# Non-Intrusive Online Timing Analysis of Large Embedded Applications

**Boris Dreyer**
Computer Systems Group, TU Darmstadt, Germany
dreyer@rs.tu-darmstadt.de

**Christian Hochberger**
Computer Systems Group, TU Darmstadt, Germany
hochberger@rs.tu-darmstadt.de

## Abstract

A thorough understanding of the timing behavior of embedded systems software has become very important. With the advent of ever more complex embedded software e.g. in autonomous driving, the size of this software is growing at a fast pace. Execution time profiles (ETP) have proven to be a useful way to understand the timing behavior of embedded software. Collecting these ETPs was either limited to small applications or required multiple runs of the same software for calibration processes. In this contribution, we present a novel method for collecting ETPs in a single shot of the software at very high quality even for large applications.

**2012 ACM Subject Classification** Computer systems organization → Real-time systems; Computer systems organization → Embedded systems

**Keywords and phrases** WCET, Execution Time Profiling, ARM CoreSight, Event Stream Processing

**Digital Object Identifier** 10.4230/OASIcs.WCET.2019.2

## 1 Introduction and Background

### 1.1 Motivation

A thorough understanding of the execution time of embedded systems software is crucial for the development of reliable systems. Yet, an analytical approach for this understanding has become almost impossible with modern System-on-Chip (SoC) architectures. The overall execution time is dominated by unpredictable effects like caches with random replacement policy.

Also, this effect demands more understanding of the timing as just min and max times for functions or loops. Rather, it requires a full execution time profile (ETP), which shows the probability for all possible execution times.

Different ways exist to collect such an ETP. Although it is possible to instrument the code to create a statistics for all parts of the code, this way highly influences the execution time and thus comes with severe drawbacks. The best alternative is to use the trace data that is provided by modern SoCs [1]. Unfortunately, commercial tools currently only offer to record this trace data. Given the data rate and the complexity of typical systems, this is not feasible, as the amount of data can easily reach TB. The only truly viable alternative is online processing of trace data.

### 1.2 Online Processing of Trace Data

To the best of our knowledge, the CONIRAS platform [5] was the first that was able to entirely process the trace data produced by modern ARM processor cores. This is non trivial, as the trace information is highly compressed and packetized. Only in the beginning of a packet an absolute reference is given. The remaining packet uses only relative addressing. Online processing consists of: (a) Reconstruction of the program addresses from the trace

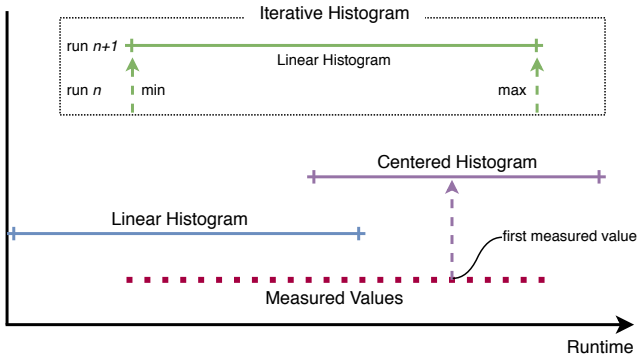


**Figure 1** Schematic comparison of our previous histogram algorithms. The range of measured values are shown as red dotted line. The covered value range by a linear histogram is shown as blue line, by a centered histogram as purple line and by an iterative histogram as green line.

stream (b) Mapping these addresses to individual basic blocks of the application under test (AuT) and (c) aggregating the information in a suitable form. Timingwise, the CONIRAS project targeted only worst case execution time (WCET). Support for histograms was added later. We started with linear and centered histograms. Both were using a bin size that the user had to choose. Later, we devised an iterative approach, where the optimal bin size is evaluated by multiple runs of the embedded software.

Essentially, the CONIRAS platform was limited by the amount of memory that was available on the underlying FPGA. All address lookups and all statistics were using BRAMs of the FPGA (even if it was not necessary).

## 1.3 Related Work

Apart from ETPs, hardware-implemented histogram are often used in image and video processing. Gautam presents a parallel histogram calculation architechture for image processing [7]. A hardware efficient, simplified Histogram of Orientation Gradient (HoG) module in Scale Invariant Feature Transform (SIFT) for describing key-point detected using Gaussian Scale Space (GSS) is proposed in [11]. Maggiani et al. propose in [10] an optimized design of a histogram extractor algorithm targeted to low-complexity and low-cost FPGA-based Smart Cameras. [14] uses them for Adaptive Histogram Equalization. In [8] and [9], they are part of the Histogram of Oriented Gradients algorithm for object detection. [15] presents a sliced Integral Histogram algorithm for efficient histogram computation. [13] uses Local Binary Patterns Histogram for face recognition.

