of the variables. If the cache replacement policy is known, we can intelligently choose which conditions to exclude; alternatively, we can arbitrarily exclude conditions as a rough approximation. Although removing checks decreases the information provided to GameTime, we still expect the estimation accuracy to improve over the original algorithm.

Chapter 4

Timing Repeatability

In this chapter, we present a formalization of the notion of timing repeatability along with two algorithms to measure the timing repeatability of a given platform for a given program.

4.1 Models of Timing Repeatability

First, consider a more informal definition. We say that a platform is *timing repeatable* for a program if the execution time of any code fragment from that program on that platform depends minimally on its *execution context*.

The execution context comprises two element:

1. The state of the platform before that code is executed, and
2. The state of the program (values of program counter and program variables) before that code is executed, which determines the program path executed.

Note that the program can influence the state of the platform, e.g., by modifying the contents of the cache.

Programmer's Perspective

Consider a programmer Paul who seeks to implement and execute a specific program P on a given hardware platform H. Paul cares only about the timing repeatability of H for the program P. In other words, if the timings of code fragments of P are independent of their execution context, then Paul would consider H to be perfectly timing repeatable.

We formalize the programmer's model of timing repeatability using the control-flow graph representation of the program, with all loops unrolled and functions inlined, so that the CFG is a DAG (as discussed in Section 2.1). Consider the CFG G of program P that has m edges and n nodes. Every edge of G is a basic block of P.

Most timing analysis tools use the basic block as the "atomic" unit of the program to predict program timing. For instance, given bounds on the execution time of a basic block,

timing analysis tools can infer bounds on the execution time of the entire program. Therefore, in our formalization, we also use the basic block as the atomic unit of a program; however, note that the theory below also extends to smaller units of a program.

Let σ denote the state of the platform just before P begins to execute. The program state at any program location in P is determined by the path $\boldsymbol{x}$ in G leading up to that location. Note that we encode all data-dependent timing into the path structure of the program as discussed in Chapter 3.

Let τ_i denote the execution time of basic block i of program P. As noted above, τ_i is determined by a combination of the platform state and the program state. Thus, τ_i is a function of σ and the path $\boldsymbol{x}$ leading up to i. We formalize this dependence by writing τ_i as

$$\tau_i(\sigma, \boldsymbol{x}) = w_i(\sigma) + \pi_i(\sigma, \boldsymbol{x}) \tag{4.1}$$

where w_i is the *nominal* execution time of basic block i that depends on the initial platform state but not on the program path that i lies on, and π_i is the path-dependent *perturbation* to the nominal execution time of basic block i. All three quantities τ_i, w_i, and π_i are functions whose co-domain is $\mathbb{R}$ (measurable in cycle counts, if each CPU cycle corresponds to a fixed amount of real time). We can stack the quantities for all basic blocks i to obtain vectors of functions τ, w, and π that take values in $\mathbb{R}^m$. From Chapter 2, recall that a program path is represented as a vector $\boldsymbol{x} \in \{0,1\}^m$. The execution time of a program path $\boldsymbol{x}$ in a platform state σ is thus the dot product

$$\tau \cdot \boldsymbol{x} \;=\; (w + \pi) \cdot \boldsymbol{x}$$

Given the above notation, one can then define the WCET and BCET as follows:

$$\text{WCET} \doteq \max_{\sigma} \; \max_{\boldsymbol{x}} \; \tau \cdot \boldsymbol{x} \tag{4.2}$$

$$\text{BCET} \doteq \min_{\sigma} \; \min_{\boldsymbol{x}} \; \tau \cdot \boldsymbol{x} \tag{4.3}$$

Our formalization of timing repeatability introduces two new metrics, denoted $\pi_{\max}$ and w_{diff}, and has the following four cases:

1. **Strict Timing Repeatability:** We say the platform is strictly timing repeatable for program P, if, for every basic block i of P, $w_i(\sigma) = w_i^*$ for all σ, and $\pi_i(\sigma, \boldsymbol{x}) = 0$.

   In other words, the execution time of a basic block is constant, independent of the starting platform state and the program path that it lies on. This is the ideal case.

2. **Bounded Path-Dependent Timing:** The platform is said to have bounded path-dependent timing, if

   - for every basic block i of P, $w_i(\sigma) = w_i^*$ for all σ, and $\pi_i(\sigma, \boldsymbol{x}) = \pi_i(\boldsymbol{x})$, and
   - for all paths $\boldsymbol{x}$, the total deviation from the nominal time $\sum_i \pi_i x_i$ is bounded in absolute value by a parameter $\pi_{\max}$.

More formally, $\forall \boldsymbol{x} \in \mathcal{P}$,

$$|\sum_i \pi_i x_i| \;=\; |\pi \cdot \boldsymbol{x}| \;\leq\; \pi_{\max}$$

When possible, we compute $\pi_{\max}$ as $\max_{\boldsymbol{x}\in\mathcal{P}} |\pi \cdot \boldsymbol{x}|$.

In other words, the execution time of a basic block depends only on the program path it lies on, and the amount of path-dependent variation in execution time is bounded. The constant $\pi_{\max}$ is a measure of how timing repeatable a given platform is. The smaller the value of $\pi_{\max}$, the greater the timing repeatability.

This case is relevant when the programmer has control over the starting platform state, and thus is only interested in the timing repeatability with respect to variation due to control flow.

3. **Bounded Platform-Dependent Timing:** The platform is said to have bounded platform-dependent timing, if

   - for every basic block i of P, $\pi_i(\sigma, \boldsymbol{x}) = 0$, and
   - for any program path $\boldsymbol{x}$, let $\sigma_{\min}(\boldsymbol{x})$ be the platform state corresponding to the minimum execution time of $\boldsymbol{x}$, and $\sigma_{\max}(\boldsymbol{x})$ be that corresponding to the maximum execution time of $\boldsymbol{x}$. Let $w_{\text{diff}}(\boldsymbol{x})$ be the difference between these execution times. Formally

$$\sigma_{\min}(\boldsymbol{x}) \doteq \text{argmin}_{\sigma} \tau \cdot \boldsymbol{x}$$
$$\sigma_{\max}(\boldsymbol{x}) \doteq \text{argmax}_{\sigma} \tau \cdot \boldsymbol{x}$$

   and

$$w_{\text{diff}}(\boldsymbol{x}) \doteq \max_{\sigma} \tau \cdot \boldsymbol{x} \;-\; \min_{\sigma} \tau \cdot \boldsymbol{x} \tag{4.4}$$

   We require that this difference be bounded by a parameter w^*_{diff}.

   We compute w^*_{diff} as the maximum such difference over all program paths, and employ it as a measure of timing repeatability.

$$w^*_{\text{diff}} = \max_{\boldsymbol{x}\in\mathcal{P}} \; w_{\text{diff}}(\boldsymbol{x}) \tag{4.5}$$

4. **Bounded Path-Dependent and Platform-Dependent Timing:** This is the general case, when program timing is dependent on both the platform state and the program path. In this case, both parameters $\pi_{\max}$ and w^*_{diff} are used as measures of timing repeatability.

   It is important to note that w^*_{diff} is not the same as the difference between the WCET and BCET. The difference WCET $-$ BCET combines together, in a single measure, the

variability in program timing due to platform state and program state. In contrast, $\pi_{\max}$ captures the variability due to program state alone (for a given platform state), while w^*_{diff} captures variability due to platform state alone.

From a programmer's perspective, it may be possible to reduce variability due to program state by modifying the program logic. On the other hand, variability due to platform state is typically not controllable by the programmer; it is, however, information that the platform architect can use in redesigning the platform so as to make it more timing-repeatable for a program of interest. Thus, computing $\pi_{\max}$ and w^*_{diff} can provide insight into the cause for variability in a program's timing on a given platform, separating the part that is controllable by the programmer from that which is not (the platform architect's concern). As we will see in Chapter 5, these metrics can be useful in evaluating the impact of a particular platform on a program's timing.

The following theorem captures an important property of $\pi_{\max}$ and w^*_{diff}, and its proof follows directly from the definition of strict timing repeatability.

Theorem 1 *If a platform is strictly timing repeatable, then* $\pi_{\max} = 0$ *and* $w^*_{\textit{diff}} = 0$*.*

In Section 4.2, we give an algorithmic technique to evaluate $\pi_{\max}$ and w^*_{diff} for a given combination of program P and hardware platform H.

