EDiFy: An Execution time Distribution Finder

Braams, B.; Altmeyer, S.; Pimentel, A.D.

DOI
10.1145/3061639.3062233

Publication date
2017

Document Version
Final published version

Published in
2017 54th ACM EDAC IEEE Design Automation Conference (DAC 2017)

EDiFy: An Execution time Distribution Finder

Boudewijn Braams
University of Amsterdam
bbx1992@gmail.com

Sebastian Altmeyer
University of Amsterdam
altmeyer@uva.nlm

Andy D. Pimentel
University of Amsterdam
a.d.pimentel@uva.nl

ABSTRACT

Embedded real-time systems are subjected to stringent timing constraints. Analysing their timing behaviour is therefore of great significance. So far, research on the timing behaviour of real-time systems has been primarily focused on finding out what happens in the worst-case (i.e., finding the worst case execution time, or WCET).

While a WCET estimate can be used to 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 paper, we present EDiFy, a measurement-based framework that derives execution time distributions by exhaustive evaluation of program inputs. We overcome the scalability and state-space explosion problem by i) using static analysis to reduce the input space and ii) using an anytime algorithm which allows deriving a precise approximation on the execution time distribution. We exemplify EDiFy on several benchmarks from the TACLeBench and EEMBC benchmark suites, and show that the anytime algorithm provides precise estimates already after a short time.

1. INTRODUCTION

Research on the timing behaviour of embedded real-time systems has been primarily focused on determining the worst-case execution time (WCET). This focus is clearly motivated by the need for timing verification, i.e, the need to guarantee at design time that all deadlines will be met. Figure 1 taken from the survey paper on WCET analyses [15] illustrates the simplification of this focus: It shows the fictitious execution time distribution of a real-time task, i.e the smallest individual software component within the system. A WCET analysis reduces the often complex timing-behaviour of a task to a single value. Speaking in terms of Figure 1, all values left of the WCET are ignored. Timing verification, in its traditional form, only requires bounds on the WCET of all tasks in the system. It assumes conservatively that the system operates correctly, only if it does so when all tasks run up to their WCET value. For many industries, this assumption is unnecessarily conservative and leads to costly over-provisioning of hardware resources. In fact, only very few real-time applications, mostly from the avionics industry, require a timing verification up to the highest standard. In most cases, infrequent deadline misses are acceptable and also preferable to excessive hardware costs. The state-of-the-art in timing analysis, however, does not provide the necessary means to derive richer information about the timing behaviour.

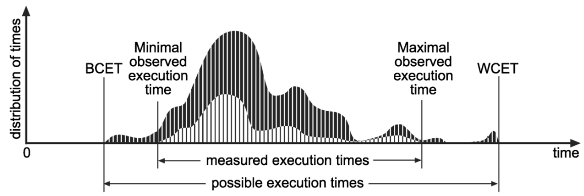

Figure 1: An execution time distribution, with annotated best-case execution time (BCET) and worst-case execution time (WCET), source [15] (modified)

In this paper, we close this gap and present *EDiFy*, a framework for the estimation of execution time distributions of embedded real-time tasks. 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 real-time embedded systems, and can answer questions such as: *is the worst-case a common or a rare case*, *what is the average execution time*, or, *what is the difference between best and worst-case execution time*.

Deriving execution time distributions is an even more complex task than bounding the WCET value, as it subsumes the former: A correct and complete execution time distribution would also contain the information about the WCET value. Consequently, we have to restrict the problem setting: First of all, we rely on measurements instead of static analyses. Relying on measurements implies that the resulting execution time distribution will never show the *full picture*, unless the input space allows for exhaustive measurements, which is highly unlikely. Secondly, *EDiFy* is task-centric: we assume that only the task under examination is running on the hardware. Analysing the timing behaviour of a complete task set and schedule is future work. Thirdly, we assume an input value probability distribution to be provided. Even though it may seem as a strong assumption, it is an absolute necessity. Not even the average execution time is well defined, if we do not know which inputs occur how often. The burden of providing these distributions is clearly on the system designer. We

DAC '17, June 18 - 22, 2017, Austin, TX, USA

ISBN 978-1-4503-4927-7/17/06. . . $15.00

DOI: http://dx.doi.org/10.1145/3061639.3062233

also consider this assumption feasible as sample data can be derived via test runs, control simulations and so on. Last but not least, we concentrate for now on control applications, instead of data-intensive image or video-processing benchmarks due to the size of the input data. With these restrictions in place, which we consider reasonable and realistic, the problem remains computationally intractable. EDiFy overcomes this obstacle by a combination of i) a static analysis to reduce the state-space, ii) a distributed anytime algorithm, and iii) an *evenly-distributed* state-space traversal that ensures quick convergence of the anytime algorithm.