In contrast to these related works, the distribution of function runtimes is highly dynamic. Therefore, we could not use the known methods for constructing histograms in hardware and had to develop new histogram algorithms [2, 3].

The remainder of this paper is structured as follows: Section 2 explains the limitations of the CONIRAS platform and analyzes the causes. Our novel approach for collecting ETPs is presented in Section 3. The overall platform and the tool flow are presented in Section 4. Finally, in Section 5 a conclusion and an outlook are given.

## 2 Limitations of Previous Hardware-optimized Histogram Algorithms

In this section we want to recapitulate existing hardware-optimized histogram algorithms and explain why these algorithms are not adequate for analyzing large embedded applications. In [2] and [3] we presented a linear, a centered and an iterative histogram algorithm. The schematic bin distributions of these histograms are shown in Figure 1.

**Linear Histogram Algorithm.** Linear histograms span a range from 0 to an upper bound with equally distributed bins over the full range. The size of the last bin is unlimited. In this way, the histogram uses its last bin to count the amount of values that were outside of its range. This information is used to judge the quality of the histogram. The linear algorithm was developed for the runtime measurement of Waypoint Edge Events (WPEs), as they usually take only a few cycles.

The blue lines of Figure 2 show the coverage of the debie1 task runtimes with linear histograms. It can be seen that a linear distribution with starting point 0 is only suitable for the measurement of functions with short runtimes.

**Centered Histogram Algorithm.** The centered bin distribution has been developed by assuming that the expected runtimes are clustered around a reference value. To obtain a more detailed view of the measured values, the bins are distributed around this reference value [2]. In contrast to the linear histogram, the first and last bin are used to store the number of values that are outside the distribution. This algorithm requires more memory than the linear algorithm because the reference value of each histogram must be stored during the analysis of the AuT.

The purple lines of Figure 2 show the coverage of the debie1 task runtimes with centered histograms. The histograms have been configured to use the first measured value as reference value and that the reference value is in the upper quarter of the total distribution. For functions with longer runtimes, this distribution is better suited than the linear distribution. Nevertheless, the coverage is not satisfactory to create good ETPs.

**Iterative Histogram Algorithm.** The iterative algorithm was developed for the runtime measurement of functions. Unlike the execution time of WPEs, the runtime of functions can have high dynamics. For example, if sensors supply input data and the runtime of a function depends on these inputs. The algorithm consists of multiple runs. In each run, a linear histogram is constructed and the minimum and maximum measured values are stored. The lower and upper bound of these histograms are derived from the measured minimum and maximum values of the previous runs. The bounds for the first run are defined by the user. The algorithm terminates when the amount of entries in the smallest and largest bin are below a user defined threshold. It is important to know that adjusting the minimum and maximum boundary of a linear histogram requires to re-synthesize the histogram hardware and this takes several minutes.

Therefore, this iterative approach is unsuitable for large embedded applications. It may take a long time before the function to be analyzed is executed. If the function has high dynamics in its runtime, several iterations are necessary to get good histograms which increases the time drastically.

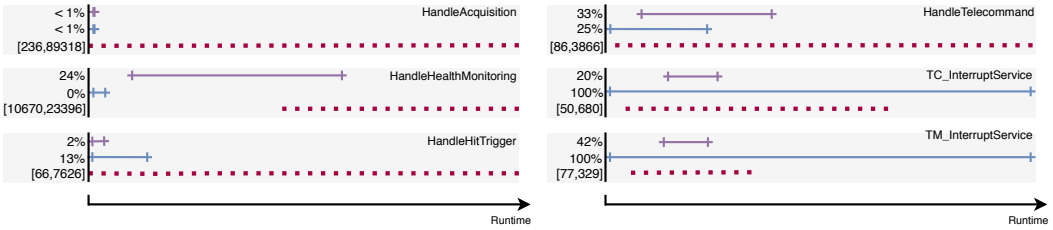


**Figure 2** Measured debie1 task execution cycles (red dotted lines) covered by linear (blue lines) and centered (purple lines) histograms with 128 bins of size 8. Numbers for linear and centered distribution are taken from [2]. The iterative algorithm covers the complete range of measured values after the second run and is therefore not drawn as evaluated in [3].

