Informatica — Universiteit van Amsterdam

Bachelor Informatica

University of Amsterdam

Deriving an Execution Time Distribution by Exhaustive Evaluation

Boudewijn Braams

June 8, 2016

Supervisor(s): Sebastian Altmeyer

Abstract

Safety-critical real-time embedded systems are subjected to stringent timing constraints, analysing their timing behaviour is therefore of great significance. Research regarding the timing behaviour of real-time systems has been primarily focussed on finding out what happens in the worst-case (i.e. finding the worst case execution time, or WCET). There is no established method or tool for deriving the execution time *distribution* of such a system. While a WCET estimate can verify that a system is able to meet deadlines, it does not contain any further information about how the system behaves *most* of the time. An execution time distribution does contain this information and can provide useful insights regarding the timing behaviour of a system. In this thesis, a measurement-based framework is presented that derives an execution time distributions by exhaustive evaluation of program input.

Contents

CHAPTER 1

Introduction

Since the dawn of information processing systems, there has been a trend towards miniaturization. Up until the 1980s these systems largely manifested themselves as mainframes, often occupying entire rooms. The 1990s saw the advent of the personal computer (PC), small enough to fit on a desk. One can however argue that we are heading towards a post-PC era, an era in which computers are no longer present as dedicated units, but rather manifest themselves as integrated systems in the objects we interact with every day. This is what defines an *embedded system*: an information processing system embedded into an enclosing product [19].

While *embedded systems* are nothing new, take for example the Apollo Guidance Computer used in sending the first humans to the Moon, their application domain has never been greater. Embedded systems are present in phones, televisions and printers, but also in more industrial applications such as cars, aircraft or even medical equipment. Modern instantiations of such products are practically guaranteed to have embedded information processing power.

In contrast to consumer electronics, industrial application of embedded systems are subjected to stringent reliability requirements. More often than not, these systems are entrusted with performing *safety-critical* tasks and should therefore have predictable timing behaviour. Take for example the anti-lock braking system (ABS) of a modern car: it automatically adjusts braking pressure on a wheel to prevent it from locking up, thereby avoiding uncontrolled skidding. In order to achieve this, the system needs to measure the rotational speed of all the wheels, perform computations with this data and subsequently adjust the braking pressure if needed. It is evident that such a system, in addition to functioning correctly, requires predictable timing behaviour. Failing to deliver the results of the computations within a certain time frame could possibly result in a catastrophe. What characterizes such embedded systems is that they must adhere to strict timing constraints; the term for such a system is a *real-time* embedded system.

An information processing system is considered to be *real-time* when the correctness of the system behaviour not only depends on the logical result of the computations, but also on the physical time at which these computations are completed [21]. It is evident that thorough analysis of the timing behaviour of safety-critical real-time embedded systems is of utmost significance.

Research on the timing behaviour of real-time embedded systems has been primarily focussed on developing methods and tools for finding out what happens in the worst-case (i.e. finding the worst-case execution time or WCET, see Figure 1.1). The goal of this thesis is to develop an analysis that is able to derive the actual *distribution* of execution times (the curve in Figure 1.1).

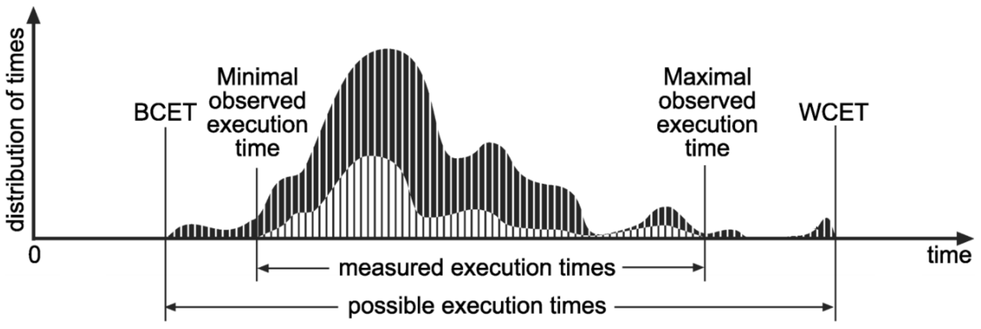

Figure 1.1: An execution time distribution, with annotated best-case execution time (BCET) and worst-case execution time (WCET), source [23] (modified)

An execution time distribution potentially contains valuable information regarding the timing behaviour of a system. While a WCET estimate merely describes the execution time in the worst case scenario, a distribution describes the execution times in all possible scenarios. It can therefore be a valuable asset to the development process of safety-critical real-time embedded systems.

1.1 Approach

The challenge with deriving an accurate execution time distributions lies in dealing with computational complexity. In order for a derived distribution to fully reflect the real-world timing behaviour of a system, all factors influencing timing behaviour have to be evaluated exhaustively. Thus the aim of this project is to develop an analysis framework capable of deriving actual execution time distributions by means of exhaustive evaluation. In doing this, the following research questions will be answered:

- Is it possible to develop an analysis that derives the actual execution time distribution by exhaustive evaluation?
- What can execution time distributions contribute to the understanding of the timing behaviour of programs?

1.2 Thesis Outline

The next chapter will discuss the fundamentals of execution time analysis, giving an overview of the factors influencing execution time, briefly discussing related work and concluding with a section on execution time distributions. Chapters 3 and 4 will cover the developed analysis framework, both in terms of design and implementation. Experimental example results will be shown and discussed in Chapter 5. Chapter 6 concludes the thesis by answering the posed research questions, giving a brief discussion of the analysis framework and touching on potential future work. An appendix is included covering how to use the framework.

CHAPTER 2

Execution Time Analysis

Execution time analysis is the process of using structured methods or tools for obtaining information about the possible execution times of a program [11]. With access to source code, program binaries and hardware specifications, automating the process of execution time analysis may seem straight-forward at first glance. However, for the general case, it is an *undecidable* problem. Developing such a tool would require one to solve the halting problem, as it involves determining whether an arbitrary program terminates at all.

Fortunately, we are in the domain of safety-critical embedded systems. Programs running on these systems are usually imposed certain restrictions such as limited or no recursion and explicitly bounded loop iterations [23]. In addition to this, termination is enforced in order to simplify the timing analysis. This overcomes the need to solve the halting problem and thus allows for the development of usable methods and tools for execution time analysis.

This chapter covers the parameters of execution time, related work and execution time distributions.

2.1 Timing Parameters

Execution time is bound to a multitude of factors, take for example input values, the contents of the CPU cache or interrupts handled by an operating system. These factors can be subsumed under the following three parameters:

1. Hardware state
2. Program input
3. Interference during runtime

2.1.1 Hardware State

Hardware state covers hardware related factors such as cache content, state of the instruction pipeline or state of the branch-predictor. It being a parameter influential to execution time is largely due to advances in modern hardware and instruction set architectures. The timing behaviour of traditional hardware is fairly predictable, instructions are executed in-order and execution times of instructions and memory accesses are known a priori (usually provided by the hardware manufacturer). Modern embedded systems however run on modern hardware and therefore make use of features like caching, instruction pipelining and speculative techniques such as branch prediction. As a consequence of this, the execution time of a sequence of instructions has become context dependent.

A memory load instruction on a cached system is a good example of how a single instruction can have varying execution times depending on context. A cache miss requires data to be retrieved from a low level in the memory hierarchy (e.g. from RAM or disk). Memory access on this level can take a considerable amount of time. A cache hit can drastically reduce this latency, thus reducing the execution time for the memory load instruction. It must however be noted that this may not always be the case. For modern processors executing instructions out-of-order, caching can produce counter-intuitive timing anomalies. As Lundqvist and Stenström [18] have shown, it is in some situations possible for a cache miss to result in a shorter execution time than a cache hit.

Given that embedded systems usually run continuously, it can be assumed that it is entirely possible for an identical piece of code to execute in different hardware contexts. The execution time of single program can thus vary significantly given the state of certain hardware elements.

