Iterative Histogram-Based Performance Analysis of Embedded Systems

Boris Dreyer, Christian Hochberger, Thomas Ballenthin, and Simon Wegener

***Abstract*—Precise execution time profiles (ETPs) are an extremely helpful instrument to assist software and system designers in analyzing the performance and timing behavior of embedded systems. Previously, we have presented an approach that exploits embedded trace units of modern system-on-chip. It allows us to compute execution time histograms during the runtime of the system under test (SuT). These histograms can easily be converted into ETPs. In this contribution, we show an extended version of this method that uses the information gathered in previous runs of the SuT to refine the binning used for the collection of the histograms. We show that these improved histograms deliver much more insight.**

***Index Terms*—Code profiling, embedded software, field programmable gate array (FPGA), reconfigurable computing, worst-case execution time (WCET).**

I. Introduction

Today, embedded systems are a central part of almost all technical systems. System-on-chip (SoC) architectures are the predominant way to implement embedded systems. They offer high performance and are flexible through their software programmability. In safety-critical systems, proper function does not only rely on correct internal sequence of operations but also, on the timing of the operations. One way to obtain measurements of the execution time of the system under test (SuT) running on modern SoCs is the use of embedded trace units (ETUs).

During the runtime of the SuT, the ETU emits various trace messages over a trace port that inform a debugger about the current state of the system. In our case, the ETU implements the ARM Coresight specification [1]. There, a trace message is emitted for every nonlinear control-flow, for example, interrupts and hardware exceptions, but also normal calls and branches. These messages carry the address where the control flow change happened and the target of the change. Since the trace messages also contain timestamps, it is possible to compute the execution time between two control-flow changes.

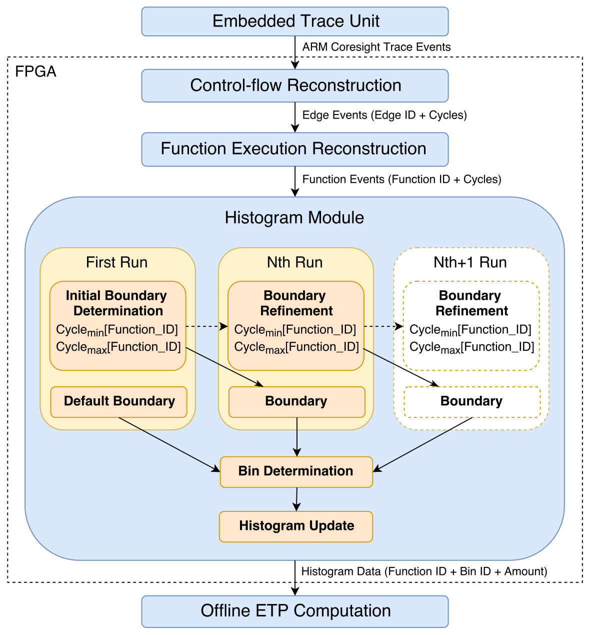

Fig. 1. Workflow of our FPGA-based analysis platform.

In previous work [2], we proposed an field programmable gate array (FPGA)-based approach for the performance analysis of embedded systems, exploiting the event stream emitted by the ETUs of modern SoC. First, in a preprocessing phase, we reconstruct the control-flow graph (CFG) of the SuT. A CFG consists of basic blocks—sequences of instructions, where each instruction except the first and the last has exactly one predecessor and one successor—and edges that describe the flow of control in a program, i.e., conditionals, routine calls, loops, etc. The CFG is used to configure our FPGA-based analysis platform, allowing to map trace messages to functions in the SuT.