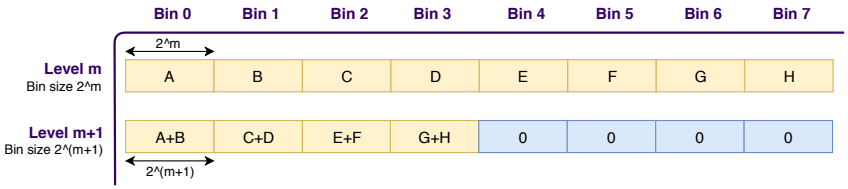


**Figure 3** Visualization of one compression step of the scalable histogram algorithm. Two adjacent bins are summed up and the remaining bins are set to zero.

## 3 Scalable Histogram Algorithm

For the calculation of meaningful ETPs, the underlying histograms must be as detailed as possible. This means that the histograms should have a small bin size and the histogram should be able to assign each measured value to a bin. The runtimes of functions, tasks and loops can differ from execution to execution. This may be caused by data dependencies or cache effects. It is not uncommon for the runtime of a function to depend on the function's parameters. To compute histograms with high event frequency in hardware, we fixed the number of bins of a histogram during the hardware synthesis, see Section 4.

Therefore, the idea of the scalable histogram algorithm is to increase the size of all histogram bins dynamically during the analysis of an AuT. This allows the histogram to process a wider range of measured values if necessary. Each histogram has an initial bin size of 1. Whenever the histogram is not capable to store a measured value because this value is too large to be stored in the highest bin the sizes of all bins are doubled. During this duplication, two adjacent bins are merged, i.e. their values are added and the remaining bins are set to zero, see Figure 3. This phase is called a compression and is repeated until the measured value can be stored in a bin. The number of times a histogram has been compressed is called the histogram's compression level and is stored together with each histogram. This level is later used by the timing analysis software to recalculate the border of each bin.

**Example.** In this section we want to demonstrate the algorithm with an example. This example uses a scalable histogram with 8 bins to process the event sequence <5,4,11,7,54,10>. Figure 4 shows how this sequence is processed in steps (a) to (j). These steps are now explained.

At the beginning of this example the histogram is in its initial state, shown in step (a). This means that all bins are cleared and the histogram's compression level is 0. Therefore, each bin has a size of 1 and the histogram can process values between 0 and 7 without increasing its compression level, i.e. the need of one or more compressions.

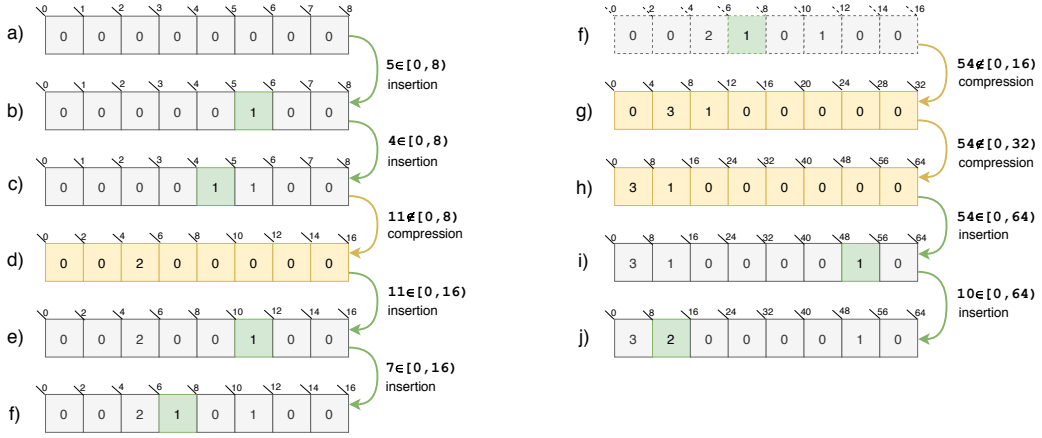


**Figure 4** Scalable histogram algorithm example showing an 8 bin histogram processing the event sequence <5,4,11,7,54,10> step by step.