2.1.2 Program Input

Program input is the most intuitive timing parameter and refers to both user-provided input values and sensor data. In contrast to hardware state, it is explicitly handled in the source code of a program. The effect of program input on timing behaviour is of course solely dependent on the algorithm implemented in the system. It is however safe to assume that the majority of real-world embedded systems contain some form of input-dependent control flow. Input-dependent control flow implies the presence of multiple possible execution paths, each of which can potentially result in a different execution time for the program.

An input value could for example explicitly dictate the amount of recursive calls, as is the case for the following recursive factorial algorithm (input value n):

```
int fac ( int n ) {
    if ( n == 0 ) {
        return 1;
    } else {
        return ( n * fac ( n - 1 ) );
    }
}
```

Listing 2.1: Recursive Factorial Algorithm

It is clear that for each unique value of n, the execution path will be different (in this case most likely requiring more execution time).

A more subtle example would be a sorting algorithm like bubble sort (input values the elements of array a):

```
void bubble_sort ( int *a, int n ) {
    int i, t, s = 1;
    while ( s ) {
        s = 0;
        for ( i = 1; i < n; i++ ) {
            if (a[ i ] < a[ i - 1 ]) {
                t = a[ i ];
                a[ i ] = a[ i - 1 ];
                a[ i - 1 ] = t;
                s = 1;
            }
        }
    }
}
```

Listing 2.2: Bubble Sort Algorithm [1]

In this case it is not necessarily the *value* of the input dictating the amount of recursive calls or loop iterations, but rather the *ordering* of the input values. The program will therefore show a relation between the 'sortedness' of the array and the possible execution times.

These two examples are rather simplistic and are merely given to illustrate input-dependent control flow. They do however clearly show that program input can have a significant influence on the execution time of a program.

2.1.3 Interference during Runtime

This parameter covers factors not directly related to a program's computations. A good example of this would be any timing related interference by an operating system, think for example of the pre-emptive scheduling of processes or the handling of interrupts.

However, due to the specialized nature of them, industrial real-time embedded systems often execute programs on a custom real-time operating systems or on bare metal (i.e. without an operating system) [8]. This greatly reduces the significance of this parameter with regard to execution time analysis.

2.2 Related Work

Research regarding execution time analysis of real-time systems has been primarily focussed on finding the worst-case execution time (see Figure 2.1). In the literature there is some ambiguity as to what a WCET implies. While some regard it as an upper *bound* on the execution time, others regard it as the *actual* worst-case execution time. Either way, while the aim of this thesis is to develop a framework capable of deriving the *distribution* of execution times, the methods and tools developed for WCET analysis are closely related.

There are two main approaches to execution time analysis; the first being based on static analysis of source code and the second being based on actual run-time measurements of execution times [23]. While the developed framework employs a measurement-based approach, both approaches will be briefly discussed in the following sections.

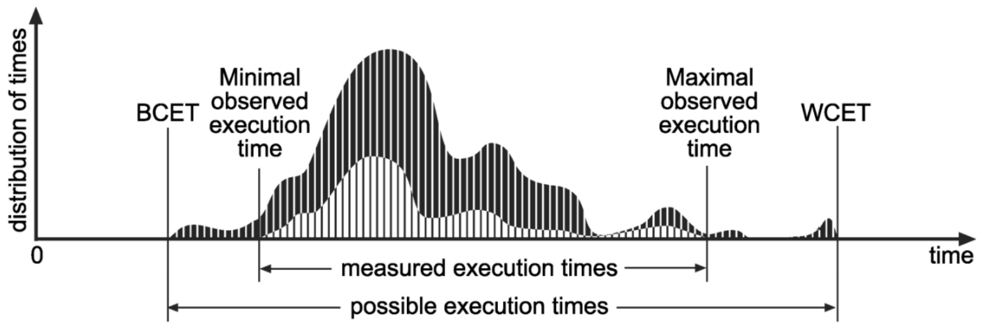

Figure 2.1: An execution time distribution, with annotated best-case execution time (BCET) and worst-case execution time (WCET), source [23] (modified)

2.2.1 Static Analysis

Static analysis is solely based on analysing program code and control flow, it does not involve any actual execution of program code. In order to be able to analyse control flow, a control flow graph[1] (CFG) is constructed from the source code. This in combination with an abstract model of the hardware provides a means for execution time analysis.

The first step involves constructing a CFG from the program code, itself not a trivial task. Generally speaking it is easier to construct one from the original source code (i.e. non-machine code), since this describes the logic and thus the flow of control at a relatively high level of abstraction. However, such a CFG may not accurately reflect the control flow exhibited by the compiled binary, since compiler optimizations and linking may alter the possible control flow

[1]A control flow graph (CFG) is a directed graph in which the nodes represent basic blocks of code (i.e. linear sequences of instructions with a single entry and exit point) and the edges represent possible control flow paths [9]

paths [20]. While construction of a CFG from machine code is possible, it can be a very involved process considering modern instruction set architectures. In many cases the actual CFG will have to be approximated [22].

In order to derive timing behaviour from control flow analysis, a model of the hardware is required. This model provides the mapping from instruction sequences to actual execution time. For modern hardware, such a model can get very complex considering modern hardware features such as caching, instruction pipelining and branch prediction [13]. Besides the challenge of accurately modelling hardware, an established model will usually be bound to a specific processor architecture or hardware platform [23].

Regardless of the inherent complexity of this approach, various static analysis tools have been proven successful in the real-time embedded systems industry (e.g. aiT [12] and Bound-T [15]). However, these tools are designed for producing WCET estimates, and are therefore not suited for deriving execution time distributions.

2.2.2 Measurement-based Analysis

In contrast to static analysis, measurement-based analysis extracts timing behaviour by taking actual run-time measurements of execution times. This method is inherently simpler. While the static approach requires both decomposition of program logic in the form of a CFG and a model of the hardware, a measurement-based approach merely requires the program code and/or binary and a means to execute it (either on real hardware or in a simulated environment).

A major advantage of a measurement based analysis methods is that all intricacies of modern hardware with regard to timing behaviour will be accounted for in the measurements; there is no need to model them. Actual execution/simulation of the program will in general also provide more accurate results, simply due to the fact that in order for static WCET analysis to produce *safe*[2] estimates, any model of context dependent timing behaviour will have to introduce overestimations to execution times [17].

The relative simplicity of measurement-based analysis does come at the expense of computational complexity. In order for timing analysis to capture all possible timing behaviour, it has to be exhaustive. With regard to program input, this means the program has to be executed with all possible combinations of input values influencing control flow. With regard to hardware state, this means all these combinations have to subsequently be executed in the context of all possible hardware states. For general purpose programs this computational complexity quickly becomes unmanageable, for small real-time embedded systems it remains feasible due to their restricted nature.

There is a commercial measurement-based tool available, RapiTime [7], it is however a commercial product and focussed on WCET analysis and therefore not suitable for this project. Due to the relative ease of implementation of measurement-based analysis, the framework discussed in this thesis employs this method.

2.3 Execution Time Distributions

An execution time distribution describes the probabilities of all possible execution times for a system (see Figure 2.1 for an example distribution). That is, it gives the likelihood of a certain execution time t occurring:

$$P(T = t) \tag{2.1}$$

Such a distribution captures the complete timing behaviour of a system and therefore contains useful information.

In the context of embedded systems, one can easily imagine how such a distribution can help in the design process, especially for system optimization. Outliers in a distribution could for example indicate the presence of corner cases of an algorithm, or indicate bottlenecks in a hardware platform. Optimizations can be verified by comparing the distributions before and after

[2]An established WCET is considered *safe* when it does not underestimate the actual WCET

applying certain optimizations. There are various hypothetical situations in which an execution time distribution can prove useful.

2.3.1 Input Value Distribution

Program input is an influential factor for the execution time of a program, as described in Section 2.1.2. It is therefore also a parameter affecting the execution time *distribution* of a program.