Statistical Variants. In certain settings, it is meaningful to consider statistical variants of the above parameters. For instance, for certain programs, reasonable assumptions can be made over the distribution of starting program states (inputs to a program) that determines which paths are executed. Examples of such programs include cryptographic kernels, such as certain encryption algorithms, where the secret key determines the program path executed, and this key can be assumed to be uniformly distributed especially when it is combined with a pseudo-randomly generated "nonce".

In such settings, we can define variants of $\pi_{\max}$ and w^*_{diff} that are based on computing the mean or variance over the space of all paths or platform states, rather than the maximum. In our experimental evaluation in Section 5.2, we discuss an example where such statistical measures are relevant.

Architect's Perspective

Consider a computer architect Alice who seeks to implement a timing-repeatable hardware platform H. Alice must ensure that any instruction in any program, no matter when it executes, or what the input arguments, should take the same amount of time.

This notion of timing repeatability is much more stringent than the one from the programmer's perspective, since it requires the platform to be timing repeatable for *all* possible programs that would ever be executed on it.

Let the set of instructions defined by the platform be $\{I_1, I_2, \ldots, I_k\}$. Denote the time taken by instruction I_j by T_j. Then, we can write T_j as

$$T_j \;=\; T_j^{\text{nom}} + T_j^{\text{data}} + T_j^{\text{plat}} \tag{4.6}$$

where T_j^{nom} is the nominal instruction timing, T_j^{data} is the variation due to input arguments, and T_j^{plat} is the variation due to platform state.

For example, in certain ARM processors, a multiply instruction takes a variable amount of timing depending on the magnitude of the input arguments — with 8-bit arguments, it takes a single cycle; with 16-bit, two cycles, and so on. In this case, T_j^{nom} could be taken as the time for 8-bit arguments, any variation in timing is due to T_j^{data}, while $T_j^{\mathrm{plat}} = 0$.

In this paper, our focus is mainly on timing repeatability from the programmer's perspective. Hence, we do not focus in more depth on evaluating timing repeatability from the architect's perspective. However, we note that there is a connection between these two. Specifically, given bounds on T_j^{data} and T_j^{plat} for every instruction I_j, one can compute bounds on the variation of timing of any basic block (and hence program path) due to both platform state and program state. Such bounds are likely to be conservative unless fine-grained platform and program state information can be inferred at every program point. Also, in practice, such fine-grained timing information is often unavailable even from the processor reference manuals.

Therefore, in the next section, we focus on evaluating timing repeatability *without* detailed knowledge of the timing repeatability of the underlying instruction set architecture, using algorithmic techniques that operate at the program level.

4.2 Implementation

The key step in evaluating the timing repeatability of a hardware platform (from a programmer's perspective) is to compute the parameters $\pi_{\max}$ and w^*_{diff}. Based on this computation, we can also determine whether the platform is strictly timing repeatable.

Estimating $\pi_{\max}$

Our approach for estimating $\pi_{\max}$ is based on the notion of basis paths introduced in Chapter 2. We hold the initial platform state fixed for this estimation – thus the only variation in execution time is due to the program path.

The basic idea is as follows. Given the CFG G of the program P, we first compute a set of feasible basis paths of G as described in Chapter 2. Denote this set of b paths as $\{\boldsymbol{b}_1, \ldots, \boldsymbol{b}_b\}$, which is a subset of the set of all possible paths $\mathcal{P}$. We then use a combination of symbolic execution and SMT solving to compute test cases that drive execution down these paths. Next, the program P is executed on these test cases to obtain end-to-end timing measurements for each basis path. Denote these timing measurements by $t_1, t_2, \ldots, t_b$.

We then solve the following equation:

$$Bv = t \tag{4.7}$$

where B is the $b \times m$ matrix where the ith row is the vector $\boldsymbol{b}_i$ for the ith basis path, v is an $m \times 1$ vector of variables, and t is the $b \times 1$ vector whose ith element is the measurement t_i.

Note that since in general $b < m$, the matrix B is not square and this is an underconstrained system. We therefore solve for v as

$$v = B^{+}t$$

where B^{+} is the Moore-Penrose pseudo-inverse of B, obtained as $B^{+} = B^{\intercal}(BB^{\intercal})^{-1}$. It holds that $BB^{+} = I_b$.

We then interpret $v \in \mathbb{R}^m$ as the vector of weights on the edges of the CFG G. Using these edge weights as the nominal edge weights w, we evaluate the lengths of all program paths in G. Denote the length of path $\boldsymbol{x}$ obtained this way as $t(\boldsymbol{x})$.

We then measure the actual execution times of each path $\tau(\boldsymbol{x})$. $\pi_{\max}$ can then be computed as

$$\pi_{\max} = \max_{\boldsymbol{x} \in \mathcal{P}} |\tau(\boldsymbol{x}) - t(\boldsymbol{x})| \tag{4.8}$$

If we find that $\pi_{\max} > 0$, we can conclude that the platform is not strictly timing repeatable. To see this, note that if it were strictly timing repeatable, the actual time for any path $\boldsymbol{x}$, $\tau(\boldsymbol{x})$, is the dot product $\boldsymbol{x} \cdot w$ where w is the nominal path timing. This is expressible as $(\sum_i \alpha_i \boldsymbol{b}_i) \cdot w = \sum_i \alpha_i (\boldsymbol{b}_i \cdot w) = \sum_i \alpha_i t_i$, which can be written as $\alpha \cdot t$ where α is the $1 \times b$ vector of coefficients $\alpha_1, \ldots, \alpha_b$. Additionally, $\boldsymbol{x} \cdot v = (\alpha \cdot B) \cdot v = \alpha \cdot (B \cdot v) = \alpha \cdot (BB^{+}t) = \alpha \cdot t$. In other words, if the platform were strictly timing repeatable, $\forall \boldsymbol{x}$, the actual path timing $\tau(\boldsymbol{x}) = \boldsymbol{x} \cdot w$ equals the predicted timing $t(\boldsymbol{x}) = \boldsymbol{x} \cdot v$, implying the LHS of Equation 4.8 to be 0.

When comparing two platforms, one can compare not only the actual value of $\pi_{\max}$, but also the value of $\pi_{\max}$ normalized to the actual path timing, as follows:

$$\pi_{\max}^{\text{norm}} = \max_{\boldsymbol{x} \in \mathcal{P}} \frac{|\tau(\boldsymbol{x}) - t(\boldsymbol{x})|}{\tau(\boldsymbol{x})} \tag{4.9}$$

Practical Considerations. Note that the above computation of $\pi_{\max}$ requires an exhaustive enumeration of program paths. Such enumeration is feasible for small programs, but not for programs with a huge path space. In the latter case, here are some options:

(i) *Test Suite:* Every embedded task undergoes some amount of testing and thus, one must have an associated test suite. This test suite corresponds to a subset $\mathcal{P}_{\text{sub}}$ of the set of all paths $\mathcal{P}$. Thus, instead of computing $\pi_{\max}$ as a max over $\mathcal{P}$, we can compute the max over elements of $\mathcal{P}_{\text{sub}}$. Although the result is only a lower bound on the true $\pi_{\max}$, but assuming that the test suite was derived systematically for coverage criteria or for validating other (non-timing) correctness properties, it seems reasonable to use the resulting $\pi_{\max}$ estimate as a measure of timing repeatability.

   Similarly, if information about the distribution of program inputs (starting program states) is available, one can sample from this distribution to construct the subset of paths $\mathcal{P}_{\text{sub}}$.

(ii) *Program Fragments:* One can perform the analysis only on a fragment of the program, e.g., on a single function in the program for which all paths can be easily enumerated. Such analysis can be useful when the programmer has some prior insight into the location of timing variability in the program, and seeks to perform a more detailed analysis by computing $\pi_{\max}$ for that portion of the program.

(iii) *Over-Approximation:* Use symbolic approximation techniques, e.g., based on abstract interpretation, to find an upper bound $\pi_{\max}$. This approach requires having a good abstract model of the platform, and is only useful if the derived upper bounds are fairly tight and preserve the ordering of the true values of $\pi_{\max}$. For the platforms considered in this paper, such models and tools were not available; however, this appears a promising direction to scale up the analysis further, and we leave investigation of this option to future work.

Note that Options (i) and (ii) can in many cases determine that a platform is *not* strictly timing repeatable. In our experimental evaluation, we use small programs to evaluate different platforms for which we can perform exhaustive path enumeration. For other programs, we fall back to Option (i) where we use a test suite.