The first sequence value is 5 and because it is within the current range of the histogram the corresponding bin is updated (b). The next sequence value is 4 and is also within the current range of the histogram so the corresponding bin can be updated too (c). The following value is 11 and is not within the range of the histogram. Before this value can be processed, the histogram must be compressed to double its range (d). The compressed histogram can process values between 0 to 15 and is able to process the value 11. The updated bin is shown in step (e). The next sequence value is 7 and can be processed without further compressions (f). Then value 54 is processed, which requires the two consecutive compressions depicted as steps (g) and (h) before the corresponding bin can be updated in step (i). The first compression extends the range of the histogram to process values between 0 and 31 and the second compression to process values between 0 and 63. The final value 10 can be processed without any compressing.

**Evaluation.** In this evaluation we compare ETPs constructed from scalable histograms with 64 and 128 bins with perfect ETPs. Perfect ETPs are the most detailed ETPs, i.e. they are constructed from histograms with bin sizes of 1 which could not be realized in hardware. Since the number of histogram bins now depends on the measured values, we constructed these perfect ETPs through simulation.

Figure 5 overlays these ETPs for all debie1 tasks and Interrupt Service Routines (ISRs). The debie1 benchmark was chosen to ensure comparability with the evaluation results of [2] and [3]. It can be seen that we can construct ETPs with a good precision/memory trade-off for each debie1 task and ISR with our new algorithm. Unlike our previous algorithms, this construction was done in one analyzing pass and without any previous runtime information about the tasks to be analyzed.

## 4 Scalable Histogram Platform

In this section we want to introduce a timing analysis platform that calculates scalable histograms and simple statistics in hardware. These calculations are done in realtime during the analysis of an AuT. This platform is an extension of the CONIRAS platform which had the original purpose to compute the WCET of an AuT.