One should note that for real-world applications of real-time embedded systems it can be assumed that input values are not uniformly distributed. Take for example a control system in a modern car having engine temperature as an input variable. Initially the engine temperature will be low, but after driving the car for a while the engine will remain warm. In other words, a high temperature value is more likely to occur than a low one. It is evident that this input value distribution influences the execution time distribution, and should therefore be taken into account.

CHAPTER 3

Framework Design

The aim of this project is to develop an analysis that is able to derive execution time distributions of programs by exhaustive evaluation. The framework will employ a measurement-based approach, and only consider the program input timing parameter.

This chapter will describe the design choices made for the framework:

- How execution time will be measured
- Why only program input will be regarded
- How programs and their respective input will be supplied to the framework
- What tools will be used to realize the framework

3.1 Measuring Execution Time

Accurately measuring the execution time of a program can be a challenging task. Externally measuring the execution time of software running on actual hardware typically requires the use of additional hardware such as a logic analyzers to perform the timing measurements [14]. Measuring execution time in software usually requires introducing additional code to perform the measurement. This extra processing can influence the timing behaviour of the program and can thereby degrade the accuracy of the measurement. This holds especially true for real-time embedded systems, for which accurate timing is instrumental.

The framework will measure execution time by making use of a cycle-accurate system simulator, *gem5* [10]. Using this simulator for program execution drastically simplifies the problem of measuring execution time, since gem5 has built-in timing mechanisms. These allow for the execution time of a program to be obtained in terms of CPU cycles, an accurate reflection of program control-flow timing. While physical execution time is bound to the speed of execution/simulation, a cycle count is invariant to it. The major downside of simulation is that the accuracy of the timing measurement will be bound to the accuracy of the simulation. However, for our purposes this will be taken for granted.

3.2 Timing Parameters

As described in Section 2.1, there are three main parameters influencing the timing behaviour of a program. It was decided for the framework only to regard the program input parameter, thus deriving an execution time distribution solely by exhaustively evaluating input variables specified by the end-user.

The *hardware state* parameter has been excluded due to the inherent complexity of parametrising it given the time frame of the project. In theory it should be possible to devise a way in which hardware state can be injected into the simulator, given that gem5 is a modular platform providing a high degree of customization with regard to how individual hardware components (CPU, RAM and caches) are simulated. However this would require heavy modification to the simulator as is, something deemed beyond the scope of this project.

The third parameter (*interference during runtime*) has been excluded for reasons described in Section 2.1.3, namely the fact that programs running on safety-critical real-time embedded systems often run on specialized real-time operating systems or even without an operating system at all. Not including this parameter will thus assume non-preemptive scheduling, i.e. that a program is allowed to finish without being temporarily interrupted.

Solely focussing on program input means that any execution time distribution derived by the framework will not fully reflect the real-world timing behaviour of a program. However, this does not mean that a derived distribution has no utility. Such a distribution can still provide useful insights into how a program responds to input values.

3.3 Input Programs

The framework aims at being a tool to be used in the development process of real-time embedded systems. Therefore the input programs will not be regarded as black boxes, i.e. availability of program source code and prior knowledge about input values will be assumed. This will allow the framework to compile the program with a certain a input and subsequently execute it in the simulator.

The simulator used is capable of simulating a wide array of instruction set architectures[1]. The user must therefore also supply an appropriate cross compiler[2] for the desired target architecture. Since gem5 features a mode in which system calls are emulated, it is capable of directly executing Linux program binaries. This avoids the user having to also provide basic system and memory management facilities alongside the input program.

3.3.1 Program Input Specification

A priori knowledge about the program input is assumed, including the data types of the input variables and possibly ranges on these values. The framework should provide a compact means of describing these, thereby allowing it to algorithmically generate all input values on-the-fly.

A compact representation is needed since storing all possible input values explicitly would introduce significant and a likely unmanageable overhead in memory (for example, storing all possible 32-bit integer values requires 16GB of memory[3]).

Input Value Distribution

To let the derived execution time distribution more accurately reflect real-world behaviour of real-time embedded systems, the framework will also allow for an input value probability distribution to be specified for each input variable. The implementation of this will need to tie in with the compact representation of the input values.

3.4 Tools

The involved tasks of compilation and simulation will be handled by a cross compiler and gem5 respectively. The tasks of the framework itself are limited to the generation of input values and the aggregation of simulation results. Therefore, in order to allow for relative ease and speed of development, the framework will be written in an interpreted language.

[1] ALPHA, ARM, MIPS, Power, SPARC, x86

[2] A cross compiler is able to compile code to an architecture other than that of the host system

[3] 2^{32} possible values, occupying 4 bytes each, $2^{32} \cdot 4 = 17179869184$ bytes = 16 gigabytes

The interpreted language of choice is *Python* [6]. This language was chosen mainly due to its extensive standard library (providing a vast array of built-in features) and the availability of established external libraries such as *NumPy* [5] (a library for numerical scientific computing) and *matplotlib* [3] (a plotting library). Making use of built-in language features and external libraries will very likely result in a relatively compact codebase for the framework, making it easy to modify and extend.

CHAPTER 4

Framework Implementation

This chapter discusses the implementation of the framework, highlighting several technical and mathematical aspects of its realisation. The first section gives an overview of the framework, describing its internal architecture. The following section describes how input programs and their respective input are specified. After this, a section on how the input is generated, followed by a section on how the framework derives the final execution time distribution.

4.1 Overview

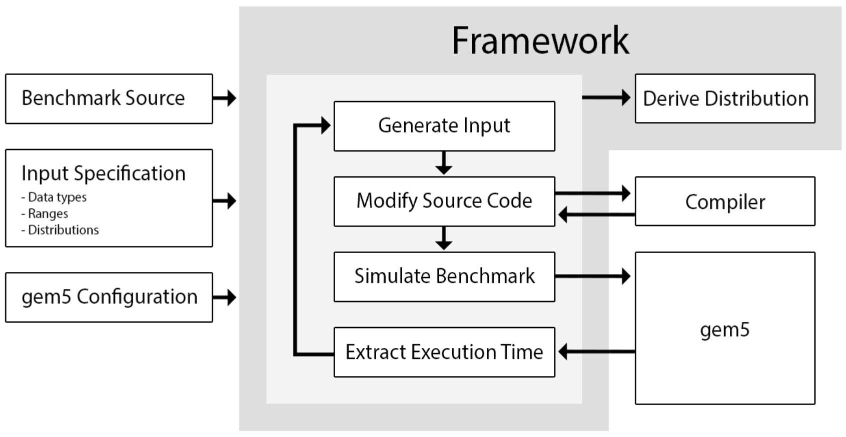

Figure 4.1: Internal architecture of framework

The framework derives the execution time distribution of an input program (benchmark) by simulating its execution for all possible combinations of input values. In order to do this, the end-user must supply the benchmark source code along with a specification of its input variables. Exhaustive evaluation of the input space is performed using the following measurement cycle:

1. Generate next combination of input values
2. Modify benchmark source code, inject input values
3. Compile benchmark with external compiler
4. Simulate benchmark in *gem5*
5. Extract and save execution time

After exhausting the input space, the results are processed and the execution time distribution is saved and plotted. See Figure 4.1 for a graphical representation of the internal architecture of the framework.

Note that while benchmarks are not inherently bound to any specific programming language, only benchmarks written in the C programming language have been tested.

4.2 Benchmark Format

In order for the framework to be able to inject input values and perform proper timing measurements, some minor adjustments will have to be made to the original benchmark source code.