We note that our work differs fundamentally from probabilistic timing analyses as currently advocated for real-time verification. We do not employ any statistical methods. Instead, EDiFy uses a heuristic to approximate the distribution. If executed for a sufficiently long time, EDiFy will eventually result in the *ground-truth* under the restriction detailed above, and assuming that the distribution of the distribution of the input values is provided. Hence, as a side-effect, the EDiFy framework enables us to evaluate the precision and correctness of measurement-based probabilistic timing analyses [4]. Yet, we are ultimately interested in providing a precise approximation on the execution-time distribution, and not in providing estimates on the WCET.

Related Work. Execution time analyses can be classified as static analyses or measurements-based analyses [15]. Static analyses are based on analysing program code and control flow and do not involve any actual execution of program code. *aiT* [11] and *Bound-T* [12] are examples of commercial static analysis tools used in the real-time embedded systems industry. However, these tools are designed for producing WCET estimates only, and are not suited for deriving execution time distributions. In 2004, David and Puaut [9] developed a static analysis to derive complete execution time distributions, but without considering any hardware effects, such as caching, branch-prediction or pipelining. Their approach consists purely of a source-code analysis, hence the resulting distribution can only be an abstract indication of the actual execution times on real hardware.

Measurement-based analyses, in contrast, extract timing behaviour by taking actual run-time measurements of execution times. This method is inherently simpler and merely requires the program code and/or binary and a means to execute it (either on real hardware or in a simulated environment). *RapiTime* [3] represents an example of a commercial measurement-based tool. As it is in general intractable to derive all measurements, measurement-based WCET analyses tend to steer the input values towards the worst-case. This is again in stark contrast to our approach, where we need to cover a wide range of input values. Recently, probabilistic timing analyses, especially measurement-based probabilistic timing analyses [4], have received ample attention in the real-time community. Despite arguing about execution time distributions in general, these approaches only serve to derive upper bounds on the execution and employ extreme-value theory [8] or copulas [5] to this end. As a consequence, these methods rely on strong assumptions about the probabilistic nature [13] of the hardware and input values, and foremost, they only derive cumulative distribution functions to assign exceedance probabilities to WCET estimates. To the best of our knowledge, no methods are available so far to derive complete execution time distributions on modern hardware.

Structure. The paper is structured as follows: Section 2 introduces the EDiFy framework, its inputs and outputs and the tools used. In Section 3 we detail the state-space pruning, and in Section 4, we detail the anytime algorithm. Section 5 provides an evaluation based on selected TACLeBench and EEMBC benchmarks, and Section 6 concludes the paper.

2. THE EDIFY FRAMEWORK

In this section, we explain the overall structure of the EDiFy framework and the required input and derived output. The input to the framework is the C-code of the program to be analysed, and the input value probability distributions of each input variable. The output is an approximation on the corresponding execution time distribution. We define an execution time distribution (ETD), as a probability distribution, which gives the likelihood of a certain execution time t occurring: $\mathrm{ETD}\colon \mathbb{N} \to \mathbb{R}$ with $\sum_t \mathrm{ETD}(t) = 1$. Such a distribution captures the complete timing behaviour of a system.

We note that for real-world applications of embedded real-time systems it can be assumed that input values are not uniformly distributed. Take for example a control system in a modern car where the engine temperature is an input variable. Initially, the engine temperature will be low, but after driving the car for a while the engine will remain warm. It is hence evident that the input value distribution (IVD) influences the ETD, and must therefore be taken into account. An IVD assigns each value of an input variable its likelihood: $\mathrm{IVD}\colon \mathbb{V}_T \to \mathbb{N}$ with $\sum_{v\in\mathbb{V}_\mathbb{T}} \mathrm{IVD}(v) = 1$ where $\mathbb{V}_\mathbb{T}$ is the set of values of a variable of type T. We require an IVD for each independent input variable, and a conditional probability distribution for a dependent variable. We note that dependency between variables do not change the complexity of the EDiFy framework, as the input probabilities are solely used to weight the measured execution times.