Then, during the runtime of the SuT, the trace messages emitted by the ETU are fed into our analysis platform, see Fig. 1. There, the trace messages are decoded to reconstruct the dynamic control-flow of the SuT. We accumulate the messages in our system to produce a stream of function events, each with an associated execution time. This function event stream is fed into the histogram module of our analysis platform. There, the events are further aggregated to execution time histograms for each function in the SuT. One such histogram is depicted in

Manuscript received January 30, 2018; revised March 19, 2018 and June 18, 2018; accepted July 10, 2018. Date of publication October 16, 2018; date of current version May 27, 2019. This work was supported by the German Federal Ministry for Education and Research through the Project CONIRAS under Grant 01IS13029. This manuscript was recommended for publication by A. Easwaran. *(Corresponding author: Boris Dreyer.)*

B. Dreyer and C. Hochberger are with the Computer Systems Group, TU Darmstadt, 64289 Darmstadt, Germany (e-mail: dreyer@rs.tu-darmstadt.de; hochberger@rs.tu-darmstadt.de).

T. Ballenthin is with HBM GmbH, 64293 Darmstadt, Germany (e-mail: thomas.ballenthin@hbm.com).

S. Wegener is with AbsInt Angewandte Informatik GmbH, 66123 Saarbrücken, Germany (e-mail: wegener@absint.com).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/LES.2018.2876561

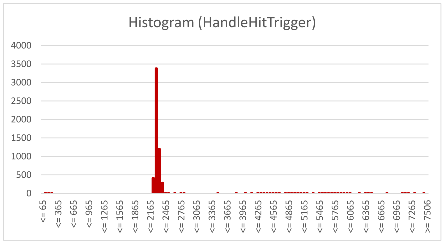

Fig. 2. Histogram of one task of the debie1 benchmark. Configuration: iteratively refined linear bin distribution, $k = 128$ bins.

Fig. 2. It shows the distribution of observed execution times for one interrupt service routine (ISR).

In a post-processing step, these histograms are used to compute execution time profiles (ETPs) in complementary cumulative representation (see Fig. 3), allowing to assess its performance. Each ETP describes the probability that the execution time of a task or ISR is greater than a given threshold. Moreover, the resulting ETPs can be used to schedule tasks while maintaining a given quality of service level [3]; to estimate the execution times of programs and tasks [4]; or to compute the response time distribution of each task in a system [5].

The major challenge for our approach is the high frequency of the trace message stream of 200 MHz. Since we do not store the message stream for offline processing, we need to handle one event per cycle in order to continuously measure execution times. Thus, the distribution of events into the histograms must be predefined in some way. In our previous work [2], we presented two different modes to aggregate the histograms as follows.

1) In the first mode, it uses a linear bin distribution, where k bins, each *step* cycles wide, are used to cover the range $[0, k \times \mathit{step}[$. By construction, histograms using a linear bin distribution can only track events well with an execution time of less than $k \times \mathit{step}$ cycles. All events with a higher execution time are accumulated in the last bin.
2) The second mode uses the centered bin distribution. There, $k = k_l + k_u$ bins are centered around some reference value r, where k_l is the number of bins below r and k_u is the number of bins above r. Each of the bins is $\mathit{step} = \lfloor r/k \rfloor \times f$ cycles wide. The factor f is used to adjust the width of the bins. The linear bin distribution is used as fallback if $\mathit{step} = 0$. The whole histogram thus covers the range $]r - k_l \times \mathit{step}, r + k_u \times \mathit{step}[$. All events with lower or higher execution time are accumulated in the outer bins. For the centered bin distribution, it is thus important that the used reference value fits the real distribution of execution times.

In both modes, our histogram module needs some *a priori* knowledge of the performance in order to compute meaningful histograms, i.e., histograms where the majority of the events is not aggregated in the outer bins. For the linear bin distribution, some knowledge about the worst-case behavior is needed to properly configure *step*. For the centered bin distribution, some knowledge about the average-case behavior is needed to properly configure r, and some knowledge about the best-case and worst-case behavior is needed for f.

For the evaluation in [2], we used the first observed execution time of a function as value for r, and took educated guesses for the other parameters. The results showed that histograms were less than optimal, either because we underestimated the worst-case behavior, or because the first observed execution of a function was not a good representative of its average behavior.