The value of $\pi_{\max}$ computed above, when non-zero, can be sensitive to the choice of basis. Therefore, we repeat the above experiment for K different choices of bases, for heuristically selected K. Each choice of basis is generated by using a new random order of rows of matrix B on Line 1 of Algorithm 1. We then take the minimum over all values of $\pi_{\max}$ obtained (for all K bases), and use that minimum as our final estimate $\pi_{\max}$. This is sound since the choice of basis that yields the smallest $\pi_{\max}$ yields a solution vector v that, when interpreted as the vector w of nominal timing values, yields $\pi_{\max}$ as the bound on the maximum perturbation of timing of any path.

Measuring $w^*_{\mathbf{diff}}$

To estimate w^*_{diff}, we simply evaluate Equations 4.4 and 4.5 presented in the preceding section. In other words, for each program path, we enumerate the possible different starting platform states, finding the biggest spread between path timings.

Since the number of starting platform states can also be very large, we can run into similar practical considerations as discussed in the preceding section. Our approach, in such cases, is similar to Option (i) above: we sample a subset of all possible starting platform states. We empirically find that this approach, although not exhaustive, can still provide valuable comparisons. For control software tasks that run inside an infinite sense-compute-actuate loop, one can sample various platform states at the start of different loop iterations to gain a realistic picture of the various possible starting platform states.

Chapter 5

Experimental Results

This chapter presents experiments across a range of benchmarks on a few different platforms. We first discuss the details of these platforms and what relevance they have to data dependency and timing repeatability. Next we discuss the structure and purpose of each benchmark that we utilize. Finally we analyze the results of for data dependency experiments along with an evaluation of our algorithms for finding $\pi_{\max}$ and w^*_{diff}. In the experiments discussed in this chapter, we utilize three primary platforms–ARM Cortex-M3, ARM Cortex-M4, and PTARM.

The ARM Cortex-M3 and Cortex-M4 platforms are standard 32-bit embedded processors. For the ARM Cortex-M3 platform, we use the STM32L1 Discovery board and for the ARM Cortex-M4 platform, we use the STM32F4 Discovery board. Using GDB, we can load programs to the board. In addition, we can obtain accurate cycle counts with GDB by looking at the cycle count (DWT_CYCCNT) register before and after the function of interest. We use a pre-built GNU toolchain targeted at embedded ARM processors that is maintained by ARM [6].

PTARM is a PRET machine developed by Liu et. al. [12] that is based on a subset of the ARMv4 ISA; we utilize both the FPGA implementation and the simulator for timing repeatability experiments. We load PTARM onto an Atlys Spartan 6 FPGA development kit. Code is loaded via USB using the built-in bootloader. When a program finishes execution, PTARM sends the total cycle count along with other metrics over the USB interface.

We also utilize the Simit-ARM simulator, a cycle-accurate simulator for the StrongARM-1100 processor. It is based on the Operation State Machine formalism presented in [16]. For compilation, we use the compilers that come with the distribution of the simulator.

Data dependency experiments are run using either the Cortex-M3 or Cortex-M4, while $\pi_{\max}$ evaluation compares PTARM with the Cortex-M4. For the w^*_{diff} measurements, we use a simulator for PTARM along with the Simit-ARM simulator. Since it can be difficult to control the modification of platform state on real hardware, we opted to use simulators for these experiments.

5.1 Platform Details

The various features of PTARM are discussed in [11]. PTARM utilizes a six-stage thread-interleaved pipeline where each stage executes a different thread. The threads are scheduled round-robin so each thread occupies only one pipeline stage at any given time. The pipeline design avoids the need for data hazard detection, bypassing, and speculative execution. The memory hierarchy consists of off-chip main memory with fast on-chip scratchpad memories; these scratchpad memories are used to avoid the unpredictability of hardware caches. A memory wheel is employed to handle accesses to the off-chip main memory. Each thread is given a window for memory accesses; if a thread misses its window, it must wait until the start of the next window. Although this may cause a thread to block, it does not affect any other threads and the window behavior is predictable.

The ARM Cortex-M3 platform [3] features a three-stage pipeline and utilizes three bus interfaces (ICode, DCode, and System). ICode and DCode are used for instruction fetches and data accesses, respectively, from Code memory space. The System interface is used for instruction fetches and data accesses to any other memory region. On this platform, divides take 2-12 cycles and multiplies take 1 or 3-5 cycles depending on if it is a "long" multiply or not. In Section 5.3, we look at both of these data-dependent instructions in detail.

The ARM Cortex-M4 platform [4] is very similar to the ARM Cortex-M3. The primary difference between the two is the addition of DSP extensions along with an optional FPU. In addition, all multiplies on the Cortex-M4 take a single cycle; however, divides still take 2-12 cycles. On the STM32F4 Discovery board, flash memory requires 5 wait states at the target clock rate (168MHz), while SRAM requires 0 wait states. In order to mitigate the inherit speed limitations of flash, ST incorporates the ART accelerator, a proprietary cache that can store 64 128-bit slices (4-8 instructions per slice) [23]. By default, flash uses Code memory space (instruction on ICode bus and data on DCode bus), while SRAM uses a different memory region (instruction and data on System bus). We utilize this cache for our analysis of load/store dependencies in Section 5.3.

5.2 Benchmarks

This section summarizes the various benchmarks used in the following experiments. Table 5.1 lists the benchmarks along with some relevant metrics.

The `divide` and `multiply` benchmarks correspond to programs that solely perform one operation (divide or multiply). Each variant illustrates a different functional representation of the operation of interest. `divide_exact` utilizes the exact divide function derived via inspection, while `divide_dtl_full` and `divide_dtl_random` correspond to functions derived via the DTL approach (see Section 3.2 for details). Note that `divide_dtl_full` is also an exact representation of the divide operation, as the dataset used here fully characterizes the exact cases found via inspection, and the decision tree had no restrictions. In most cases, this would result in overfitting the data, but since our dataset was precise and minimal,

Benchmark	*LoC*	*CFG Characteristics*			*b*
		n	*m*	*Total*	
divide_exact	47	76	86	12	10
divide_dtl_full	1485	2230	2599	372	372
divide_dtl_random	63	94	107	15	15
multiply_dtl	25	66	73	17	6
modexp (4 bits)	44	28	31	16	5
control	80	154	173	4096	9
compare	53	58	64	42	8
sbox	72	94	105	315	19
stabilisation	433	64	72	144	10
cctask	178	102	118	257	18
irobot	158	234	268	234	36
modexp (32 bits)	221	140	171	4294967296	33

Table 5.1: Characteristics of benchmarks where LOC is lines of code, n is the number of nodes, m is the number of edges, *Total* is the total number of feasible paths in the CFG, and b is the number of basis paths

this produced a function with simple $x < c$ conditionals that captured all cases for divide. Therefore, we are able to compare two different implementations of the exact same function; `divide_exact` has much fewer cases that are more complex, while `divide_dtl_full` has many more cases that are all extremely simple.

The remaining programs listed generally have variants that include the data-dependent extensions, but we only list the metrics for the base program. `modexp` performs the modular exponentiation mentioned in Section 2.2. This benchmark is particularly nice as we can easily tweak the number of `exponent_bit` in order to increase the complexity of the problem. In addition, `modexp` features two divides (via the mod operator) and two multiplies per `exponent_bit`. The `compare` benchmark is a modified version of the program listed in Section 3.1, which serves as a simple benchmark for load/store dependencies; `sbox` is also used to benchmark this type of data-dependency. The remaining benchmarks are different types of state machines and control tasks that could be found in safety-critical real-time applications.