For the sake of simplicity, we only present the equations assuming independent variables. Further details on handling depending types can be found in [7].

2.1 Structure of the EDiFy Framework

The framework (see Figure 2) consists of two main components, a static part to prepare the input space, shown on the left side, and a dynamic part to run the measurements, shown on the right side. The input preparation is executed once, and performs a static program analysis to derive the set of variables that indeed influence the execution time of the task, and how these variables influence the execution time. The rational behind this step is to reduce the input space by pruning irrelevant input variables and variable ranges: not each input variable influences the execution behaviour, and not each input value leads to a distinct execution time. Section 3 provides the details on the input preparation. The measurements, i.e. the dynamic part of the EDiFy framework, are executed distributively by a fixed number of worker processes. Each process is assigned a dedicated range of the complete input space, and traverses this range until either each input value of the assigned range has been visited, or until the algorithm is aborted. In each iteration for each process, an input generator computes the next state of the input space to be visited, injects these values in the test-harness of the C-code provided by the user, and creates a stand-alone executable to be executed in the simulator. The result of each measurement is forwarded to the execution time distribution calculator, which weighs the measured execution times with the input distribution. The ETD calculator continuously updates the resulting ETD estimation. The algorithm can thus be aborted at any time, while still producing meaningful results. Section 4 provides the details on the anytime algorithm.

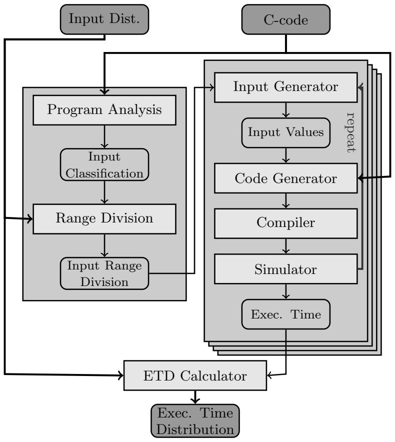

Figure 2: Toolchain of the EDiFy Framework. The input to the framework are the C-code, and the Input distributions for each input variable, the output is the corresponding execution time distribution. The left part of the toolchain (input preparation) is executed exactly once, whereas the right part (measurements) is executed iteratively and distributively.

2.2 Supported Input Types

For each supported type, we require a bijective function that maps the complete domain of the variable to the natural numbers $\mathbb{N}$, which enumerates all values of that type $E\colon \mathbb{T} \rightarrow \mathbb{N}$. For example, the enumeration function for Boolean values simply assigns 0, resp. 1, to the values *true* and *false*. The enumeration function for integer shifts the complete range by the minimum integer value, i.e., $E_{\text{int}}(x) = x + |\text{INT\_MIN}|$ to ensure that the enumeration function starts at 0, instead of a negative value. For arrays with n unique values, the *lexicographic order* is used as an enumeration function.

Built-in types such as *floats* and *doubles* can also be integrated: We can simply interpret the *bit-representation* of a *float* value as an *integer* value, and apply the enumeration function for in integers E_{int}. Compound types such as *structs*, or *arrays* are supported by interpreting each component as an independent input variables. Only domain specific knowledge must be encoded using a dedicated enumeration function.

We note that input variables can influence the execution time either by influencing the control flow directly, i.e., through conditionals, or *loop*-statements, or through instructions with variable execution times, such as *floating-point* or *memory operations*. While the EDiFy framework also supports variable instruction execution times by exhaustive evaluation of *pointer* or *floating point* values, dedicated support for this type of input-dependent execution time is future work.

2.3 Hardware State

The EDiFy Framework is task-centric, meaning that we assume no other tasks or code to be executed on the same hardware system. Consequently, there are only two hardware states that can occur, a *cold* system, where no data or instructions have yet been cached, or a *warm* system, where the cache has already been filled with data from the task under examination. Results for the first can be achieved by resetting the simulation after each measurement, and the second by executing the same task with the same input data twice, but only measuring the second iteration.