II. Iterative Histogram Refinement

To address the problem of under- and overflows we designed an iterative process to improve the quality of the histograms over the process of multiple program runs. During the first execution we apply a linear bin distribution with a fixed parameter *step*. Furthermore, we record the minimal and maximal cycle that occurred per function: $\mathit{cycle}_{\min}\mathit{[function\_id]}$ and $\mathit{cycle}_{\max}\mathit{[function\_id]}$. For all subsequent runs we have a more detailed understanding of the previously observed cycle spread. Given this information we are able to calculate the optimal step value individually for every *function_id*, as shown in (1). Distributing $k-2$ bins between $c_{\min}$ and $c_{\max}$ provides the possibility to have extra bins for observed execution times that are smaller or larger than the preset boundaries

$$\begin{aligned} c_{\min} &= \mathit{cycle}_{\min}\mathit{[function\_id]} \\ c_{\max} &= \mathit{cycle}_{\max}\mathit{[function\_id]} \\ \mathit{step} &= \left\lceil \frac{c_{\max} - c_{\min}}{k-2} \right\rceil. \end{aligned} \tag{1}$$

For every bin j, the lower and upper boundaries x_j^l and x_j^u are calculated by applying a linear bin distribution, with $x_0^l = 0$, $x_{j+1}^l = x_j^u + 1$, and x_j^u as defined in

$$x_j^u = \begin{cases} \max(c_{\min} - \mathit{1}, \mathit{1}), & j = 0 \\ x_{j-1}^u + \mathit{step}, & 1 \le j \le k-2 \\ \infty, & j = k-1. \end{cases} \tag{2}$$

In case the execution times per *function_id* are fluctuating strongly between independent runs, a refinement can be achieved by repeatedly performing the histogram analysis, widening the $\mathit{cycle}_{\min/\max}$-ranges, until a satisfying distribution is achieved. Even for single runs, recording $\mathit{cycle}_{\min/\max}$-values provides a significant benefit: Whenever function events are sorted in the first or last bin, the $\mathit{cycle}_{\min/\max}$-values provide absolute lower/upper boundaries with the constraint that these bins are not equidistant.

III. Experimental Setup and Results

The *debie1 benchmark* [6], [7] is based on the on-board software of the DEBIE-1 satellite instrument for measuring impacts of small space debris or micro-meteoroids. It defines six analysis problem sets, each derived from the original real-time requirements of the satellite instrument.

Our target SoC was a Xilinx Zynq XC7Z020 featuring a dual-core ARM Cortex-A9 running at 667 MHz. The memory subsystem of the SoC consists of separate L1 instruction and data caches, each storing 32 kilobytes,