5.3 Data Dependency Results

Data-Dependent Instructions

Using the methods discussed in Section 3.2, we evaluate two operations on the STM32L1 Discovery board–divide and multiply. We first look at the instructions in detail in order

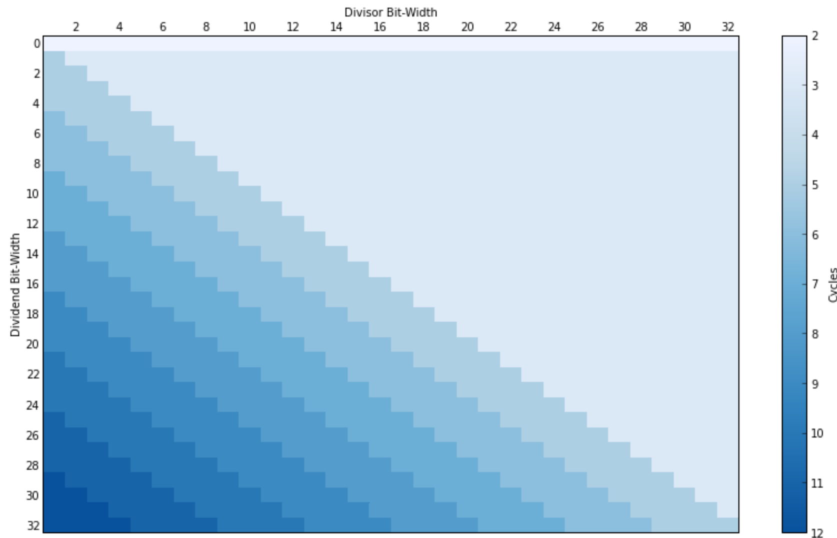


Figure 5.1: Exact representation of divide cases for ARM Cortex-M3

to derive suitable representations of them. Next, we use these representations in various benchmarks in order to see the estimation accuracy improvements.

Instruction Learning with Decision Trees

In order to determine the cases for the two instructions, we begin with the ARM Cortex-M3 technical reference manual [3], which states divide takes 2-12 cycles while multiply takes 3-5 cycles. Although the exact cases are not provided, the document does provide some insights into each instruction via a couple footnotes. For multiply, the instructions utilize early termination based on the size of the input arguments, while divide instructions use early termination based on the number of leading zeros and ones in the input arguments.

Given this information, we created a test suite for divide geared towards varying the number of leading ones and zeros. Therefore we sampled a large dataset based on the bit-width of the input arguments (note that the bit-width of a given number x is $\lceil \log_2(x+1) \rceil$). After some analysis, we recognized that the number of cycles depended on the difference between the bit-width of the divisor and the dividend. For instance, $a = 39,257$ requires 16 bits to represent and $b = 267$ requires 9 bits; thus, the bit-width difference is $16 - 9 = 7$. Similarly, $a = 2,147,483,648$ requires 31 bits and $b = 9,393,195$ requires 24 bits, which also has a bit-width difference of 7. Therefore, these two divisions take the same number of cycles. Figure 5.1 illustrates the number of cycle counts for all possible bit-widths. Note that $a = 0$ is a special case that always takes 2 cycles to complete, regardless of the value of the divisor. In addition, we see that the cycle count changes for every positive bit-width difference of 4

and that the cycle count is always 3 whenever the bit-width difference is negative.

While we were able to figure out the exact cases for divide in this situation, we would ideally like to avoid such a manual process. In light of this, we also analyzed divide using the decision tree learning approach. Figure 5.2 shows four different variants using DTL with varying parameters. For these experiments, we utilized a single dataset of $10,000$ labeled (a, b) pairs. As the maximum depth increases and the minimum samples per leaf increases, we see that the decision tree gets closer and closer to the exact representation of divide; however, this also results in a larger number of nodes in the tree, which means the resultant C code has more cases. Although the bit-width difference is not extremely amenable to the simple $x < c$ constraints used by DTL, we still get reasonable and usable approximations that we can use with GameTime.

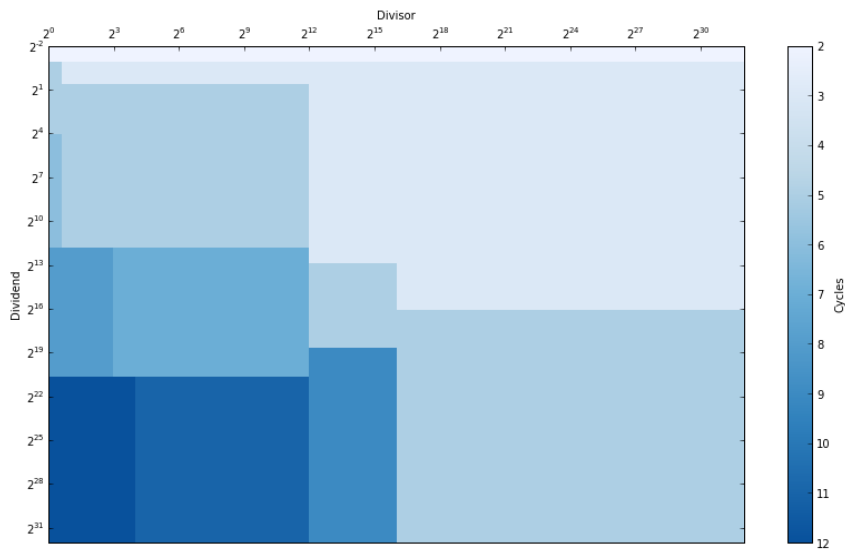
For multiply, we again focused on evenly sampling from the various powers of 2. We found that the timing depended on whether each input was greater than or less than 2^{16}. However, after some inspection, we found that the timing was one cycle lower when an argument was an exact power of 2. In lieu of this, we chose to utilize the simpler representation that captures the majority of cases. In addition, since the cycle counts are lower for that special case, we never underestimate the timing of a given multiply. Figure 5.3(a) shows the decision tree we use for multiplies and Figure 5.3(b) shows the resultant C code.

Evaluation of Learned Instructions

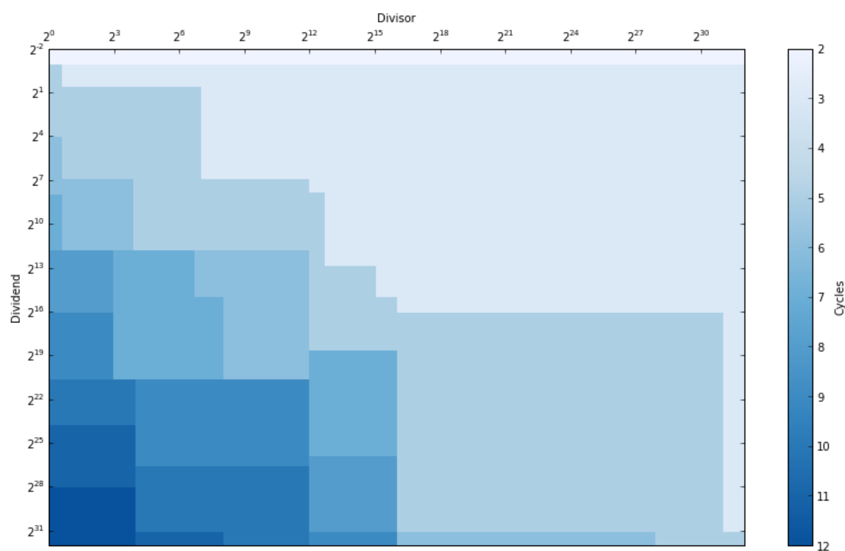
As discussed in Section 5.2, we evaluate the various representations of multiply and divide using a simple program that only uses the operation once. For the normal case where the operation is not replaced by anything, we have a single-path program according to GameTime; however, when we introduce a functional representation of the operation with various cases, the program becomes multi-path and GameTime can generate basis paths.

Although `divide_exact` looks to be the best option for our division replacement since it has the lowest number of basis paths and the best accuracy of representation (see Table 5.1, it unfortunately does not scale well. This is primarily due to the complexity of the conditionals used for the different cycle counts. In particular, `divide_exact` calculates the bit-width of each input argument using a bit twiddling hack from [2]. The input arguments are rounded up to the nearest power of 2, which represents $2^{\texttt{bitwidth}}$; then the first argument is divided by the second resulting in $2^{\texttt{bitdifference}}$. Using this information, we generate cases for each bit-width difference accordingly. As benchmarks include more and more divide statements that depend on the results of previous divide statements, it becomes increasingly difficult to find satisfying assignments to drive the program down a particular path. Conversely, the DTL approximation of division scales much better since they utilize only simple $x < c$ conditionals. However, `divide_dtl_full` contains too many conditionals per divide to be useful even though they are all simple; the basis calculation for this benchmark took over 4000 seconds, while all others took less than 10 seconds.