We acknowledge that this restriction is rather substantial. We do however believe that the information about the execution time distribution based on warm or cold system hardware states only, is already valuable on its own. The extension to other hardware states is considered future work.

2.4 Implementation details

The EDiFy framework is implemented using *Python* to control the tool chain and the anytime algorithm. The static program analysis is implemented within the CIL framework [14]. As target architecture, we have selected the *ARMv8*, for which a cycle accurate simulator *(gem5)* [6] and a *gcc* cross-compiler are freely available. The implementation of the framework is made available online [2].

3. INPUT-SPACE ANALYSIS

The main obstacle to overcome is the prohibitively large number of input variations. The input space is simply too large to naively derive an execution time measurement for each element in the input space. Our first goal is thus to remove superfluous input values and to cluster input ranges for which we can guarantee that the execution time values will be the same. The questions we need to ask here are:

- Which input parameters influence the execution times?
- How do they influence the execution times?

Static program analysis is the natural way to provide safe and complete answers to these question. The input analysis is implemented as a backwards program analysis that derives the set of variables that influence the execution time either *directly* or *indirectly*. With *directly*, we mean that the variables appear in the expression within an *if*-statement or *loop*-statement (or within a *float* or *pointer* operation, in case of variable instruction times) and with *indirectly*, we refer to variables that only influence variables from the first set.

3.1 Program Analysis

In the following, we describe the basic program analysis, which derives the set of all variables that influence the control-flow of the program. All other program analyses are derived from this basic analysis using minor modifications.

The domain of the analysis is the powerset of the set of variables $\mathbb{V}$: $D = 2^{\mathbb{V}}$ with $\emptyset$ being the bottom and $\mathbb{V}$ the top element. Since we are interested in a safe analysis, we use set-union $\bigcup$ as the combine-operator to be invoked in case of control-flow merges. The auxiliary function $\text{varUsed}\colon \mathit{Expr} \rightarrow 2^{\mathbb{V}}$ derives the set of variables used within an expression. The transfer function $\text{tf}\colon \mathit{Instr} \rightarrow (2^{\mathbb{V}} \rightarrow 2^{\mathbb{V}})$ selects all variables used within an expression in an *if* or *loop* statement, and also all variables used within an expression if the result of the expression is assigned to an execution time influencing variable. It is defined as follows:

$$
\begin{aligned}
&\text{tf}(I)(V) = \mathit{match}\ I\ \text{with} \\
&\qquad \mathit{if}\ (\mathit{exp})\ \_ \qquad \rightarrow V \bigcup \text{varUsed}(\mathit{exp}) \\
&\qquad \mathit{while}\ (\mathit{exp})\ \_ \quad \rightarrow V \bigcup \text{varUsed}(\mathit{exp}) \\
&\qquad v = \mathit{exp} \qquad\quad\ \rightarrow \text{if}(v \in V)\ \text{then} \\
&\qquad\qquad\qquad\qquad\quad V \bigcup \text{varUsed}(\mathit{exp})\ \text{else}\ V \\
&\qquad \_ \qquad\qquad\qquad \rightarrow V
\end{aligned}
\tag{1}
$$

where I is an instruction and V is the set of input-influencing variables. We assume for the sake of simplicity a simplified instruction set where all loops have been transformed to *while* loops, as within the CIL framework [14], in which we have implemented the program analysis.

The analysis can be modified to cover instructions with variable execution times. The analysis iterates over all expressions within a program, whenever the analysis encounters an expression of type *float*, or an expression used to index a memory address, respectively, all variables used within the expression are added to the current data-flow value.

The presented analysis derives all directly and indirectly influencing variables. To derive directly influencing variables only, we simply have to omit the case distinction $v = exp$ and directly forward the data flow value V without any additions.

We acknowledge that further program analyses, such as a *value*- or a *pointer*-analysis can be integrated to further reduce the input-space. These analyses, however, exceed the scope of the paper. The main purpose of the presented program analysis is to correctly classify all input variables and to ensure completeness, i.e., to ensure that each input-influencing variable is correctly identified. Bounds on the minimal or maximal values of variables, or additional information about the input variables can be provided by the user.