The input values generated by the framework are injected simply by finding and replacing certain placeholder values in assignment statements. To facilitate this, the framework requires the initialization of input variables to be decoupled from the rest of the benchmark (see Listing 4.1 and 4.2 below for example code). Every benchmark is accompanied by a template file, linking generated input values with the placeholder values in the *init.c* file.

```
#include "init.c"
#include "m5op.h"

...

int main() {
    benchmark_init();
    m5_reset_stats();
    benchmark_main();
    return( benchmark_return() );
}
```

Listing 4.1: benchmark.c

```
type v_1;
type v_2;
...
type v_n;

void benchmark_init() {
    v_1 = placeholder_v_1;
    v_2 = placeholder_v_2;
    ...
    v_n = placeholder_v_n;
}
```

Listing 4.2: init.c

In order to exclude any overhead in the execution time due to the initialization of a benchmark and its input variables, the main code body of a benchmark is preceded with a call to the *m5_reset_stats()* function. This ensures that only the cycles required for the actual benchmark computations are counted. The benchmark executable will thus have to be linked to the *gem5* utility library.

4.3 Input Specification

Alongside the benchmark source code, the framework requires a specification of the input variables. This specification will dictate the input type and any other appropriate attributes such as *value range* or *value probability distribution*. This specification will provide the framework with enough information to exhaustively generate all possible combinations of input values. The *value probability distribution* will be used in a later stage to derive the final execution time distribution.

4.3.1 Input Type

Internally, input type is described by the *Input* class, in essence providing nothing more than an interface to a generator function and the distribution of an input variable. Currently the framework implements three ranged basic data types, two array types and one fixed data type, Table 4.1 gives a brief description for each of these.

Type	**Attributes**	**Description**
int	Range (*min*, *max*)	Whole numbers
	Distribution	Generates full range from *min* to *max*
float	Range (*min*, *max*)	Single precision IEEE 754 floating point number
	Distribution	Generates full range from *min* to *max*
double	Range (*min*, *max*)	Double precision IEEE 754 floating point number
	Distribution	Generates full range from *min* to *max*
array	Size (n)	Array of integers
		Generates all combinations of set $\{0, 1, .. \ n-1\}$
uniquearray	Size (n)	Array of integers
		Generates all permutations of set $\{0, 1, .. \ n-1\}$
int_fixed	Value (v)	Single integer value
		Generates the single integer value v

Table 4.1: Supported input types

Note that the framework is not restricted to just these six input types. Since data types and their implementation can be platform specific, especially considering the variety of platforms for embedded systems, new input types can easily be defined by the end-user.

4.3.2 Value Probability Distribution

Ranged input types (*int*, *float* or *double*) allow for a value probability distribution to be specified. Since the framework generates discrete input values, the function to describe this distribution must be a probability mass function[1] (PMF). Accurately modelling real-world input distributions can be challenging, especially when a distribution can not be described using a (single) mathematical function. The framework therefore implements the PMF of an input distribution as a normalized combination of several individual probability distributions. This provides a flexible means of modelling real-world input distributions.

Given a ranged input type, the end-user can assign subsets of the full range a certain probability distribution along with a ratio dictating its contribution to the compound probability mass function. The manner in which this is implemented is best illustrated by an example: Table 4.2 contains a value probability distribution specification for an *int* input type with range $[0, 99]$, Figure 4.2 shows the normalized combined probability mass function.

Subrange	Ratio	Distribution
$[0, 40)$	2	Gaussian, $P(I = v) = \exp(-\frac{(v-31)^2}{2 \cdot 8.0^2})$
$[40, 50)$	1	Uniform, $P(I = v) = 1.0$
$[50, 71)$	4	Gaussian, $P(I = v) = \exp(-\frac{(v-62)^2}{2 \cdot 8.0^2})$
$[71, 100)$	2	Uniform, $P(I = v) = 1.0$

Table 4.2: Example specification of value probability distribution for an integer input type with range $[0, 99]$

[1]A probability mass function gives an exact probability for a given value

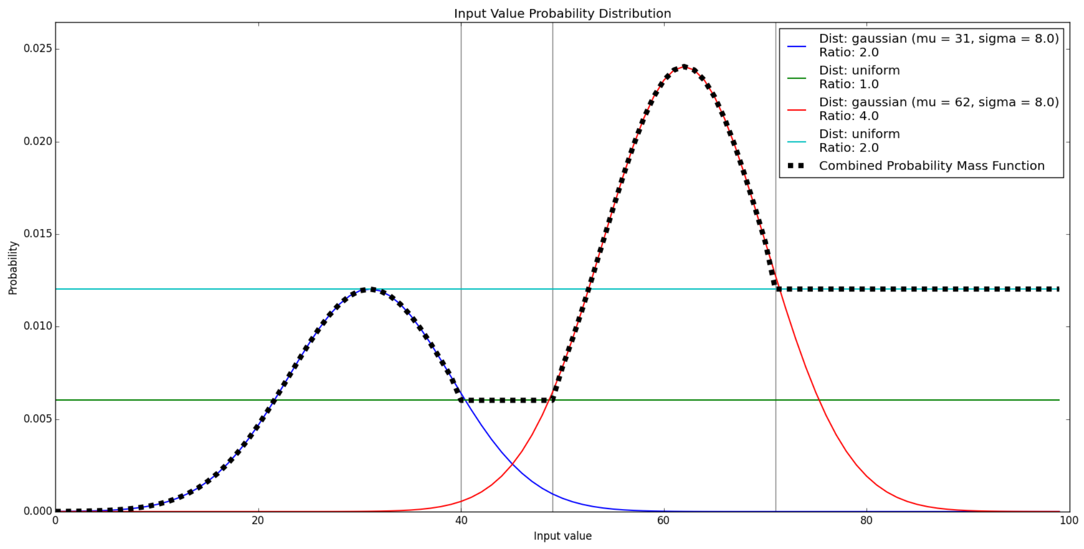

Figure 4.2: Example compound probability mass function

The framework allows complete freedom in defining the distribution functions for subranges. Several distributions have been readily provided (uniform, Gaussian, normal), but the end-user is able to let any custom Python function define the distribution for a subrange.

4.4 Input Generation

As mentioned in Section 3.3.1, explicitly storing all possible input values would require enormous amounts of memory. In order to avoid memory starvation, the framework makes use of Python's *generator* functions [2] to generate input values *lazily* (or *on demand*). This means input values can be sequentially generated, avoiding the need to store all possible values.

Generator functions in Python maintain state after being called, allowing them to return a follow-up result on every call. Take for example a generator function for the *int* input type:

```
def gen_ints(a, b):
    for i in xrange(a, b):
        yield i
```

Listing 4.3: Generator function for *int* input type

On every subsequent call, this function returns the next integer in the range $[a, b)$.

The use of generator functions makes even more sense when considering multiple input values. Since the framework exhaustively evaluates the input space of a benchmark, all *combinations*[2] of input values have to be generated. Not using some form of *lazy* generation for this would require immense amounts of memory, as demonstrated by the following example:

Consider a program with three unsigned 32-bit integer input variables, the range of each spanning the entire range of the data type (i.e. $[0, 2^{32} - 1]$). There are approximately $8e^{28}$ possible combinations of the input values ($2^{32} \cdot 2^{32} \cdot 2^{32}$), each one requiring 12 bytes of memory (3 values, 4 bytes each). Naively storing all possible combinations would thus require approximately $9.5e^{20}$GB of memory[3], something far from practical with current technology.

[2]Combinations in this context means the Cartesian product of the n individual sets of input values ($I_1 \times I_2, ...I_n$), that is, the set of all ordered n-tuples

[3]950,000,000,000,000,000,000 GB

Floating point numbers

Generating ranges of floating point numbers is not as trivial as it is for integers numbers. Since the generated sequence needs to cover all representable floating point numbers between a lower and upper bound, naively incrementing the value by a constant will not suffice (as is the case for integers). The framework implements generation of IEEE 754 floating point ranges by making use of the *nextafter(x, y)* function defined in the C99 standard [16] (as implemented in the *NumPy* library [4]). This functions returns the next representable floating point number following x in the direction of y.

4.5 Deriving an Execution Time Probability Distribution

In general, any discrete distribution can be captured by a frequency table. The framework therefore stores all measured execution times in a relative frequency table. A relative frequency table, as opposed to a regular frequency table, is required to reflect the value probability distributions of input variables.