We analyze the effects of including `divide_dtl_random` with the `modexp` benchmark (note for this experiment we set `exponent_bits` $= 8$). Using GameTime, we compute the basis

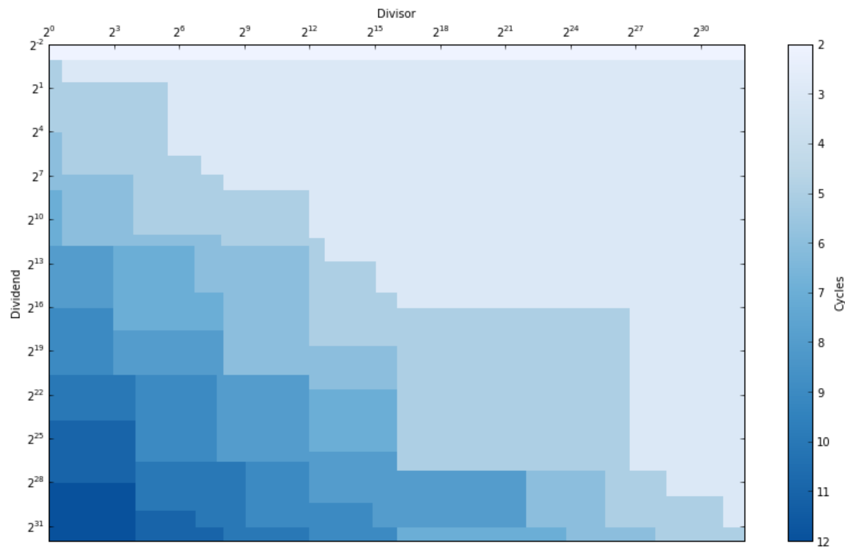


(a) `max_depth` = 5, `min_samples_leaf` = 50 (`divide_dtl_random`)

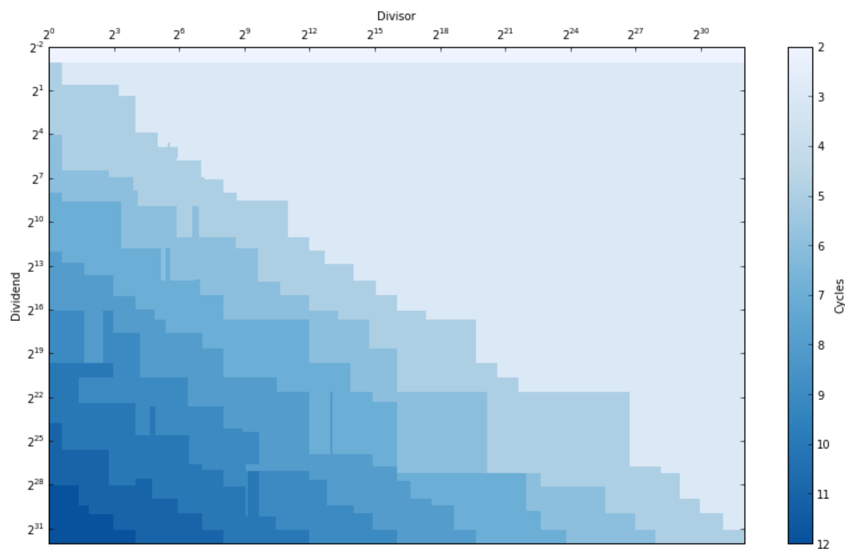


(b) `max_depth` = 7, `min_samples_leaf` = 25

Figure 5.2: Various approximations of divide using DTL ($\log_2$ scale)

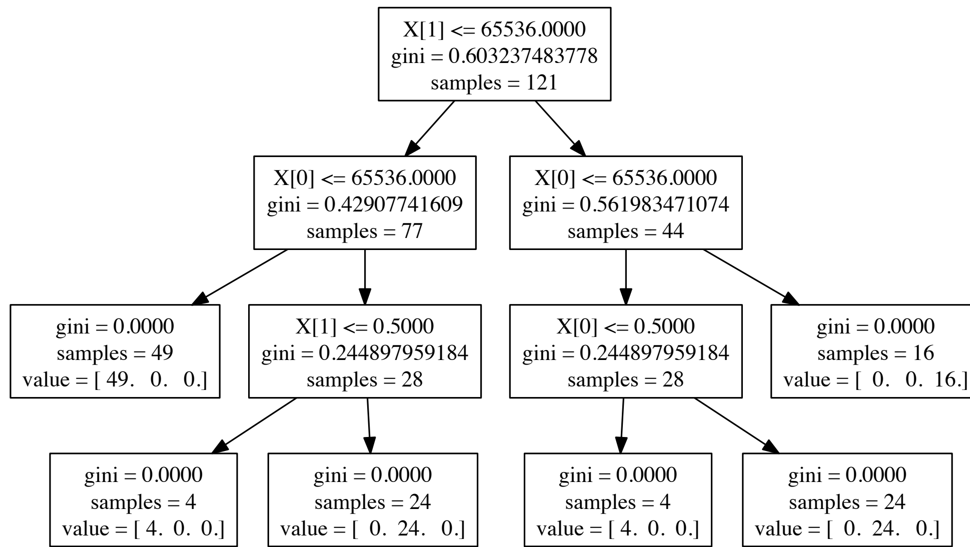


(c) `max_depth` $= \infty$, `min_samples_leaf` $= 20$



(d) `max_depth` $= \infty$, `min_samples_leaf` $= 1$

Figure 5.2: Various approximations of divide using DTL ($\log_2$ scale)



(a) Decision tree for multiply

```
1  uint32_t multiply(uint32_t a, uint32_t b) {
2    uint32_t x;
3    if (a == 0 || b == 0) {
4      x = a * b;
5    } else if (a <= 65536 && b <= 65536) {
6      x = a * b;
7    } else if (a <= 65536 && b > 65536) {
8      x = a * b;
9    } else if (a > 65536 && b <= 65536) {
10     x = a * b;
11   } else {
12     x = a * b;
13   }
14   return x;
15 }
```

(b) C code for multiply

Figure 5.3: Multiply results using DTL approach

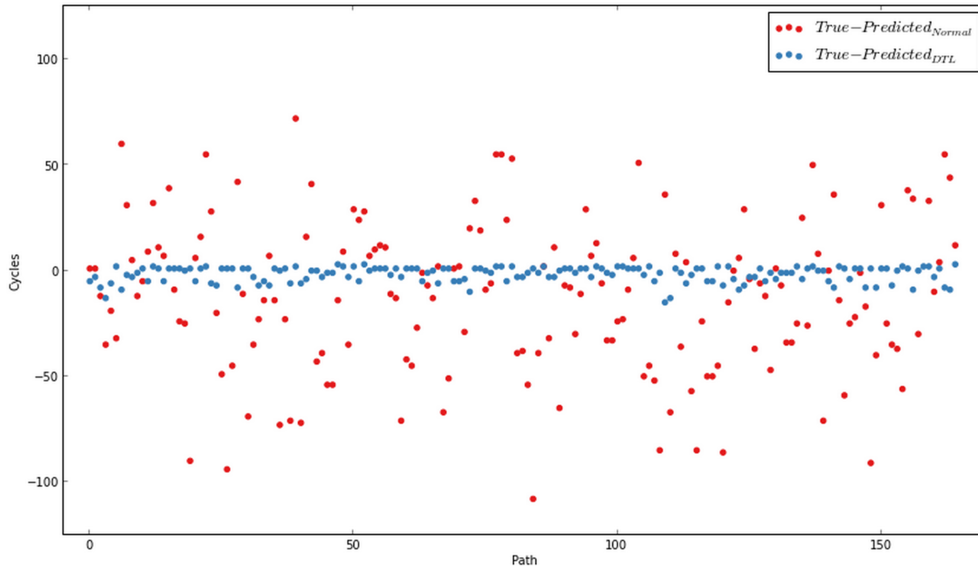


Figure 5.4: Modexp prediction accuracy with and without `divide_dtl_random`; the red dots represents the absolute error between the true measurements and normal predictions, while the blue dots represents the absolute error between the true measurements and predictions using `divide_dtl_random`

paths for both the normal and DTL variants of `modexp` and generate many random path predictions. Figure 5.4 illustrates the prediction accuracy of each variant for over 150 paths. We can see clearly that by introducing the data-dependent operation, we obtain much better prediction accuracy. Note that we look at the prediction minus the true cycle count for this plot. Thus, it is also interesting to note that the DTL variant tends to provide an upper bound whenever it mispredicts, while the variant without data-dependency is an overestimation as often as an underestimation.

Single path programs are also a very interesting application area for data-dependent instructions. These programs typically consist of many math operations without any flow control, such as a feedback control algorithm. Each step consists of computing the control action given the inputs to the system. We analyze some example controllers that only utilize addition, multiplication, and bit shifts; we use `multiply_dtl` for the multiplication operation. For the `control` benchmark listed in Table 5.1, we obtain 81 paths with a maximum prediction error of 1 cycle, an average error of 0.086 cycles, and an absolute percentage error of 0.813%. If we had instead looked only at the original single path program, we would only have a single measurement to represent all of these paths; using a random sample as our estimate, we receive an average error of 3.25 cycles and an absolute percentage error of 8.990%, which is significantly larger than the data-dependent case.