3.2 Classification

The result of these program analyses is a classification of the input variables along two orthogonal lines: directly or indirectly influencing, and through loops, conditionals or variable instruction times. This classification is a prerequisite to divide the input space in a meaningful manner. We note that this classification is not exclusive, i.e., a variable may influence the execution times in more than only one single category. Furthermore, as an implicit result of this classification, we can validate whether the user has specified an input distribution for all relevant variables, and we can omit irrelevant variables from further examination. We denote the set of execution time influencing variables with $\mathbb{V}_I$.

3.3 Handling Multiple Variables

In case of multiple variables, we project the multi-dimensional input space to the natural numbers $\mathbb{N}$ using $\hat{E}\colon \mathbb{V}_1 \times \mathbb{V}_2 \times \ldots \mathbb{V}_n \to \mathbb{N}$ with

$$\hat{E}(v_1, v_2, \ldots, v_l) = \sum_{i=1}^{l} \left(\left(\prod_{j<i} E_{V_j}^{\max} \right) * E_{T_i}(v_i) \right) \tag{2}$$

where E_{T_i} is the enumeration function for Type T_i and $E_{V_j}^{\max}$ denotes the size of the domain of variable V_j.

Similarly, we compute the input probability for the tuple of input variables $(v_1, v_2, \ldots, v_l)$ assuming that all variables are independent as follows:

$$\mathrm{IVD}'(v_1, v_2, \ldots, v_l) = \prod_{j<i} \mathrm{IVD}(v_i) \tag{3}$$

Dependent variables have to be handled and defined explicitly, see [7] for further details.

3.4 Input Range Division

The measurements will be distributed to different processes so that we can exploit the parallelism of modern architectures. To this end, we evenly distribute the entire input space to all spawned processes used by the anytime algorithm.

4. ANYTIME ALGORITHM

The elimination of non-relevant input values is unlikely to reduce the input space sufficiently for an exhaustive evaluation. In most cases, approximation is inevitable.

In this section, we detail the anytime algorithm. In particular, we describe how the input ranges assigned to each processor are traversed to achieve an even coverage, and we describe how the resulting measurements are weighted by their corresponding input value distribution.

The anytime algorithm works by spawning various worker processes to perform the measurements, and an additional process which continuously accumulates and processes the execution times produced by the worker processes. This provides immediate availability of the latest results and thereby allows for the execution time distribution to be derived on the fly.

To derive a meaningful approximation of the execution time distribution early on, we divide the input space over several processes, and employ a specific traversal function. Our assumption is that the execution time distribution can be approximated quickly by evaluating the input space *evenly*.

4.1 Range Traversal

We have to ensure that we traverse the input space, or to be specific, the range of the input space assigned to a worker process, in a meaningful way. When we start to traverse the range from one corner and move to the other step by step, we achieve a full coverage of a part of the range, whereas the other side remains unvisited until the entire range has been visited. We refer to this type of state traversal as *linear traversal*.

To cover the entire input space evenly already early on we propose an alternative traversal function tr. The rational behind this function is to always hit the middle of the unvisited space. Assume an input range given by $[0 : 127]$. The traversal function starts in the middle of the range, $\mathrm{tr}(1) = 63$, followed by the middle of the left sub-range $[0 : 63]$, $\mathrm{tr}(2) = 32$, and of the right subrange $[63 : 127]$, $\mathrm{tr}(3) = 96$, and so on. We refer to this traversal function as *logarithmic traversal*. We first define an auxiliary function $\mathrm{tr}'\colon \mathbb{N} \to (0 : 1)$ which computes the range pointer within the range $(0 : 1)$, e.g., $tr'(1) = 0.5$, $tr'(2) = 0.25$, $tr'(3) = 0.75$, irrespective of the size of the range of tr. It is defined as follows:

$$\mathrm{tr}'(x) = \left(\frac{1}{2^{\lfloor log_2(x) \rfloor + 1}} + \frac{x - 2^{\lfloor log_2(x) \rfloor}}{2^{\lfloor log_2(x) \rfloor}} \right), \tag{4}$$