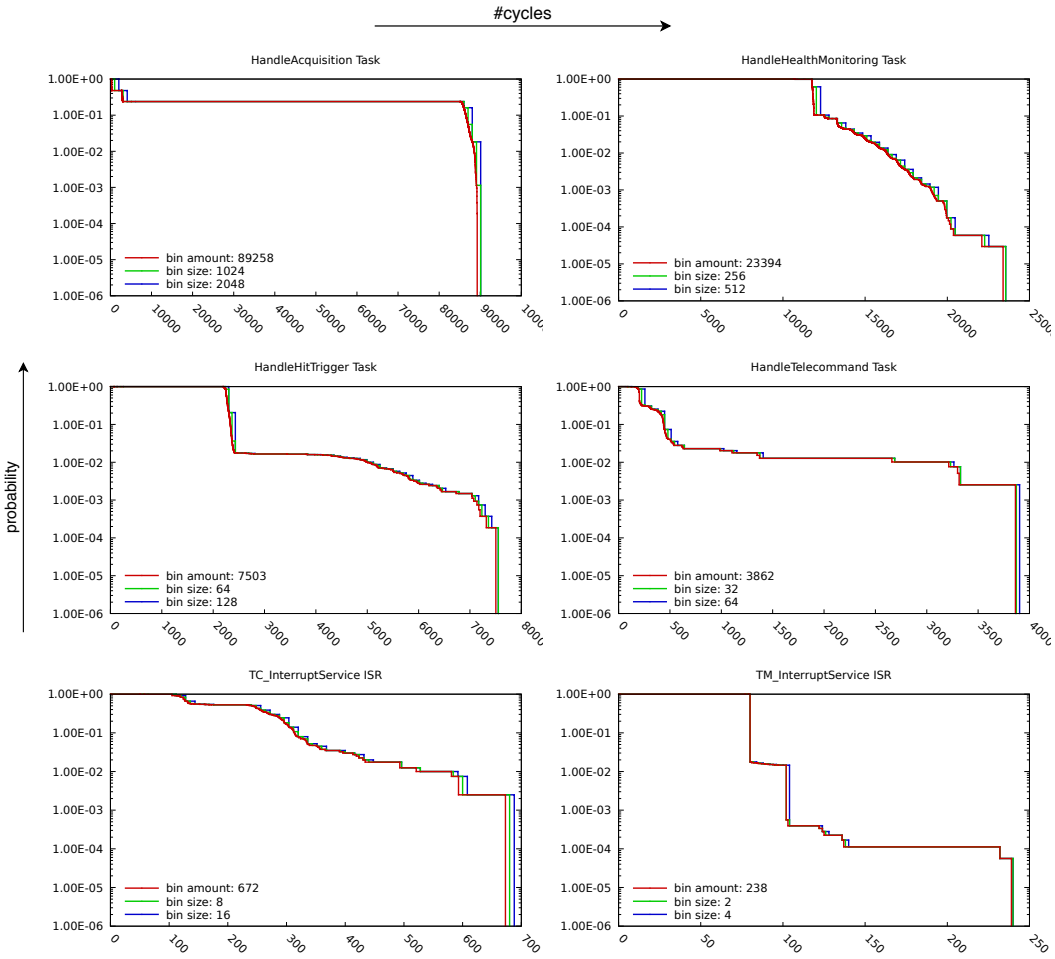


**Figure 5** Execution cycles ETPs of all debie1 tasks and ISRs generated from 70,000,000 WPE. Red lines are simulated perfect ETPs i.e. the underlying histograms have a bin size of 1; green lines are ETPs from scalable histograms with 128 bins, blue lines are ETPs from scalable histograms with 64 bins.

## 4.1 Workflow

Figure 6 shows the workflow to compute timing statistics of embedded applications online in a non-intrusive way. This workflow only requires the application as binary file.

### Preprocessing

The first step to analyse an AuT is to reconstruct its Waypoint Graph (WPG) [4] with tools like aiT [6] and Angr [12].

After that, the *Timing Analysis Module Generator* generates the *Timing Analysis Module* that measures the function runtimes and loop iterations of the AuT and also calculates and stores function runtimes and loop iterations statistics in hardware. In order to let the user specify which functions and loops of an AuT should be analyzed and which statistics should be generated we developed the *Statistics and Logging Language* (SALL). SALL is not in focus of the paper and will therefore not be discussed further.

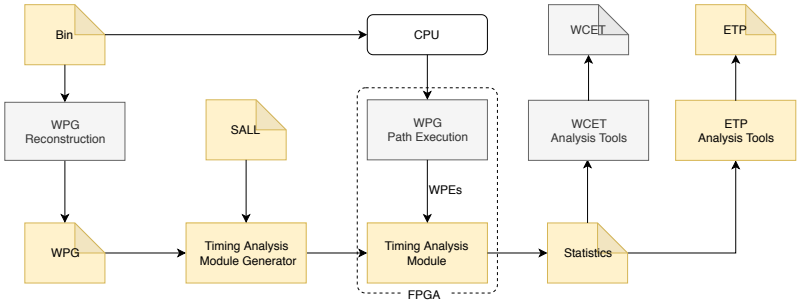


**Figure 6** Workflow of our timing analysis platform for analyzing large embedded programs. The *Timing Analysis Module* has been developed for this purpose. Gray parts could be reused from our CONIRAS platform and its extensions.

### Online Analysis

The *Timing Analysis Module* uses Waypoint Edge Events (WPE) emitted by the trace-based *WPG Path Execution Module* to calculate and store scalable histograms for function runtimes and loop iterations. A WPE consists of the executed edge of the WPG, i.e. an edge ID and the amount of cycles since the previous WPE was emitted.

Instead of histograms the module can also calculate and store simple statistics. Simple statistics means to compute and store the minimum, maximum and total runtime of a function and the minimum, maximum and total iterations of a loop. It also stores how often a function and loop was executed. Furthermore, it can also calculate and store simple statistics of low-level context-sensitive WPE. Context-sensitive WPE statistics means to compute and store the minimum, maximum and total runtime of an event that were executed during the first iteration of its innermost surrounding loop separated from the measured executions of that event during further iterations of its innermost surrounding loop. It also context-sensitively stores how often that WPE was executed.

### Postprocessing

After the analysis is done the *Timing Analysis Module* transfers the generated statistics to the host computer. This computer can than be used to perform high level analysis like ETP or WCET calculation.

## 4.2 Timing Analysis Module

This module consists of a loop and function automata cluster, forwarding trees, storage controllers and statistics storages. Each automaton generates context-sensitive WPE runtimes, loop and function runtimes, and loop iterations amount data. The forwarding and poll trees are used to serialize these data. The poll tree uses a round robin mechanism to poll information out of the automata's output buffer, depicted in Figure 7. This buffer mechanism is necessary because multiple automata can emit loop and function events at the same time. Whereas a simple forwarding is sufficient to forward the context-sensitive WPE runtimes, because this information is only emitted by the innermost function or loop. The *Storage Controller* uses the statistics layout information of the automata cluster together with their emitted values to generate control signals for the *Statistics Storage Module* to calculate simple statistics and histograms.