Load/Store Dependencies

For load/store dependencies, we add auxiliary variables to represent array indices that have been previously accessed, as described in Section 3.2. We analyze a benchmark very similar to the `compare` example listed in Chapter 3; in this case, we look at a three-way compare that takes three indices and returns the largest indexed element of an array. Compared to the original program predictions by GameTime where all predictions were fairly close to each other, we found the resultant predictions with consideration for load/store dependencies had a bimodal distribution of cycles. This is likely due to the fact that we only had a single array in this program, and we were looking at hits and misses on this cache; thus the paths with hits form one set of cycles and the paths with misses form the other set.

We also looked at single path programs for load/store dependencies. In particular, `sbox` is a simple example that illustrates substitution boxes, a commonly used component in cryptography. The basic idea is to take some number of input bits, n, and transform it into some number of output bits, m. S-boxes are typically represented as lookup tables of size 2^n with m-bit entries. Thus this substitution from input to output is just an array lookup; furthermore, this substitution often occurs multiple times in succession. For this benchmark, we consider the case with four consecutive array accesses. Using our techniques for load/store dependencies, we obtain an average error of -1.45 cycles (an absolute percentage error of 3.146%). For the original single path `sbox`, we can only use a single value for all predictions. By sweeping over a variety of possible measured values and computing the error with respect to all possible paths, we obtain average errors around -8.87 and 11.13 cycles (absolute percentage errors of 6.015% and 7.909% respectively). Although the gap between the average error is not very extreme in this case, it should become wider as we increase the number of array accesses, as we would have a wider range of cache hits versus cache misses. By considering these different cases explicitly, we are able to track the variations in timing in much greater detail.

Unfortunately we were unable to look at larger benchmarks for this technique, as the process is currently manual for adding the auxiliary variables. In the future, we hope to integrate this into the preprocessor so that the index tracking variables and conditionals can be added automatically.

5.4 Timing Repeatability Results

Evaluating $\pi_{\max}$

We compute $\pi_{\max}$ and $\pi_{\max}^{\text{norm}}$ by exhaustively enumerating the actual times $\tau(\boldsymbol{x})$ and the predicted times $t(\boldsymbol{x})$ for all paths in the program and then solving Equation 4.8 and Equation 4.9 respectively. Table 5.2 lists the results of these experiments. For all of the benchmarks, the time to compute the basis is well under a minute.

$\pi_{\max}^{\text{norm}}$ is used for comparisons between the two platforms because the raw timing values provide little insight as the Cortex-M4 is significantly faster than PTARM. For `cctask` and

Table 5.2: $\pi_{\max}^{\text{norm}}$ results for the PTARM and ARM Cortex-M4 platforms (the $\pi_{\max}^{\text{norm}}$ results are scaled by 1000 for readability)

Benchmark	*PTARM*	*Cortex-M4 Cache*	*Cortex-M4 NoCache*	*Cortex-M4 SRAM*
`modexp (4 bits)`	14.00	32.25	17.24	38.96
`stabilisation`	190.86	449.44	780.24	516.19
`cctask`	191.74	649.21	746.67	846.15
`irobot`	13.20	0.00	0.00	10.98
`modexp (32 bits)`	-	9.50	1.39	6.87

`stabilisation`, we see that PTARM is indeed more timing repeatable, as we would expect. However, our original results for `modexp` suggested that $\pi_{\max}^{\text{norm}} = 86.75$, which implies the Cortex-M4 is more repeatable for that program. The reason is not immediately obvious, but it becomes clear after analyzing the resultant assembly. For PTARM, the mod operator in `modexp` is translated to an assembly function full of conditional branches. On the other hand, the Cortex-M4 has hardware instructions for division, so it merely performs a division followed by a multiply and subtract to compute the mod operator. Thus, it appears more repeatable than PTARM since the hardware instructions have a much lower variance in cycle count compared to the assembly function. After performing our data-dependent instruction techniques on the mod operation for PTARM, we obtained $\pi_{\max}^{\text{norm}} = 14.00$ as listed in the table. Thus, we see that PTARM is, in fact, more repeatable than the Cortex-M4 for `modexp`.

It is quite interesting to compare the various memory configurations for the Cortex-M4. For `stabilisation`, the cache configuration actually performs much better than the configuration without the cache and the SRAM configuration. Although caches are typically thought to cause non-repeatable timing, it is reasonable for it to perform better than the other configurations for certain programs; if nearly all of the memory accesses and instruction fetches are hits, then the flash memory acts as 0 wait state memory. Furthermore, the configuration without cache suffers from wait state latency, which can cause non-repeatable timing. SRAM performing worse than any flash configuration is also somewhat unintuitive; however, the default SRAM configuration for this chip utilizes the System bus for both instruction fetches and memory accesses, while the default flash configuration utilizes the ICode and DCode buses. Thus flash memory is at a definite advantage since it cannot suffer from bus contention. Based on the benchmark results, this bus contention causes SRAM to be slightly less repeatable than the flash configurations most of the time.

For `irobot`, the two Cortex-M4 flash configurations actually show perfect repeatability. This is somewhat surprising; however it is important to note that $\pi_{\max}$ for PTARM and Cortex-M4 SRAM are also extremely small values (keep in mind we are scaling the $\pi_{\max}^{\text{norm}}$ by 1000; thus values around 10 are likely single cycle inaccuracies on some paths). The basic blocks for `irobot` are quite simple, primarily setting flags with very little arithmetic; thus, we expect $\pi_{\max}$ to be small. For the flash configurations, there are two separate buses

allowing instruction fetches and memory accesses to occur without contention. There is an issue of bus contention for the Cortex-M4 SRAM, and the memory wheel for PTARM can cause access latencies, which could explain the non-zero, but small, values for $\pi_{\max}$ on both of these platforms.

The final experiment, the 32-bit `modexp`, illustrates the use of the statistical variants discussed in Section 4.1. We only evaluated this benchmark for the various Cortex-M4 memory configurations. The number of cases is too large to exhaustively enumerate and measure in a reasonable amount of time, so we opt for a random sampling of 250 paths to evaluate $\pi_{\max}$. Since `modexp` is used in many security algorithms, such as RSA, uniformly sampling inputs can be a valid approach depending on the application. The results are not as large as $\pi_{\max}$ from the 4-bit `modexp` benchmark, which is to be expected since we are looking at averages over all paths rather than the max. However, this still provides us some information about the platform's repeatability that we can use to compare against other platforms. In this benchmark, the configuration without the cache performs more repeatably than the cache configuration. It is likely that the size of the program is now much larger than the cache, resulting in many more misses during instruction fetches and data accesses. SRAM performs in the middle of the two flash configurations again, likely due to System bus contention.

Evaluating w^*_{diff}

In this section, we evaluate timing repeatability with respect to varying initial platform states for the simple modular exponentiation example presented in Section 2.2.

The experiments are divided into two categories. In the first category, the modular exponentiation is performed after an array has been sorted, while in the second category, it is performed after two matrices have been multiplied. This "initialization" code is run so as to modify the platform state, such as the state of the cache and pipeline, before running the actual computation that needs to be timed.

Figure 5.5 shows the cycle counts for the experiments in the first category when run on the SimIt-ARM simulator. The first experiment merely runs the original modular exponentiation code on the 32 possible 5-bit exponents, with 2 as the base, modulo the prime number 1048583, which is the next prime number after 2^{20}. The second experiment also runs the modular exponentiation code, but the code is modified to include (unused) code to sort an array.