Since the function tr' always cuts the unvisited ranges in half, it works best for range-size of a power of 2. Next, we have to map the range of tr' to an arbitrary range $[l_{\min} : l_{\max}]$. Let s be the size of the range, i.e., $s = l_{\max} - l_{\min} + 1$. We define $\mathrm{tr}\colon \mathbb{N} \to \mathbb{N} \cup \{\perp\}$, the corrected version of tr' as follows:

$$\mathrm{tr}(x) = \begin{cases} l_{\min} + \mathrm{tr}'(x) * 2^{\lceil log_2(s) \rceil} & \text{if } \mathrm{tr}'(x) * 2^{\lceil log_2(s) \rceil} \leq s \\ \perp & \text{otherwise} \end{cases} \tag{5}$$

The value $\perp$ indicates that the result will be omitted and we directly proceed with the next range index. This is necessary to ensure that each value occurs exactly once, and to avoid performing measurements with the same input values twice.

The logarithmic traversal function tr is applied to each worker process, and hence, to each subrange individually. A weakness of tr is that it visits $l_{\min}$ and $l_{\max}$ very late, resp. at the very last. The values $l_{\min}$ and $l_{\max}$ tend to result in the lowest and highest execution time values and hence, determine the overall shape of the distribution more than values from the middle range. To overcome this drawback, $l_{\min}$ and $l_{\max}$ will be visited first within each process and only after these two measurements, the traversal using tr starts.

4.2 Derivation of execution time distribution

The measured execution times are stored in a relative frequency table. This table contains an entry for each observed execution time with a value indicating its relative occurrence in relation to all others: $\mathrm{rf}\colon \mathbb{N} \to \mathbb{R}$ with $\sum_{\forall t \in \mathbb{N}} \mathrm{rf}(t) = 1$.

As we assume the availability of the value probability distributions for each of the input variables, we have to include these in the derivation of the execution time distribution. We do this by utilising the probability functions as weight functions for the frequency table. The relative frequency of a measured execution time t is determined by the following update function: $\mathrm{rf}(t) := \mathrm{rf}(t) + \prod_{i=1}^{n} P(I_i = v_i)$. This ensures that an execution time resulting from a high probability input contributes more to the distribution than one with a low probability input. Note that we assume the probabilities of the individual input variables to be statistically independent (i.e., that the joint probability is given by the product of the individual probabilities).

The final step in deriving the execution time probability distribution is to normalise the data by dividing each value by the sum of all values. This last step ensures that all the combined probabilities add up to 1.

5. EVALUATION AND RESULTS

In this section, we exemplify the EDiFy framework on a selection of benchmarks from the *TACLeBench* [10] and the *EEMBC* [1] benchmark suites. TACLeBench is an open source benchmark suite particularly designed for the evaluation of timing analysis tools, whereas EEMBC is a commercial benchmark suite based on realistic automotive use cases. Despite the high number of available benchmarks, only a subset exhibits non-trivial timing behaviour, or an input-dependent execution time. Furthermore, in nearly all cases, a single variable per task influences the execution time behaviour. We have selected three non-trivial benchmarks to highlight different aspects of the EDiFy Framework: *bubble-sort* (from TACLeBench) has been selected to illustrate the progress of the anytime algorithm over time, and *bitmnp* and *pntrch* (both EEMBC) to show specific execution time distributions and their dependency on the input value distributions. Due to space constraints, further results are only available online [2].

The EDiFy framework was run on a system featuring a quad-core *Intel Core i7-4700MQ* processor clocked at 3.4GHz with 16GB of DDR3 RAM. The benchmarks were executed in the *gem5* cycle-accurate simulator and cross-compiled using *GCC 5.3.0*. The simulator itself targeted the 64-bit *ARMv8-A* architecture, a clock speed of 500Mhz and a cache featuring 128kB of L2 cache, 64kB of L1 data cache and 16kB of L1 instruction cache.

5.1 Anytime Algorithm

For the evaluation of the anytime algorithm, we have chosen the *bubble-sort* benchmark, as it exhibits non-trivial timing behaviour and is easily scalable. Due to the specific purpose as a benchmark, there is only one parameter which has been correctly identified by the input space analysis. We have assumed equal probability of each permutation, and use the

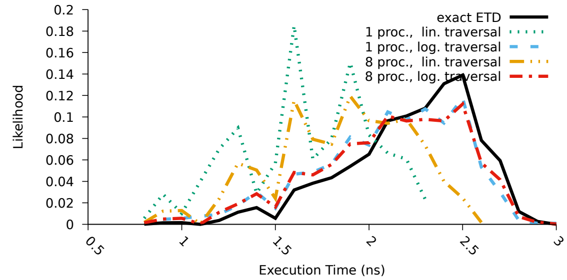

Figure 3: Approximation on the execution time distribution for benchmark *bubble sort* (TACLeBench) after 10 minutes of runtime.