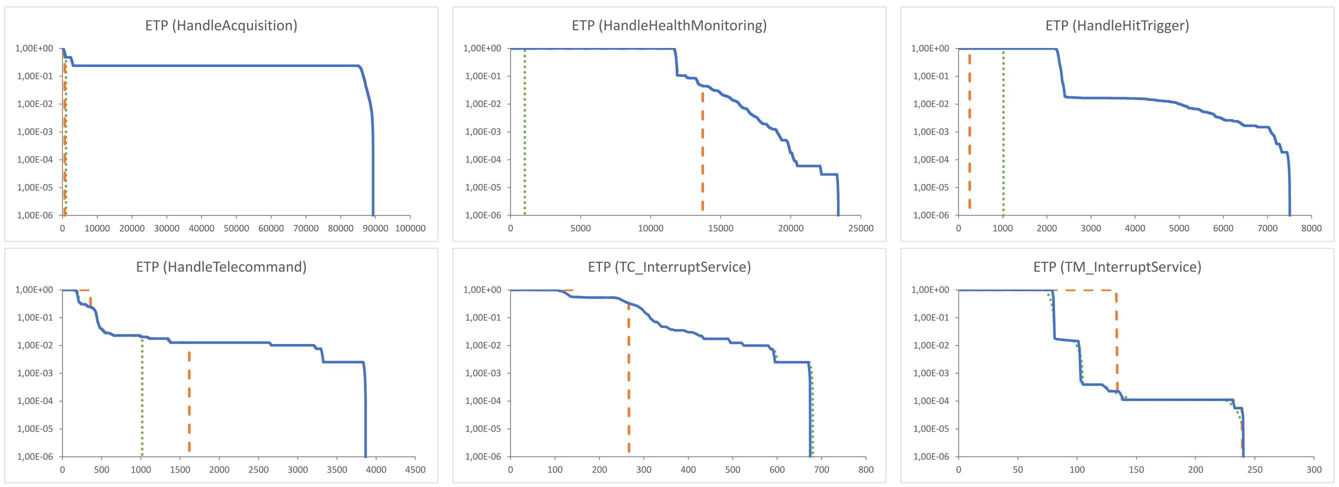

Fig. 3. ETPs of the four tasks and two ISRs of the debie1 benchmark. Configuration of blue solid lines: iteratively refined linear bin distribution, $k = 128$ bins. Configuration of green dotted lines: linear distribution, $k = 128$ bins and $step = 8$. Configuration of orange dashed lines: centered distribution, $k = 128$ bins, $k_l = 100$, and $f = 1$.

TABLE I
NUMBER OF FUNCTION EVENTS AS WELL AS MINIMAL AND MAXIMAL OBSERVED EXECUTION TIMES OF THE DEBIE1 BENCHMARK

Name	#Events	Min.	Max.
HandleAcquisition	5264	236	89259
HandleHealthMonitoring	33864	10670	23395
HandleHitTrigger	5382	66	7504
HandleTelecommand	394	86	3863
TC_InterruptService	401	50	673
TM_InterruptService	17795	77	239

512 kilobytes of shared L2 cache, and 1 gigabyte of DDR main memory. We compiled the debie1 benchmark with the C++ compiler provided with the Xilinx SDK 2016.1, `GNU C/C++ 4.9.2 20140904 (prerelease)`. The benchmark was then executed on one of the two cores, while the second core executed a program that was used to generate interferences on the shared L2 cache and the shared interconnects.

For our evaluation, we measured the execution times of the four tasks and two ISRs of the debie1 benchmark. Table I shows the number of observed function events for each of them. Moreover, it shows the minimal and maximal observed execution times of the tasks and ISRs. The ISR `TM_InterruptService`, for example, got called 17 795 times during the execution of the benchmark, with a maximal observed execution time of 239 cycles.

Fig. 3 depicts the ETPs obtained with our improved histogram module (blue solid lines). Since we have a recording of the whole event stream, we were able to replay the stream for the iterative approach. The figure also shows the ETPs constructed with the linear (dotted green line) and centered (dashed orange line) distributions.

We compared the coverage of histograms obtained with the linear, centered, and iterative distribution, see Tables II and III. The two ISRs `TC_InterruptService` and `TM_InterruptService` have short execution times and are thus fully covered with the linear distribution. However, the tasks have longer execution times, resulting in only partial coverage (between 0 % and 25 %).

TABLE II
ABSOLUTE RANGES COVERED BY DIFFERENT DISTRIBUTIONS. NUMBERS FOR LINEAR AND CENTERED DISTRIBUTION ARE TAKEN FROM [2]

Name	Linear	Centered	Iterative
HandleAcquisition	[0..1016]	[164..667]	[236..89318]
HandleHealthMonitoring	[0..1016]	[2628..13715]	[10670..23396]
HandleHitTrigger	[0..1016]	[126..251]	[66..7626]
HandleTelecommand	[0..1016]	[358..1617]	[86..3866]
TC_InterruptService	[0..1016]	[140..265]	[50..680]
TM_InterruptService	[0..1016]	[134..239]	[77..329]