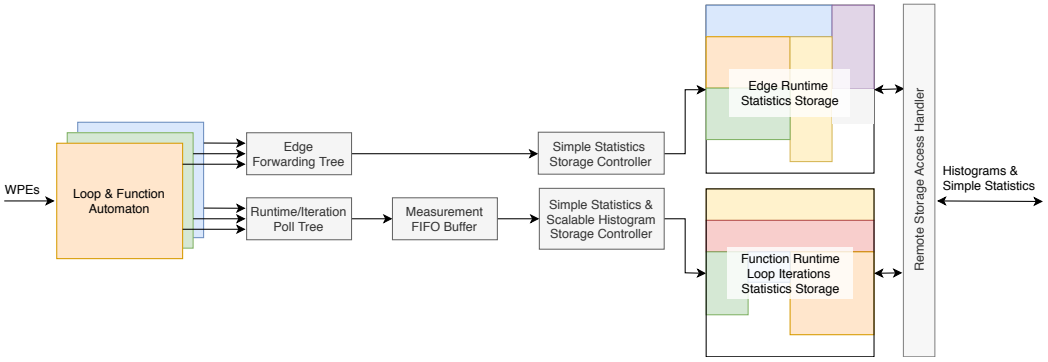


**Figure 7** Structure of the *Timing Analysis Module* consisting of a loop and function automata cluster, forwarding trees, storage controllers and statistics storages.

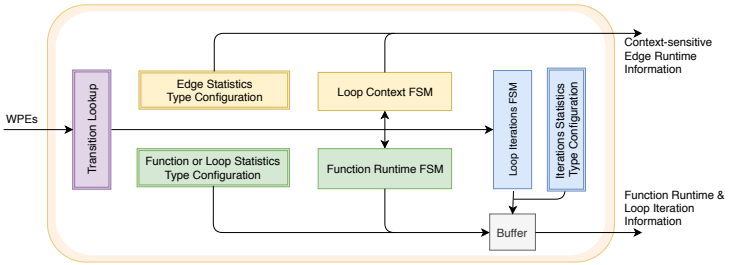


**Figure 8** Structure of one *Loop & Function Automaton Module*.

## Loop and Function Automaton Module

This module is able to analyze one loop or one function of an AuT and its structure is shown in Figure 8. It processes WPEs which are generated by the *WPG Path Execution Module* by interpreting low level trace packets like atom packets, timestamp packets, branch address packets and waypoint update packets for an ARM CoreSight [1] trace.

The green part of Figure 8 calculates the runtime of a function or loop. The blue part counts the iterations of a loop. The yellow part determines if this automaton represents the currently innermost executed loop or function. It also determines the current context of the executed WPEs. All these parts use Finite-State Machines (FSMs) to manage their computations. The transition events of these FSMs are generated by the purple part. It uses a rewritable lookup to map WPEs to transition events. These lookups are realized with distributed RAM for small functions and BRAM for large functions. By altering the lookup content through meta-configuration at runtime the automaton can be used to analyze another loop or function.

Each automaton also stores information about which statistics should be generated from its measurements and where they should be stored in *Statistics Storage Module*. This information was defined during the preprocessing phase by the *Timing Analysis Module Generator* and is used by the *Storage Controller Module* to calculate the memory addresses for updating the correct statistics.