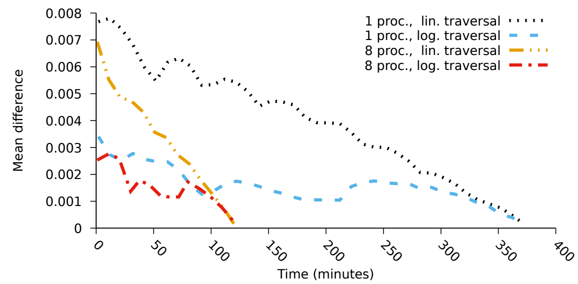

Figure 4: Progression of the *weighted mean* (with respect to the result obtained by exhaustive evaluation) of the execution time distributions for benchmark *bubble sort* (TACLeBench).

lexicographic order as bijective enumeration function, i.e., to assign each value from $[0 : n! - 1]$ a unique permutation. Figure 3 depicts the execution time distributions derived after 10 minutes using 4 different configurations: linear and logarithmic traversal executed on 1 (with one spawned process only) or on 4 processors (with 8 spawned processes). The anytime algorithm completes around 1000 measurements per processor within 10 minutes. In addition, we have added a line that shows the final, and hence exact execution time distribution after exhaustive evaluation. The measured execution times have been rounded to the closest 0.1ns to smooth the graph.

The logarithmic traversal function leads to a precise approximation after already 10 minutes, irrespective of the number of processors used, whereas, the linear traversal function shows an heavily skewed approximation on the execution time distribution, which is only slightly alleviated by using 4 processors. We have also evaluated the *mean difference* with respect to the exact distribution (see Figure 4). Exhaustive evaluation is achieved after around 120, resp. 360 minutes when executed on 4, resp. 1 processor. The graph shows the advantage of combining logarithmic traversal with distributed processing. The logarithmic traversal function ensures a tight approximation early on, irrespective of the number of processors, and the distributed processing reduces the overall runtime resulting in faster convergence. Interestingly, the *mean differences* are not monotonically decreasing for *bubble-sort* as some costly permutations are only examined towards the end of the evaluation. We note that for cases with purely integer input variables, we observe monotonic *mean differences*.

5.2 IVD-Dependency

The other two benchmarks, *bitmnp* and *pntrch* have been selected as they depict rather peculiar execution time distributions. Both of these benchmarks stem from the EEMBC automotive benchmark suite [1]. We use these benchmarks to

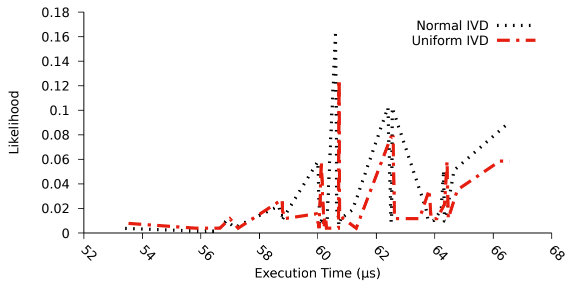

Figure 5: Execution time distribution for benchmark *bitmnp* (EEMBC)

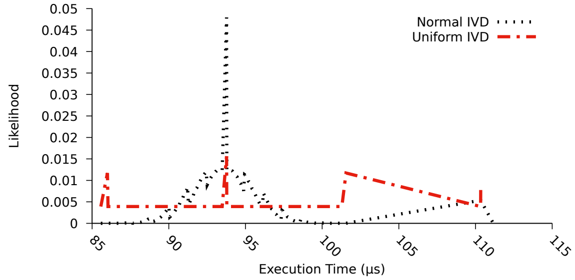

Figure 6: Execution time distribution for benchmark *pntrch* (EEMBC)

illustrate the impact of the input value distribution, instead of illustrating the progress of the anytime algorithm: for these benchmarks, the anytime algorithm has produced very precise estimates already after less than 10 minutes.