4.5.1 Frequency Table

Disregarding any input value probability distribution, the execution time distribution can be established simply by storing the measured execution times in a regular frequency table. In such a table, each entry is accompanied by an integer value dictating the number of occurrences of that entry, see the example table below:

Execution time (cycles)	Frequency
1316000	3
1187000	11
1156000	3
1116000	1
1107000	6
719000	6

Table 4.3: Example frequency table

In order to derive an execution time *probability* distribution from this table, the data has to be normalized to make all discrete probability values add up to 1.0. This can be achieved simply by dividing all individual frequencies by the sum of all frequencies:

Execution time (cycles)	Probability
1316000	0.1
1187000	0.36666667
1156000	0.1
1116000	0.0333333
1107000	0.2
719000	0.2

Table 4.4: Example probability distribution

4.5.2 Incorporating Input Value Probability Distributions

Using a regular frequency table, it is apparent that each measurement is given the same *weight*. Each execution time resulting from a simulation with a combination of input values contributes exactly 1 to the appropriate table entry. Aggregating the measurements using a regular frequency table thus employs a trivial update function. For a measured execution time t, this function can be described using the following formula:

$$\text{Frequency}(t) = \text{Frequency}(t) + 1 \tag{4.1}$$

This update function however does not take into account any specified input value probability distribution, every input value is implicitly assumed just as likely to occur. Intuitively, an execution time resulting from a high probability input value should contribute more to the final distribution than one with a low probability. The framework achieves this by adding the probability of an input variable I taking on the value v as a weight factor to the update function:

$$\text{Relative frequency}(t) = \text{Relative frequency}(t) + (1 \cdot P(I = v)) \tag{4.2}$$

Since the weight factor is equal to the probability value of the input value that results in execution time t; it can be obtained directly from the probability mass function describing the value distribution of the input variable. A consequence of this modified update function is that the resulting table will no longer contain absolute frequencies. It has become a relative frequency table, with the values indicating the occurrences of execution times in relation to each other.

The process of deriving the execution time probability distribution from the relative frequency table as described above is identical to that for the regular frequency table. All relative frequencies are normalized by dividing them by their total sum.

Multiple input variables

The update function described above only considers a single input variable and value distribution. When an execution time is derived from multiple input values, the weight factor must describe the joint probability of the individual input values. The framework implements this by assuming that the events[4] are mutually independent. For mutually independent events, the joint probability is given by the following formula:

$$P\left(\bigcap_{i=1}^{n} I_i\right) = \prod_{i=1}^{n} P(I_i) \tag{4.3}$$

The update function for the relative frequency table reflecting multiple input value distributions is therefore as follows:

$$\text{Relative frequency}(t) = \text{Relative frequency}(t) + \prod_{i=1}^{n} P(I_i = v_i) \tag{4.4}$$

Dependent events

For real-world embedded systems, it could be that the assumption of mutually independent events is false. Especially when input values are obtained from external sensors, it is possible that certain combinations of input values are more likely to occur than others. To reflect this in the derived execution time, the framework would require a specification of the *joint* value probability distribution of a set of input variables. While the framework currently does not support dependent events, there are various ways in which it could be implemented. What follows is a brief description of two ways in which this could be done.

One way in which the framework could be extended to support this would be by allowing the end-user to specify a multivariate function f explicitly describing the joint probability distribution for certain input variables:

[4]The event of an input variable taking on a certain value

Input variable	Subrange	Joint Probability
I_1	$[a_1, b_1)$	
I_2	$[a_2, b_2)$	
I_3	$[a_3, b_3)$	$P\left(\bigcap_{i=1}^{n} I_i\right) = f(v_1, v_2, v_3 ... v_n)$
...		
I_n	$[a_n, b_n)$	

Table 4.5: Example specification of joint input value probability distribution

This way, the weight factor in the update function would be directly obtained from the function f.

In a similar fashion, another way of implementing it could be by letting the end-user specify a function g describing the *relations* between the individual probabilities $(p_1, p_2, ...p_n)$, thus retaining the individual input value probability distributions:

$$P\left(\bigcap_{i=1}^{n} I_i\right) = g(p_1, p_2, ...p_n) \tag{4.5}$$

Both approaches however require finding or constructing a multivariate function. One can imagine that this can be a daunting task. Modelling a function in two dimensions (as is the case when modelling the PMF of a single input variable) is visually intuitive. When considering n-dimensions however, it becomes very hard to visualize what such a function might look like.

CHAPTER 5

Example Distributions

This chapter features example execution time distributions, demonstrating the framework and its capabilities. In order to illustrate what typical execution time distributions may look like and what information they might convey, several benchmarks from an existing embedded systems benchmark suite (TACLeBench[1]) have been adapted and implemented.

5.1 Setup

5.1.1 Hardware

Simulated hardware configuration:

- **Architecture:** ARMv8-A (AArch64) [2]
- **CPU Clock:** 500Mhz
- **L1 Instruction Cache Size:** 16kB
- **L1 Data Cache Size:** 64kB
- **L2 Cache Size:** 128kB

5.1.2 Experiments

1. **Recursive Factorial Sum**: showing the influence of an input value distribution on the execution time distribution.
2. **Naive Primality Test**: showing how certain characteristics of an algorithm can be reflected in its execution time distribution.
3. **Bubble Sort**: showing the execution time distribution of a program with non-trivial input dependent control flow.
4. **Insertion Sort**: illustrating the effect of caching on an execution time distribution.

[1]http://www.tacle.eu/

[2]All benchmarks were cross-compiled using GCC 5.3.0

5.2 Results

Note: all the execution time distributions are plotted as bar graphs, the annotated *BCET* and *WCET* values in the graphs are defined as the minimal and maximal *measured* execution times.

5.2.1 Recursive Factorial Sum

- **Algorithm**: This program computes the sum of factorials from 0 to n, the factorials are computed recursively
- **Input variables**:
  - n, integer

The following input specifications were supplied to the framework:

	Input variable	**Type**	**Range**	**Value distribution**
Specification 1	n	int	$[0, 200)$	$[0, 200)$: Uniform See Figure 5.1
Specification 2	n	int	$[0, 200)$	$[0, 200)$: Gaussian curve See Figure 5.2

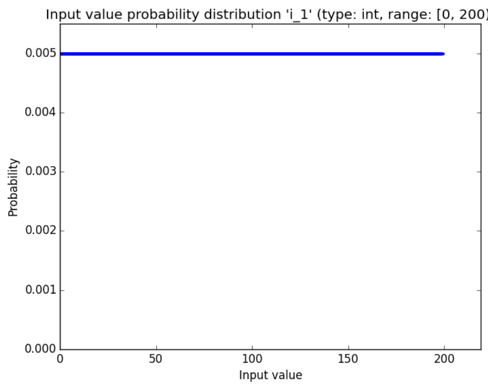

Figure 5.1: Uniform input value probability distribution for factorial benchmark

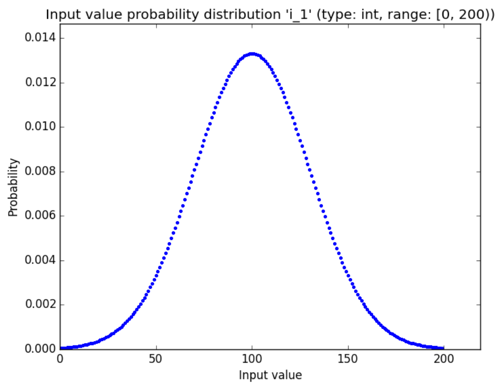

Figure 5.2: Gaussian input value probability distribution for factorial benchmark

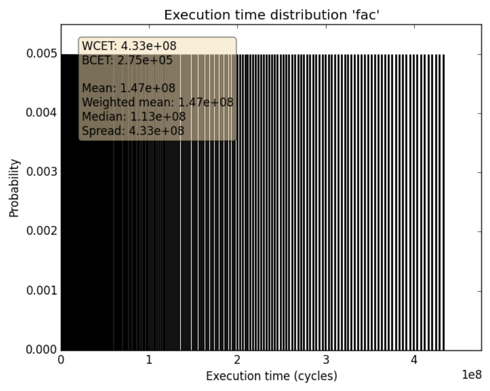

Figure 5.3: Execution time distribution for factorial benchmark with uniform input value distribution