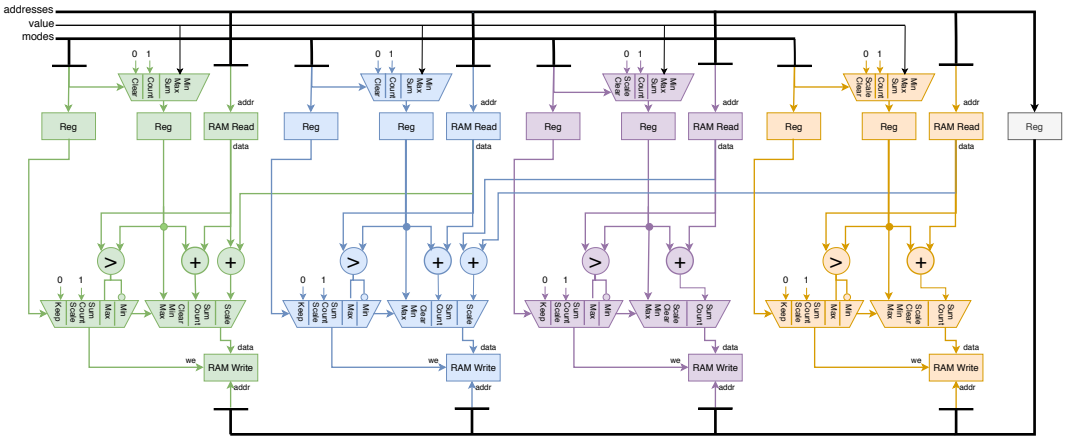


**Figure 9** Timing critical part of one *Statistics Module*. It computes and stores histograms with four bins or simple aggregation function, i.e. min, max, sum and count. Forwarding paths to update the same memory location several times in a row are not shown for clarity.

### Statistics Storage

Figure 9 gives a detailed insight of the timing critical datapath core of a *Statistics Storage Module* with four statistics cells. Each cell is colored differently for a better overview. The four cells can be used to calculate histograms with up to four bins or simple statistics. Each cell contains a dual ported RAM to store its statistics values and is connected to a *Storage Controller* that drives the individual address, mode and value signals. The mode signal determines the statistics operation which the cell applies to the value signal and the corresponding stored value. The address signal determines which stored value is updated.

Each cell is able to calculate simple min, max, sum and count statistics. The cells can also work together to calculate scalable histograms with four or two bins. If the first two cells work together to calculate histograms with two bins, the remaining cells can be used for simple statistics. This generic cell concept can be scaled to compute histograms with way more bin.

### Statistics Controller

The *Simple Statistics and Scalable Histogram Statistics Controller Module* generates the control signals for the *Statistics Storage Module* to update histograms and simple statistics. To this end, the module implements the scalable histogram algorithm and stores the compression level of each histogram.

From the automata cluster it receives the measured function runtimes and loop iterations and the statistics type to use. The statistics type defines whether simple statistics or a histogram should be created and where in the memory of the *Statistics Storage Module* these statistics should be stored.

## 4.3 Measurement FIFO Buffer Evaluation

As long as no histogram compression is necessary, the *Statistics Controller Module* can process one event every clock cycle. One histogram compression requires one additional cycle. Because the cluster can generate an event every cycle, a first-in-first-out (FIFO) buffer was placed between the cluster and the Statistics Controller.

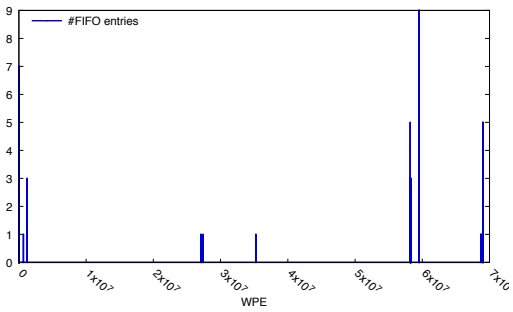


**Figure 10** Measurement FIFO buffer utilization during the complete debie1 benchmark analysis.

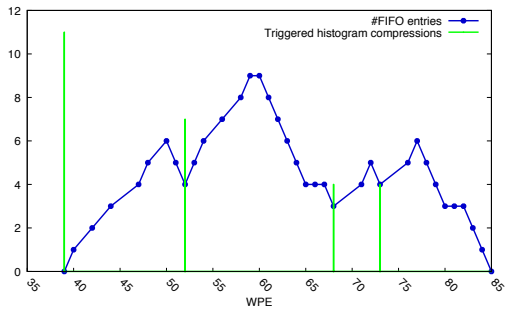


**Figure 11** Measurement FIFO buffer utilization between processing WPE 59,529,035 and 59,529,085.

It is important that this buffer does not overflow during the analysis of an AuT, as this falsifies the analysis results. It is also important that this buffer can be implemented in hardware, i.e. it does not require unrealistic amounts of RAM. Therefore, we simulated the utilization of this buffer during the analysis of the debie1 benchmarks with prerecorded WPEs from [3]. This means that our target SoC to record the WPEs was a Xilinx Zynq XC7Z020 featuring a dual-core ARM Cortex-A9 running at 667 MHz. We compiled the benchmark with the C++ compiler `GNU C/C++ 4.9.2 20140904 (prerelease)` to ensure comparability with the evaluation results of [2] and [3]. The benchmark was executed on one core, while the other core executed a program that was used to generate interferences on the shared L2 cache and the shared interconnects.