The next six experiments then use programs that initialize different kinds of arrays and may use the sorting code to sort these arrays before the modular exponentiation is performed. Through this initialization and sorting, these experiments thus provide different starting states. For example, experiment 3 initializes an array of 20 elements with the first 20 natural numbers in descending order, while experiment 4 initializes this array and then sorts this array, all before the modular exponentiation. Experiment 5 initializes a 20-element array with random numbers, while experiment 6 also sorts this array. In all of these cases, initialization is done through a for-loop. Experiments 7 and 8 sort a 20-element array that

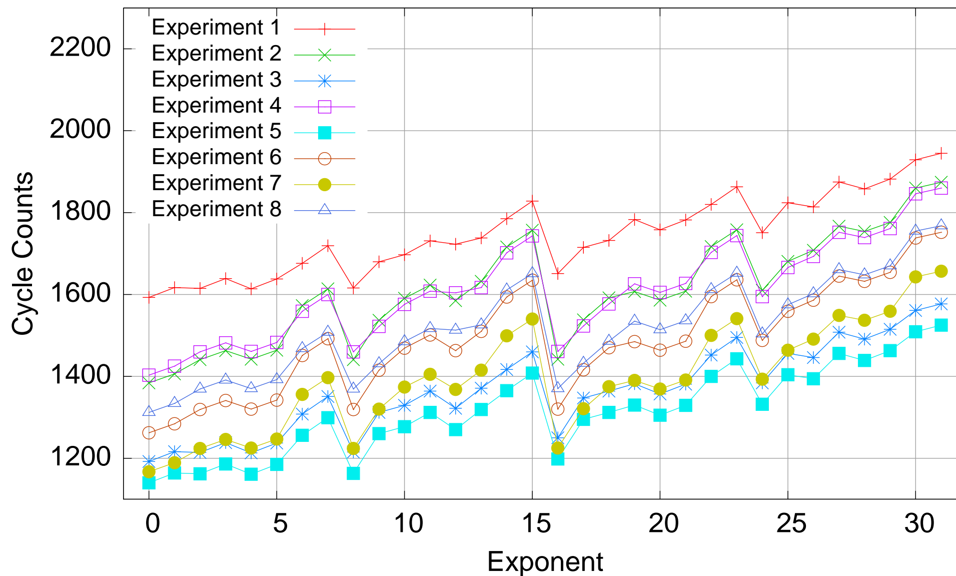


Figure 5.5: SimIt-ARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after an array has been sorted

has been statically initialized with, respectively, the first 20 natural numbers (in descending order) and random numbers. Note that the cycle counts reported are only for the modular exponentiation, and do not include the cycle counts used for sorting the array.

The experiments in the second category (see Figure 5.6) are similar, but use matrix multiplication instead of array sorting. The first experiment uses the original modular exponentiation code, and the second experiment uses this code, but modified to include (unused) code that performs matrix multiplication. The next six experiments use the unused matrix multiplication code to multiply matrices with dimensions 3×3, 4×4, 5×5, 10×10, 50×50, 100×100, and 150×150. A matrix with dimensions $n \times n$ contains numbers from 1 through n, arranged in increasing order, row-by-row, from left to right.

Together, the fifteen unique experiments provide fifteen varying initial states. In the case of the SimIT-ARM simulator, these fifteen varying initial platform states result in a value of 453 for w^*_{diff}.

Figure 5.7 and Figure 5.8 show the cycle counts for the experiments in the first and second category respectively, when run on a PTARM simulator. For this platform, the value of w^*_{diff} is 7, a much smaller figure. This result indicates that the PTARM platform appears to be more timing-repeatable with respect to the initial platform state than the StrongARM-1100. Because the PTARM pipeline design ensures that a thread only occupies a single pipeline stage at any given time, varying the initial state cannot cause any timing variation due to the pipeline. The primary cause of timing variation for PTARM is likely due to missed windows on the memory wheel; however, the StrongARM-1100 also has memory latency issues along with more susceptibility to the initial pipeline state.

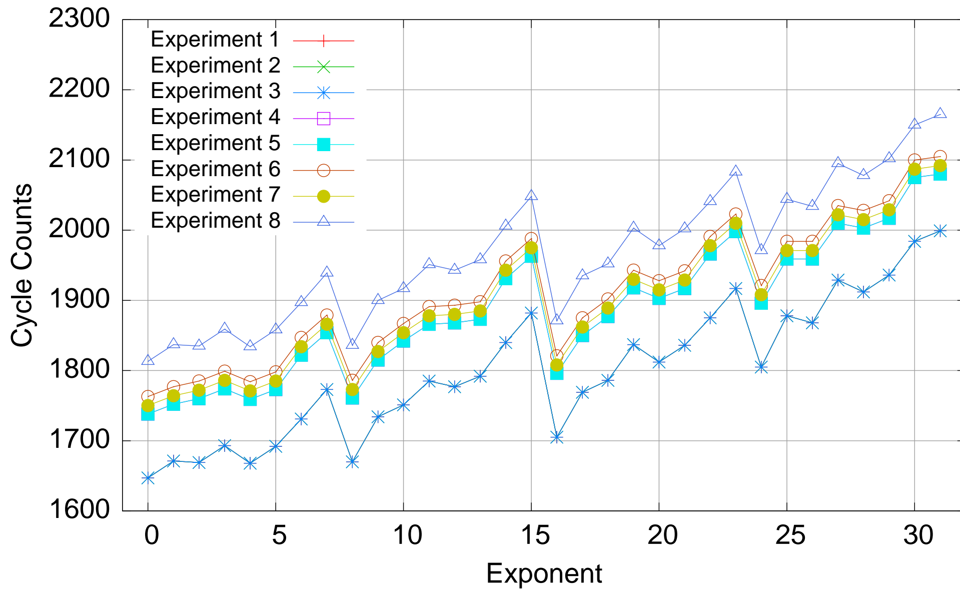


Figure 5.6: SimIt-ARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after two matrices have been multiplied

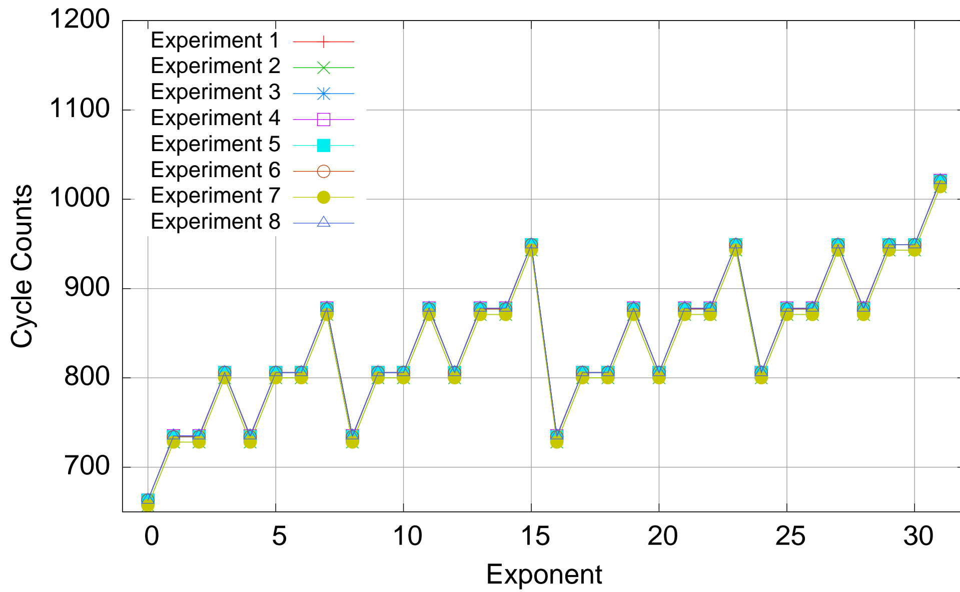


Figure 5.7: PTARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after an array has been sorted

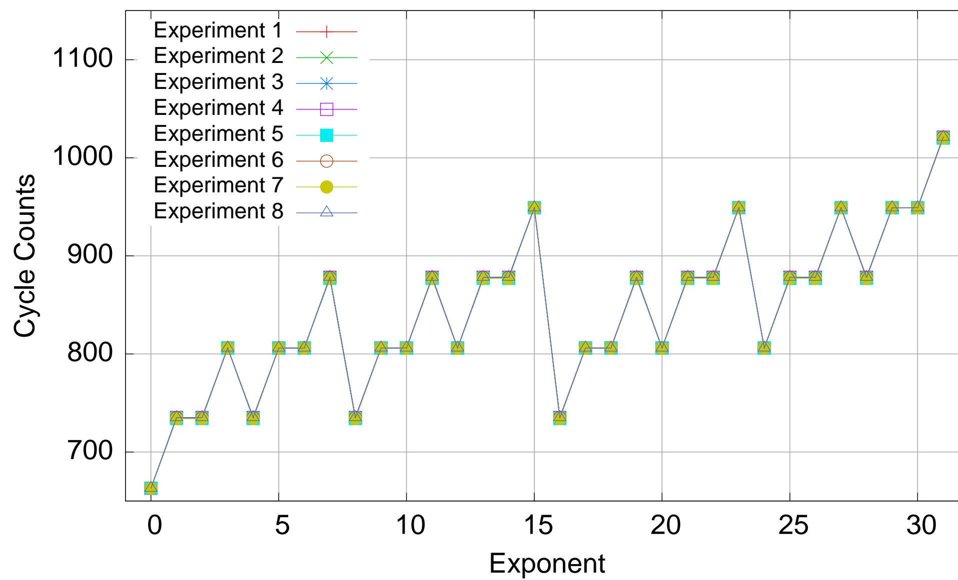


Figure 5.8: PTARM cycle counts for modular exponentiation with respect to different initial platform states, when performed after two matrices have been multiplied

Chapter 6

Conclusion and Future Work

In this thesis, we have explored two main topics and provided evaluations of each of them using real-world benchmarks and architectures.