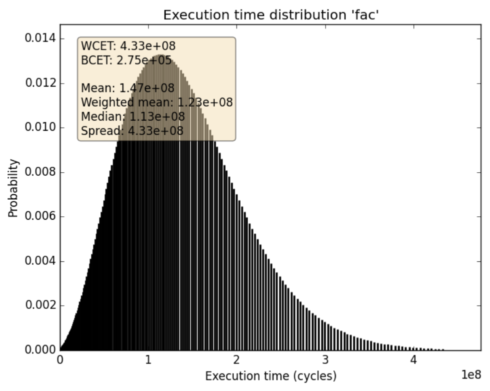

Figure 5.4: Execution time distribution for factorial benchmark with Gaussian input value distribution

Discussion

Since the implemented algorithm produces distinct execution times for every value of n (n directly dictates the amount of recursive calls), the derived distributions clearly reflect the input value distributions. There is a one-to-one mapping of the input distribution to the execution time distribution.

For a uniform input value distribution, Figure 5.3 confirms what one would expect. Namely a uniform execution time distribution. Figure 5.4 shows how the non-uniform input value distribution is reflected in the final execution time distribution. It is interesting to see that the symmetric Gaussian curve of the input distribution is skewed slightly in the derived execution time distribution. This is due to the non-linear increase in execution time for incremented values of n (i.e. execution times in the low end of the spectrum are more densely packed).

5.2.2 Naive Primality Test

- **Algorithm**: This program determines whether the number n is prime by means of trial division
- **Input variables**:
  - n, integer

The following input specification was supplied to the framework:

	Input variable	**Type**	**Range**	**Value distribution**
Specification 1	n	int	$[0, 50000)$	$[0, 50000)$: Uniform

Results

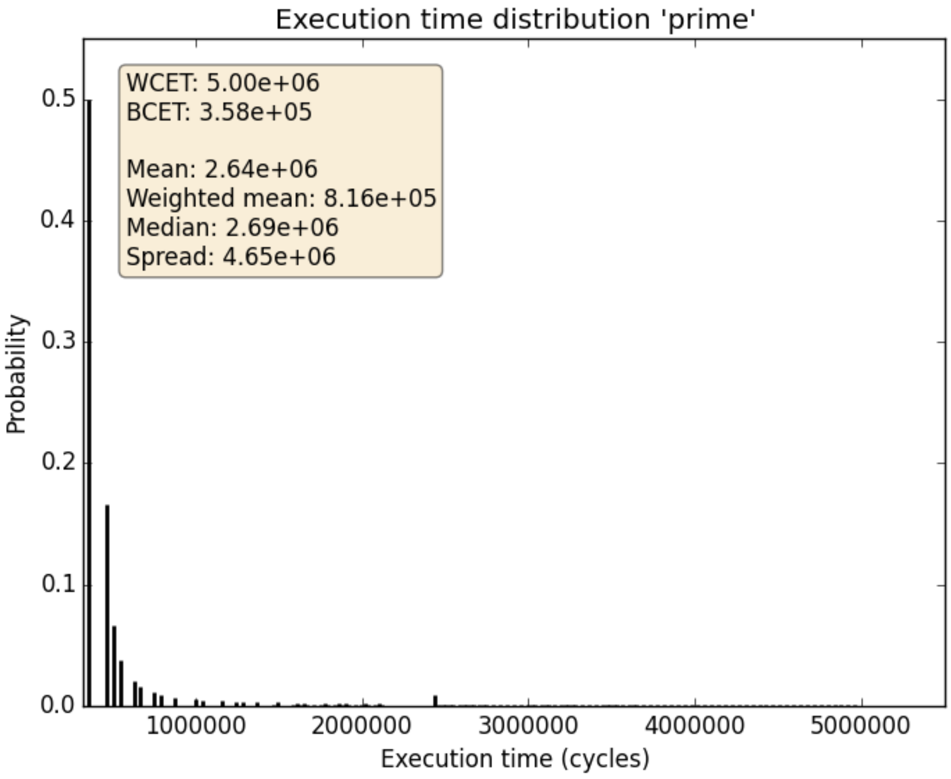

Figure 5.5: Execution time distribution for prime benchmark with uniform input value distribution

Discussion

This benchmark illustrates how an execution time distribution can display characteristics of the implemented algorithm. In this case it clearly reflects the sparseness of prime numbers. Since a prime number has no positive divider other than 1 and itself, it requires the maximum amount of possible trial divisions (and thereby requires more processing time than a neighbouring non-prime). The relative high probability of a low execution time displayed in Figure 5.5 is due to the fact that most numbers are not prime and thus do not require many trial divisions. The prime numbers are embodied in the tiny peaks at the higher end of the execution time range.

5.2.3 Bubble Sort

- **Algorithm**: This program sorts an array of integers by stepping through the array and swapping adjacent elements if they are not in the in the right place, this process is repeated until the array is sorted
- **Input variables**:
  - a, fixed size array of integers

The following input specifications were supplied to the framework:

	Input variable	**Type**	**Array size**
Specification 1	a	uniquearray	6
Specification 2	a	array	6

Results

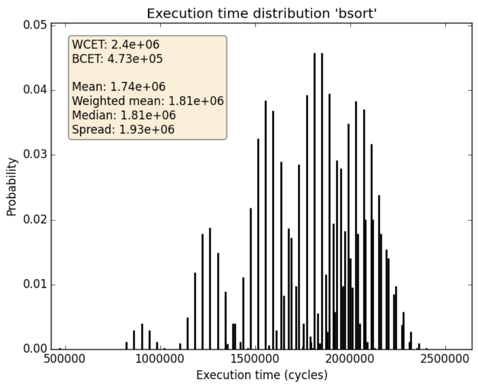

Figure 5.6: Execution time distribution for bubble sort benchmark with uniquearray input type

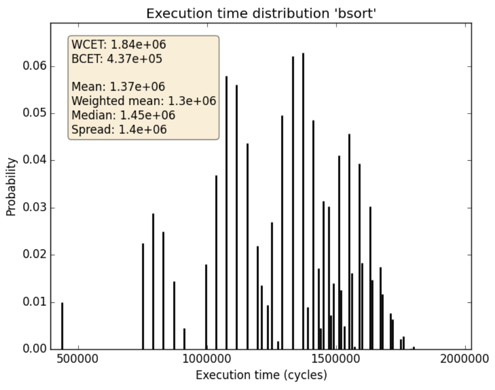

Figure 5.7: Execution time distribution for bubble sort benchmark with array input type

Discussion

In contrast to the factorial benchmark, this benchmark exhibits a more varied execution time distribution. Both the distributions for the array permutations (Figure 5.6) and the array combinations (Figure 5.7) contain visually distinctive clusters or peaks. It is interesting to see that an algorithm such as bubble sort seems to display a typical distribution for both input types, as both distributions are visually similar.

5.2.4 Insertion Sort

- **Algorithm**: This program sorts an array of integers by building up a sorted array by sorting one element at a time
- **Input variables**:
  - a, fixed size array of integers

The following input specification was supplied to the framework:

	Input variable	**Type**	**Size**
Specification 1	a	uniquearray	6

The simulation was run twice, once with the simulated hardware cache disabled and once with it enabled. Note that in contrast to the bubble sort experiment, the two simulations were run with *identical* input.