Figure 10 shows the FIFO utilization during the analysis of the complete benchmark. To this end, almost 70,000,000 prerecorded WPEs were processed and 240 scalable histograms with 64 bins were constructed to be able to analyze all 68 loops and 172 functions of the benchmark. It can be seen that the FIFO is almost always empty and only buffers a maximum of 9 events at the same time.

This is shown in detail in Figure 11. The blue line shows the amount of buffered events for each processed event. The green lines show whenever a processed event triggers one or more compression steps of a histogram. The height of these green lines show the amount of compression steps that are necessary to process this event.

This evaluation has shown that the buffer can be implemented in hardware and requires only a few resources for the debie1 benchmark.

## 5 Conclusion and Future Work

In this contribution, we have presented a novel approach to collect histograms of execution times for large embedded applications. No instrumentation of the software is needed and thus, the timing behavior of the embedded system is not influenced. The histograms are gathered in a single shot of the AuT, yet they are so precise that we can compute very good ETPs from them. We have presented a measurement platform that can be tailored to the needs of the user. Also, we have analyzed the critical elements of the platform with respect to their HW implementation.

In the future, we want to use our domain specific language to automatize the customization of the measurement platform. This will enable us to automatically realize lookup functions in the best suitable type of memory. Since we will need less BRAMs for the lookup, more histograms can be gathered at the same time then.

## References


1 ARM Ltd. CoreSight™ Architecture Specification v2.0, 2013. ARM IHI 0029B.

2 T. Ballenthin, B. Dreyer, C. Hochberger, and S. Wegener. Hardware Support for Histogram-Based Performance Analysis of Embedded Systems. In *2017 IEEE 20th International Symposium on Real-Time Distributed Computing (ISORC)*, 2017.

3 B. Dreyer, C. Hochberger, T. Ballenthin, and S. Wegener. Iterative Histogram-based Performance Analysis of Embedded Systems. *IEEE Embedded Systems Letters*, 2018.

4 Boris Dreyer, Christian Hochberger, Alexander Lange, Simon Wegener, and Alexander Weiss. Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs. In *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016.

5 Boris Dreyer, Christian Hochberger, Simon Wegener, and Alexander Weiss. Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation. In *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2015.

6 Christian Ferdinand and Reinhold Heckmann. aiT: Worst-case execution time prediction by static programm analysis. In René Jacquart, editor, *Building the Information Society. IFIP 18th World Computer Congress, Topical Sessions, 22–27 August 2004, Toulouse, France*, pages 377–384. Kluwer, 2004.

7 K. S. Gautam. Parallel Histogram Calculation for FPGA: Histogram Calculation. In *2016 IEEE 6th International Conference on Advanced Computing (IACC)*, pages 774–777, February 2016. doi:10.1109/IACC.2016.148.

8 M. E. Ilas. New Histogram Computation Adapted for FPGA Implementation of HOG Algorithm: For Car Detection Applications. In *2017 9th Computer Science and Electronic Engineering (CEEC)*, 2017.

9 C. Kelly, F. M. Siddiqui, B. Bardak, and R. Woods. Histogram of Oriented Gradients Front End Processing: An FPGA based Processor Approach. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, 2014.

10 L. Maggiani, C. Salvadori, M. Petracca, P. Pagano, and R. Saletti. Reconfigurable architecture for computing histograms in real-time tailored to FPGA-based smart camera. In *2014 IEEE 23rd International Symposium on Industrial Electronics (ISIE)*, pages 1042–1046, June 2014. doi:10.1109/ISIE.2014.6864756.

11 E. P. R. Raj, B. S. Paul, and G. L. Narayanan. Simplified SIFT Histogram of Oriented Gradients Bin Locator on FPGA. In *2018 9th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–4, July 2018. doi:10.1109/ICCCNT.2018.8493928.

12 Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 138–157, May 2016.

13 N. Stekas and D. v. d. Heuvel. Face Recognition Using Local Binary Patterns Histograms (LBPH) on an FPGA-Based System on Chip (SoC). In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2016.

14 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 *2015 Fifth National Conference on Computer Vision, Pattern Recognition, Image Processing and Graphics (NCVPRIPG)*, 2015.

15 Yang Yang, Yun-Xia Liu, and Qi-Fan Dong. Sliced Integral Histogram: An Efficient Histogram Computing Algorithm and its FPGA Implementation. *Multimedia Tools and Applications*, 2017.