For data-dependency, we presented models of instructions with variable timing and load-/store dependencies. We discussed multiple techniques for learning the cases for a data-dependent instruction, including a method that automatically generates test cases, measures timing on the platform of interest, constructs a decision tree, and ultimately produces a C code representation of the operation. We also presented a method of source-to-source translation in order to allow GameTime to reason about these situations. We then evaluated these extensions with a set of benchmarks that illustrate the improvements to prediction accuracy. The results suggest that our approximations of operations (using decision tree learning) greatly decrease the errors between our predictions and the true cycle counts.

We also provided a formalization of timing repeatability from the perspective of the programmer and the architect. With our focus on the programmer's perspective, we specified four types of timing repeatability, and we developed algorithms for measuring the two metrics of interest–$\pi_{\max}$ and w^*_{diff}. In our evaluation of these techniques, we compare PRET machines to standard ARM microprocessors using the data-dependent extensions from this thesis. While none of the platforms is perfectly timing-repeatable, the metrics help explain relative timing behavior across platforms. Our approach thus presents an objective method to compare two different platforms for timing repeatability. In general, we found that the PRET machines are more timing repeatable than ARM, which matches our original intuition.

Looking towards the future, our focus would be on evaluating these techniques on a broader range of benchmarks along with more industrial size benchmarks. Although our results illustrate the benefits of these techniques, they primarily serve as a proof-of-concept. In order to evaluate larger benchmarks for the data-dependent experiments, we need to automate more of the process. While learning operations is almost completely automated, replacing operations and adding auxiliary variables to the programs of interest is still very much a manual process. In addition, more work is needed to improve the scalability of the $\pi_{\max}$ and w^*_{diff} algorithms, potentially by performing a more symbolic search.

Bibliography

[1] S. Andalam, P. Roop, and A. Girault. "Predictable multithreading of embedded applications using PRET-C". In: *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference on*. July 2010, pp. 159–168. DOI: 10.1109/MEMCOD.2010.5558636.

[2] Sean Eron Anderson. *Bit Twiddling Hacks*. http://graphics.stanford.edu/ seander/bithacks.html.

[3] ARM. *ARM Cortex-M3 Processor Technical Reference Manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0337i/DDI0337I_cortexm3_r2p1_trm.pdf.

[4] ARM. *ARM Cortex-M4 Processor Technical Reference Manual*. http://infocenter.arm.com/help/topic/com.arm.doc.ddi0439d/DDI0439D_cortex_m4_processor_r0p1_trm.pdf.

[5] Stephen A. Edwards and Edward A. Lee. "The Case for the Precision Timed (PRET) Machine". In: *Design Automaton Conference (DAC)*. 2007, pp. 264–265.

[6] GCC ARM Embedded Maintainers. *GNU Tools for ARM Embedded Processors*. https://launchpad.net/gcc-arm-embedded.

[7] M.A. Kinsy, M. Pellauer, and S. Devadas. "Heracles: Fully Synthesizable Parameterized MIPS-Based Multicore System". In: *Field Programmable Logic and Applications (FPL), 2011 International Conference on*. Sept. 2011, pp. 356–362. DOI: 10.1109/FPL.2011.70.

[8] Raimund Kirner and Peter Puschner. "Obstacles in Worst-Case Execution Time Analysis". In: *ISORC*. 2008, pp. 333–339.

[9] Edward A. Lee. *Computing Foundations and Practice for Cyber-Physical Systems: A Preliminary Report*. Tech. rep. UCB/EECS-2007-72. University of California at Berkeley, May 2007.

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

[11] Ben Lickly et al. "Predictable Programming on a Precision Timed Architecture". In: *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems (CASES), Atlanta, Georgia*. Ed. by Erik R. Altman. ACM. Oct. 19, 2008, pp. 137–146. URL: http://chess.eecs.berkeley.edu/pubs/475.html.

[12] Isaac Liu et al. "A PRET Microarchitecture Implementation with Repeatable Timing and Competitive Performance". In: *Proceedings of International Conference on Computer Design (ICCD)*. Oct. 2012.

[13] Sharad Malik and Lintao Zhang. "Boolean Satisfiability: From Theoretical Hardness to Practical Success". In: *Communications of the ACM (CACM)* 52.8 (2009), pp. 76–82.

[14] F. Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

[15] Peter Puschner et al. "Compiling for Time Predictability". In: *Computer Safety, Reliability, and Security*. Ed. by Frank Ortmeier and Peter Daniel. Vol. 7613. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012, pp. 382–391. ISBN: 978-3-642-33674-4. DOI: `10.1007/978-3-642-33675-1_35`. URL: `http://dx.doi.org/10.1007/978-3-642-33675-1_35`.

[16] Wei Qin and S. Malik. "Flexible and formal modeling of microprocessors with application to retargetable simulation". In: *Design, Automation and Test in Europe Conference and Exhibition, 2003*. 2003, pp. 556–561. DOI: `10.1109/DATE.2003.1253667`.

[17] Reinhard Wilhelm et al. "The Determination of Worst-Case Execution Times—Overview of the Methods and Survey of Tools". In: *ACM Transactions on Embedded Computing Systems (TECS)* (2007).

[18] Martin Schoeberl. "Is Time Predictability Quantifiable?" In: *International Conference on Embedded Computer Systems (SAMOS 2012)*. IEEE, 2012.

[19] Martin Schoeberl et al. "Towards a Time-predictable Dual-Issue Microprocessor: The Patmos Approach". In: *Bringing Theory to Practice: Predictability and Performance in Embedded Systems*. Ed. by Philipp Lucas et al. Vol. 18. OpenAccess Series in Informatics (OASIcs). Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2011, pp. 11–21. ISBN: 978-3-939897-28-6. DOI: `http://dx.doi.org/10.4230/OASIcs.PPES.2011.11`. URL: `http://drops.dagstuhl.de/opus/volltexte/2011/3077`.

[20] Sanjit A. Seshia and Jonathan 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.

[21] Sanjit A. Seshia and Alexander Rakhlin. "Game-Theoretic Timing Analysis". In: *Proc. IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2008, pp. 575–582.

[22] Sanjit A. Seshia and Alexander Rakhlin. "Quantitative Analysis of Systems Using Game-Theoretic Learning". In: *ACM Transactions on Embedded Computing Systems (TECS)* (2012).

[23] ST. *STM32F407xx Datasheet*. `http://www.st.com/st-web-ui/static/active/en/resource/technical/document/datasheet/DM00037051.pdf`.

[24] Theo Ungerer et al. "Merasa: Multicore Execution of Hard Real-Time Applications Supporting Analyzability". In: *IEEE Micro* 30 (2010), pp. 66–75. ISSN: 0272-1732. DOI: `http://doi.ieeecomputersociety.org/10.1109/MM.2010.78`.

[25] Jack Whitham and Neil Audsley. "MCGREP–A Predictable Architecture for Embedded Real-Time Systems". In: *Real-Time Systems Symposium, 2006. RTSS '06. 27th IEEE International*. Dec. 2006, pp. 13–24. DOI: `10.1109/RTSS.2006.28`.

[26] Reinhard Wilhelm and Björn Wachter. "Abstract Interpretation with Applications to Timing Validation". In: *Proceedings of the 20th international conference on Computer Aided Verification*. CAV '08. Princeton, NJ, USA: Springer-Verlag, 2008, pp. 22–36. ISBN: 978-3-540-70543-7. DOI: `10.1007/978-3-540-70545-1_6`. URL: `http://dx.doi.org/10.1007/978-3-540-70545-1_6`.