Results

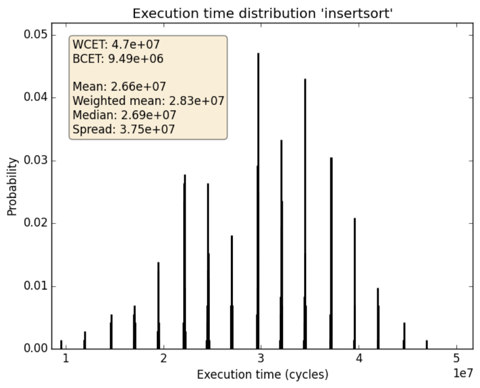

Figure 5.8: Execution time distribution for insertion sort benchmark with hardware cache disabled

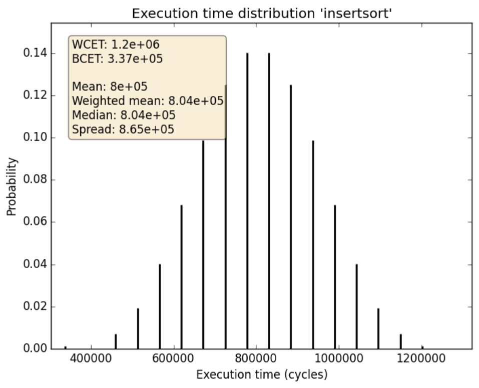

Figure 5.9: Execution time distribution for insertion sort benchmark with hardware cache enabled

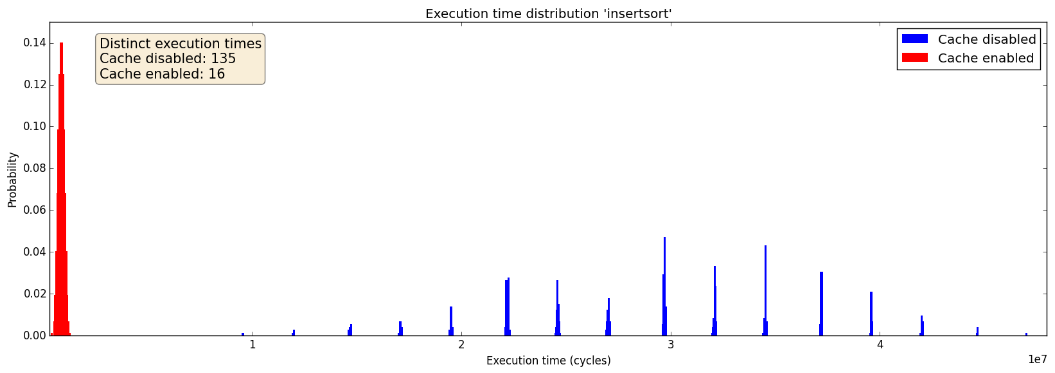

Figure 5.10: Comparison of execution time distributions for insertion sort benchmark with hardware cache enabled and disabled

Discussion

Comparing the statistics from both the non-cached (Figure 5.8) and the cached (Figure 5.9) simulations, we can clearly see an overall increase in performance. Looking at the weighted means, we can derive an average increase in performance with a factor of approximately 35[3].

While an increase in overall performance is to be expected when introducing a cache, seeing how it affects the execution time distribution reveals more detailed information. Looking at both distributions plotted on the same scale (Figure 5.10), we can see that enabling the hardware cache has not only significantly shifted the centre of the distribution, but has also condensed the distribution itself. The non-cached distribution contains 135 distinct execution times while the cached distribution only contains 16. What is also interesting to see is that despite the reduction in data points, the general shape of the distribution is retained. Caching seems to have smoothed out the distribution.

From this we can conclude that execution time distributions are able to *visualize* the effects of caching (and possibly other hardware features). Instead of just a numerical average increase in performance (weighted mean), a distribution is able to show the effect of something like caching on a program's timing behaviour with much more detail.

[3] $\frac{2.83e^7}{8.04e^5} \approx 35$

CHAPTER 6

Conclusion

Safety-critical real-time embedded systems are subjected to stringent timing constraints. Correctness of these systems is not only tied to functional correctness, but also to temporal constraints. Proper analysis of the timing behaviour of such systems is therefore of great importance. While previous research on execution time analysis has been mainly on focussed on establishing upper bounds on execution time, this thesis presents an analysis framework capable of deriving the *distribution* of execution times.

The presented framework shows that it is possible to derive an execution time distribution by means of exhaustive evaluation. The framework is capable of measuring the execution times of a program for all possible combinations of input values. To realistically reflect the timing behaviour of real-world embedded systems in the derived distributions, the framework also allows for input value probability distributions to be specified. Memory complexity is minimized by providing a compact means of specifying input and generating combinations of input values *on demand*.

The provided example distributions show that execution time distributions can contain interesting information regarding the timing behaviour of a program. Not only are they able to reflect characteristics of the underlying algorithms and input spaces, they are also able to visualize the effect of input value distributions and hardware features such as caching. Furthermore, one can easily imagine that execution time distributions derived before and after an alteration to hardware or program code can prove a valuable asset in the development process of embedded systems.

The research presented in this thesis is far from conclusive and the developed framework is therefore not an all-encompassing timing analysis tool for deriving *actual* execution time distributions.

The experiments only regarded relatively small input spaces and thus did not require unmanageable amounts of processing time; larger input spaces might prove problematic in this regard. Any timing behaviour related to hardware state or runtime interference is also unaccounted for, since only the input space is evaluated. Moreover, the framework executes programs in a simulated environment, not on actual hardware. It is for these reasons that the derived distributions may not accurately reflect the real-world timing behaviour of a system.

However, it has been demonstrated that even incomplete execution time distributions can provide interesting insights into the timing behaviour of systems. The presented research and framework should therefore be regarded as the first steps towards an analysis capable of deriving realistic execution time distributions.

6.1 Future work

There are several aspects of the framework which can be improved and expanded upon. Two of these have already been addressed in the thesis, namely the possibility of hardware state injection with the gem5 simulator and an implementation of joint value probability distributions. A third noteworthy aspect is runtime performance.

While the framework in its current state performs simulations purely sequentially, there is a clear potential for multi-threading in the exhaustive evaluation of input space. Since the results of subsequent simulations are completely independent, simulations for entire input ranges can easily be divided into multiple workloads. This would not require any drastic alterations to the framework as is, and could drastically reduce the amount of processing time required to derive an execution time distribution.

APPENDIX A

Framework Guide

A.1 Installation

Requirements:

- Python 2.7.11[1]
- gem5[2]
- A cross compiler for target hardware architecture (e.g GCC)

The ETD framework itself can be obtained from GitHub:

- https://github.com/BDWN/etd

The framework requires the following Python packages:

- Matplotlib (tested with version 1.4.2)[3]
- Numpy (tested with version 1.9.1)[4]

These packages can be installed either manually or via the supplied requirements.txt file using the `pip` package manager:

```
pip install -r /path/to/requirements.txt
```

A.2 Configuration

After having successfully installed all required software and downloading the framework, make sure to adjust the `config.py` file before attempting to run the framework. This file contains global settings, gem5 configurations and framework output settings.

[1]https://www.python.org/downloads/
[2]https://github.com/gem5/gem5
[3]http://matplotlib.org/
[4]http://www.numpy.org/

A.3 Example benchmark specification

In order for a benchmark to be used by the framework the following points have to addresses:

- Input variable initializations must be moved to a separate file
- A complementary `gen_input.py` must be created
- A call to `m5_reset_stats()` must be added to the benchmark
- A `Makefile` must be supplied, linking the compiled binary to the gem5 utility library
- The benchmark input specification must be added to `benchmarks.py`

What follows is a short guide on preparing the following benchmark for use with the framework:

```
int x = some_value; // Input variable
int y = some_value; // Input variable

int main() {
    x = some_value;
    y = some_value;
    int z = 0;
    for (int i = 0; i < x * y; i++) {
        z += i;
    }
    return 0;
}
```

Listing A.1: test_bench.c

A.3.1 Detaching input variable initialization

First, make sure the benchmark is contained in its own directory:

```
/path/to/etd_framework/bench/test_bench
```

Now, move the initialization of desired input variables in the source code to a new file (`init.c`) and include it in the main source file[5]. The next step is to include the `m5op.h` file (provided with the framework, located in the `bench` directory) and prepend the main benchmark computation with a call to reset the simulator stats. See the example below:

```
#include "m5op.h"
#include "init.c"

int main() {
    benchmark_init();
    m5_reset_stats();
    int z = 0;
    for (int i = 0; i < x * y; i++) {
        z += i;
    }
    return 0;
}
```

Listing A.2: test_bench.c

```
int x;
int y;

void benchmark_init() {
    x = some_value;
    y = some_value;
}
```

Listing A.3: init.c

[5]Note that it is not required to assign values to variables in the `init.c` file since these will be injected by the framework. However, it can be useful to assign some test values to ensure the program still behaves as expected.