TABLE III
RELATIVE RANGES COVERED BY DIFFERENT DISTRIBUTIONS, COMPARING THE ABSOLUTE RANGES FROM TABLE II WITH THE MEASURED MINIMAL AND MAXIMAL EXECUTION TIMES FROM TABLE I

Name	Linear	Centered	Iterative
HandleAcquisition	<1 %	<1 %	100 %
HandleHealthMonitoring	0 %	24 %	100 %
HandleHitTrigger	13 %	2 %	100 %
HandleTelecommand	25 %	33 %	100 %
TC_InterruptService	100 %	20 %	100 %
TM_InterruptService	100 %	65 %	100 %

The quality of the centered distribution heavily depends on the chosen reference value r and the bin in which it is sorted in. To be able to compare our approaches, we used the same configuration as in [2], i.e., the first observed execution time of a function as r, $k = 128$, $k_l = 100$, and $f = 1$. While the centered distribution was an improvement over the linear distribution for `HandleHealthMonitoring`, where the coverage increased from 0 % to 24 %, it was also degradation for `TM_InterruptService`, where the coverage decreased from 100 % to 20 %.

When comparing the coverage of our refined iterative approach with the linear and centered distributions, we see that the iterative approach is a vast improvement. Our refined approach covers 100 % of all interrupt routines and tasks. Thus, no event is sorted in the first or last bin of the iteratively refined linear distribution, i.e., no information is lost. For example, we see that for `HandleHitTrigger`, some executions of this task took about 7500 cycles, but a huge cluster of events has an associated execution time between 2000 and 2500 cycles. According to the measurements, the worst-case and average-case behavior of `HandleHitTrigger` thus varies quite strong. Consequently, a single end-to-end measurement of the task would not reflect both the worst-case and the average performance. Moreover, given that the ETP shows that the probability of hitting the worst-case is only about 1 in 10 000, it would be rather unlikely that the worst-case is observed in a single end-to-end measurement.

IV. Related Work

Bernat *et al.* [8] introduced the concept of execution profiles to describe the statistical properties of a section of code. Execution time profiles are based on execution time histograms and can be used to describe the timing behavior of software.

In [9] a hybrid measurement-based worst-case execution time (WCET) analysis concept is presented for constructing an instrumentation point graph (IPG) from a set of timing traces. They annotate the IPG with timing and loop bound information and use integer linear programming to calculate a WCET estimate.

Apart from ETPs, hardware implemented histograms are often used in image and video processing. Urimi *et al.* [10] used them for adaptive histogram equalization. In [11] and [12], they are part of the histogram of oriented gradients algorithm for object detection. Yang *et al.* [13] presented a sliced integral histogram algorithm for efficient histogram computation. Stekas and van den Heuvel [14] used local binary patterns histogram for face recognition.

Previous approaches to obtain meaningful ETPs by recording CPU trace data for offline processing are facing the challenge to temporarily store the trace data, which does not scale well. ETPs can also be generated by instrumenting the program code. This in turn leads to the probe effect and the instrumented code shows different timing behavior. Díaz *et al.* [15] reduced the instrumentation overhead, but still have an increase in code size of 7 % to 14 %.

In contrast to these implementations we create histograms of function execution events. These events occur at high frequency. In order to process them continuously in real-time, we implemented our histogram module in hardware, exploiting the ETU of a modern SoC. Thus, we do not need to instrument the software of the SuT. This prevents the probe effect and allows us to create authentic histograms. All histograms created with our platform have the same number of bins. This number is fixed and depends on the available memory. However, each histogram has its individual bin width that is resized for each measurement run. This allows to steadily increase the coverage of function execution times, up until full coverage is reached.