First, we determined the input values using the input space analysis. From the initial set of 10 (*bitmnp*), resp. 11 (*pntrch*) global input values, 7, resp. 8, have been identified of being irrelevant to the timing behaviour by the input-space analysis, and 2 more have been eliminated as static part of the test harness. In each case, the remaining input-influencing parameter is of type *integer*. We have observed this strong reduction in nearly each benchmark of the EEMBC benchmark suite [1]. Figures 5 and 6 depict the execution time distributions for the two benchmarks, assuming either a *uniform*, or a *normal distribution* over the entire integer range. Due to space constraints, we omit the graphical representation of the input value distribution. The interested reader is referred to [2], where the complete data is available. In both cases, we can clearly see the impact of the IVD on the execution time distribution. Furthermore, we observe non-trivial distributions, with very peculiar features such as the peak in case of *pntrch* with normal distribution, or the high likelihood of the WCET in case of *bitmp*. Such information can be of high interest in the design and evaluation of the timing behaviour of real-time tasks. Even though these benchmarks are rather small, compared to other domains, they still represent typical examples from the automotive domain.

6. CONCLUSION

In this paper, we have presented EDiFy, a framework to derive the execution time distributions of embedded real-time tasks. EDiFy lifts real-time timing analysis from deriving bounds on the execution time to deriving complete execution time distributions.

The main obstacle towards deriving such execution time distributions is the computational complexity and the sheer size of the input space. We attack this state-space explosion problem by i) using static analysis to reduce the input space and ii) using an anytime algorithm which allows to derive meaningful approximation on the execution time distribution. The static analysis removes irrelevant input parameters, and hence prunes the state-space. The anytime algorithm – together with a logarithmic traversal function to achieve a balanced coverage of the input space – allows to compute a precise approximation even if exhaustive evaluation is infeasible. We have successfully exemplified the EDiFy framework on TACLeBench and EEMBC control applications.

Our framework is currently task-centric, meaning that we assume that only the task under examination is running on the hardware. As future work, we plan to extend the framework towards complete task sets, where we take the interference of different tasks on the hardware into account. Furthermore, we plan to integrate more sophisticated program analyses to further prune the input-space.

References

[1] EEMBC Autobench. http://www.eembc.org/benchmark/automotive_sl.php. Accessed: 2016-04-29.

[2] Git repository of EDIfY: The execution time distribution finder. https://github.com/BDWN/etd.

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

[4] G. Bernat, A. Burns, and M. Newby. Probabilistic timing analysis: An approach using copulas. *J. Embedded Comput.*, 1(2):179–194, 2005.

[5] G. Bernat, A. Colin, and S. M. Petters. Wcet analysis of probabilistic hard real-time systems. In *RTSS '02*, pages 279–288.

[6] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.

[7] B. Braams. Deriving an execution time distribution by exhaustive evaluation. Bachelor's thesis, University of Amsterdam, June 2016.

[8] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. QuiÃśones, and F. J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS '12*, pages 91–101.

[9] L. David and I. Puaut. Static determination of probabilistic execution times. In *ECRTS '04*, pages 223–230.

[10] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. SÃÿrensen, P. WÃd'gemann, and S. Wegener. Taclebench: A benchmark collection to support worst-case execution time research. In *WCET '16*.

[11] R. Heckmann and C. Ferdinand. Worst-case execution time prediction by static program analysis. In *IPDPS '04*, pages 26–30.

[12] N. Holsti, T. Langbacka, and S. Saarinen. Worst-case execution time analysis for digital signal processors. In *EUSIPCO '00*, pages 1–4.

[13] Y. A. Li, J. K. Antonio, H. J. Siegel, M. Tan, and D. W. Watson. Determining the execution time distribution for a data parallel program in a heterogeneous computing environment. *J. Parallel Distrib. Comput.*, 44(1):35–52, 1997.

[14] G. C. Necula, S. McPeak, S. P. Rahul, and W. Weimer. Cil: Intermediate language and tools for analysis and transformation of c programs. In *CC*, pages 213–228, 2002.

[15] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem-overview of methods and survey of tools. *ACM TECS*, 7(3):36:1–36:53, May 2008.