A.3.2 Creating gen_input.py file

Copy one of the existing `gen_input.py` files from one of the example benchmarks directory[6] to the `test_bench` directory and supply it the contents of the `init.c` as follows:

```
#!/usr/bin/env python2

import json
import argparse
from os.path import join, dirname, realpath
from string import Template

f = open(join(dirname(realpath(__file__)), "init.c"), "w")
t = Template(
"""
int x;
int y;
void benchmark_init() {
    x = $x;
    y = $y;
}
""")

if __name__ == '__main__':
    parser = argparse.ArgumentParser(description="Generate init file")
    parser.add_argument('-i', '--input', type=json.loads)
    args = parser.parse_args()
    f.write(t.substitute(args.input))
    f.close()
```

Listing A.4: gen_input.py

Note that `$x` and `$y` are the framework's internal names for the input variables. They will be used in the input specification to identify the specific variables. Make sure these are prepended with '$'.

A.3.3 Makefile

In order for the framework to be able to compile the benchmark, a `Makefile` must be provided. To facilitate the call to `m5_reset_stats()`, the compiled binary will have to be linked to the gem5 utility library. See the example Makefile below:

```
CC=aarch64-linux-gnu-gcc
CFLAGS=-march=armv8-a -static
OBJS=../m5op_arm_A64.o

all: fac

fac: fac.o
    $(CC) $(CFLAGS) -o a.out fac.o $(OBJS)

fac.o: fac.c
    $(CC) $(CFLAGS) -c fac.c

clean:
    -rm -f *.o
    -rm -f a.out
```

Listing A.5: Makefile

The `m5op.arm.A64.o` file is provided with both the framework (in the `bench` directory) and can also be found in the gem5 working directory.

[6] `/path/to/etd_framework/bench/xx/gen_input.py`

A.3.4 Input specification

After having properly modified the original benchmark source code and setting up the appropriate `gen_input.py` file, the benchmark input specification can be added to the `benchmarks.py` file as follows:

```
from input import Types
from distributions import *

benchmarks = {
    ...
    "test_bench" : { "path" : "/path/to/etd_framework/bench/test_bench/",
                     "input": [
                                  ("x", (Types.int, 1, 20,
                                    [
                                        (1, 10, (2, uniform_dist(1.))),
                                        (10, 20, (1, uniform_dist(1.))),
                                    ])
                                  ),

                                  ("y", (Types.int, 1, 30,
                                    [
                                        (1, 30, (1, uniform_dist(1.))),
                                    ])
                                  ),
                              ]
    },
    ...
}
```

Listing A.6: benchmarks.py

This example input specification defines both x and y as integer input types and sets their ranges at $[1, 20)$ and $[1, 30)$ respectively. For y, the value probability distribution is uniform. For x, a value in the range of $[1, 10)$ is deemed twice as likely as a value in the range of $[10, 20)$ (specified by the ratio $2 : 1$).

In general, an input specification must adhere to the following format:

```
# Input types are defined in input.py
# Distributions are defined in distributions.py

"name" : { "path" : "path_to_benchmark_folder",
           "input": [
                     # Ranged input type
                     (
                       "variable_placeholder_name",
                       (input_type,
                        lower_bound, upper_bound,
                        # Input value distribution
                        [
                            (min_val, max_val, (ratio, dist),
                            (min_val, max_val, (ratio, dist),
                            ...
                            (min_val, max_val, (ratio, dist),
                        ])
                     ),

                     # Array input type
                     (
                       "variable_placeholder_name",
                       (input_type, array_size)
                     ),

                     # Fixed input type
                     (
                       "variable_placeholder_name",
                       (input_type, value)
                     ),

                ]
}
```

A.4 Running the framework

The framework can be run by executing `main.py`. It will run the simulations, save the results and plot both the input value probability distributions and the derived execution time distribution.

```
usage: main.py [-h] [-n] [-o] [-d] bench
positional arguments:
  bench name of benchmark

optional arguments:
  -h, --help show this help message and exit
  -n, --nosim do not run simulations, only plot previous results
  -o, --overwrite force overwriting of any previous output
  -d, --debug show compilation and simulator output
```

Bibliography

[1] Sorting algorithms/bubble sort. http://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort#C. Accessed: 03-05-2016.

[2] Pep 255 – simple generators. https://www.python.org/dev/peps/pep-0255/. Accessed: 27-05-2016.

[3] matplotlib. http://matplotlib.org/. Accessed: 24-05-2016.

[4] Numpy core math library. http://docs.scipy.org/doc/numpy-1.10.1/reference/c-api.coremath.html#c.npy_nextafter. Accessed: 27-05-2016.

[5] Numpy. http://www.numpy.org/. Accessed: 24-05-2016.

[6] Python. https://www.python.org/. Accessed: 24-05-2016.

[7] Rapitime. https://www.rapitasystems.com/products/rapitime. Accessed: 19-05-2016.

[8] P Albertos, A Crespo, M Vallés, and I Ripoll. Embedded control systems: some issues and solutions. In *16th IFAC World Congress*, 2005.

[9] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, July 1970. ISSN 0362-1340. doi: 10.1145/390013.808479.

[10] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 39(2):1–7, August 2011. ISSN 0163-5964. doi: 10.1145/2024716.2024718.

[11] Andreas Ermedahl and Jakob Engblom. Execution time analysis for embedded real-time systems. *Handbook of Real-Time Embedded Systems*, pages 35–1, 2007.

[12] Reinhold Heckmann and Christian Ferdinand. Worst-case execution time prediction by static program analysis. In *In 18th International Parallel and Distributed Processing Symposium (IPDPS 2004, pages 26–30. IEEE Computer Society)*, 2004.

[13] Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of wcet tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003.

[14] Nat Hillary and Ken Madsen. You cant control what you cant measure, or why its close to impossible to guarantee real-time software performance on a cpu with on-chip cache. In *Proc. of the 2ndInt. Workshop on WCET Anal*, 2002.

[15] Niklas Holsti, Thomas Langbacka, and Sami Saarinen. Worst-case execution time analysis for digital signal processors. In *Signal Processing Conference, 2000 10th European*, pages 1–4. IEEE, 2000.

[16] ISO/IEC. Iso/iec 9899 - programming languages – c. page 236, 1999.

[17] Raimund Kirner and Peter Puschner. Discussion of misconceptions about wcet analysis. In *WCET*, pages 61–64, 2003.

[18] Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *Proceedings of the 20th IEEE Real-Time Systems Symposium*, RTSS '99, pages 12–, Washington, DC, USA, 1999. IEEE Computer Society. ISBN 0-7695-0475-2.

[19] Peter Marwedel. *Embedded System Design.* Springer Publishing Company, Incorporated, 2011. ISBN 9400702582, 9789400702585.

[20] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2):115–128, 2000. ISSN 1573-1383. doi: 10.1023/A:1008119029962.

[21] John A. Stankovic. Misconceptions about real-time computing: A serious problem for next-generation systems. *Computer*, 21(10):10–19, October 1988. ISSN 0018-9162. doi: 10.1109/2.7053.

[22] H. Theiling. Extracting safe and precise control flow from binaries. In *Real-Time Computing Systems and Applications, 2000. Proceedings. Seventh International Conference on*, pages 23–30, 2000. doi: 10.1109/RTCSA.2000.896367.

[23] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008. ISSN 1539-9087. doi: 10.1145/1347375.1347389.