V. Conclusion

In this contribution, we have shown that almost perfect ETPs can be acquired, if we apply an iterative method. In a first run of the SuT, we identify the span of the required bins. This can be done with our FPGA implementation of the trace event stream analyzer. In subsequent runs, we can then process the incoming trace events at full speed in our FPGA histogram implementation. Thus, no recording of the trace stream is necessary and we can observe the SuT arbitrary long. The histograms that we acquire in subsequent runs converge to 100% coverage of the execution times. Thus, the ETPs computed from these histograms show a maximum number of details. The long observation time that we can afford through the online analysis simplifies the test setup and makes it easier to steer the system into the required measurement situation.

Acknowledgment

The authors would like to thank A. Lange and A. Weiss of Accemic for providing the edge event stream.

References

[1] *CoreSight Program Flow Trace PFTv1.0 and PFTv1.1 Architecture Specification*, ARM Ltd., Cambridge, U.K., 2011.

[2] T. Ballenthin, B. Dreyer, C. Hochberger, and S. Wegener, "Hardware support for histogram-based performance analysis of embedded systems," in *Proc. IEEE 20th Int. Symp. Real Time Distrib. Comput. (ISORC)*, 2017, pp. 1–10.

[3] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli, "QoS management through adaptive reservations," *Real Time Syst.*, vol. 29, nos. 2–3, pp. 131–155, 2005.

[4] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart, "On the sustainability of the extreme value theory for WCET estimation," in *Proc. 14th Int. Workshop Worst Case Execution Time Anal.*, 2014, pp. 21–30.

[5] J. L. Diaz *et al.*, "Stochastic analysis of periodic real-time systems," in *Proc. 23rd IEEE Real Time Syst. Symp. (RTSS)*, 2002, pp. 289–300.

[6] (2015). *The Debie1 Benchmark*. [Online]. Available: https://www.irit.fr/wiki/doku.php?id=wtc:benchmarks:debie1

[7] H. Falk *et al.*, "TACLeBench: A benchmark collection to support worst-case execution time research," in *Proc. 16th Int. Workshop Worst Case Execution Time Anal. (WCET)*, 2016, pp. 1–10.

[8] G. Bernat, A. Colin, and S. M. Petters, "WCET analysis of probabilistic hard real-time systems," in *Proc. 23rd IEEE Real Time Syst. Symp. (RTSS)*, 2002, pp. 279–288.

[9] A. Betts and A. Marref, "WCET analysis of component-based systems using timing traces," in *Proc. 16th IEEE Int. Conf. Eng. Complex Comput. Syst.*, 2011, pp. 13–22.

[10] U. K. Urimi, M. R. Kongara, and C. R. Patil, "Real-time implementation of modified adaptive histogram equalization for high dynamic range infrared images in FPGA," in *Proc. 5th Nat. Conf. Comput. Vis. Pattern Recognit. Image Process. Graph. (NCVPRIPG)*, 2015, pp. 1–4.

[11] M.-E. Ilas, "New histogram computation adapted for FPGA implementation of HOG algorithm: For car detection applications," in *Proc. 9th Comput. Sci. Electron. Eng. (CEEC)*, 2017, pp. 77–82.

[12] C. Kelly, F. M. Siddiqui, B. Bardak, and R. Woods, "Histogram of oriented gradients front end processing: An FPGA based processor approach," in *Proc. IEEE Workshop Signal Process. Syst. (SiPS)*, 2014, pp. 1–6.

[13] Y. Yang, Y.-X. Liu, and Q.-F. Dong, "Sliced integral histogram: An efficient histogram computing algorithm and its FPGA implementation," *Multimedia Tools Appl.*, vol. 76, no. 12, pp. 14327–14344, 2017.

[14] N. Stekas and D. van den Heuvel, "Face recognition using local binary patterns histograms (LBPH) on an FPGA-based system on chip (SoC)," in *Proc. IEEE Int. Parallel Distrib. Process. Symp. Workshops (IPDPSW)*, 2016, pp. 300–304.

[15] E. Díaz *et al.*, "Mitigating software-instrumentation cache effects in measurement-based timing analysis," in *Proc. 16th Int. Workshop Worst Case Execution Time Anal. (WCET)*, 2016, pp. 1–11.