2021 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION EUROPE OCTOBER 26-27, 2021

Method for early performance verification of hardware-accelerated embedded processor systems in RTL simulation

Luca Sasselli, QT Technologies Ireland Limited, Cork, Ireland (*lsassell@qti.qualcomm.com*)

Mehmet Tukel, QT Technologies Ireland Limited, Cork, Ireland (*mtukel@qualcomm.com*)

David Guthrie, QT Technologies Ireland Limited, Cork, Ireland (*dguthrie@qti.qualcomm.com*)

***Abstract*— Meeting performance requirements is one of, if not the, most critical aspects of the design of a hardware-accelerated embedded processor system. Yet, performance verification is generally done late in the system development cycle, usually during silicon emulation or validation, once software and hardware are sufficiently mature to be integrated together. While some assumptions can be made during the early phases of the design, if incorrect, those choices can potentially lead to an underperforming system, or an overdesigned one, leading to an increase in non-recurring engineering cost. As such, performance extraction is required in the early stages of the system development to validate architecture choices, highlight bottlenecks, and steer the design to meet performance requirements, without compromising area and power. In this paper, we propose a method to extract, process and verify performance in hardware-accelerated embedded processor systems, that can be quickly implemented in the design verification flow and reused for a wide range of applications. Additionally, we use the flow in a case study highlight the key aspects of the methodology used, the lessons learned and the obtained results.**

Keywords—Performance verification, RISC-V, RTL design

I. INTRODUCTION

A hardware accelerated embedded processor system, is an electronic system in which a general-purpose central processing unit (CPU) offloads complex functions to dedicated hardware, to increase the data processing capabilities, reduce power consumption or area footprint. As such, performance is highly dependent, not only on the number and type of hardware resources, but also on how the software interacts with them, the system architecture, and the underlying physical implementation.

The design cycle of this type of system starts with the architecture definition: an initial analysis is performed based on the metrics of existing components, determining parallelism opportunities and defining the hardware-software partitioning, to ensure that the performance goals are met [1]. Once the architecture is established, hardware and software are often co-designed: this demands good architectural choices to be made and implemented correctly at the hardware level, to ensure that the requirements are satisfied once software is integrated. Without intermediate verification before this stage, an incorrect choice might lead to an underperforming system, requiring major components to be redesigned, thus an increase in non-recuring engineering (NRE) cost, or an overperforming one, with sub-optimal area footprint or power consumption. For this reason, performance extraction (such as latency, throughput, resource utilization, etc.) should be done in the RTL design verification flow, using dummy software to generate realistic stimuli, providing the designers with valuable information about the system performance, highlighting architecture level issues as early as possible.

In this paper we propose a framework to enable early performance extraction of a hardware accelerated embedded processor system in RTL simulation. The proposed technique allows to quickly develop software to be used as stimuli during the design verification process, and extract relevant performance measurements from it, with minimal effort.

II. RELATED WORK

Performance verification during RTL design, is rarely considered in literature for hardware-software codesigns [1] [2], relying instead on architecture based on high-level system modeling, using tools such as SystemC [3] [4]. This gap between architecture and implementation represent a significant risk especially in the hardware development process [5], and many have highlighted the advantage of performance verification in the design verification flow to highlight performance bugs [6] [7] [8]. While benchmarks programs and test suites, such as Dhrystone [9] and MiBench [10], do exist and provide performance metrics for embedded systems, their scope is limited to the processor core itself, ignoring any external hardware acceleration.

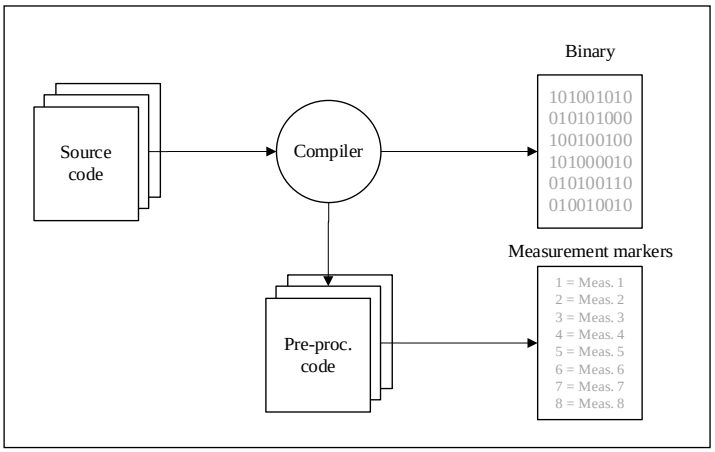

Figure 1 Measurement extraction flow.

III. PROPOSED APPROACH

To extract performance, we propose a simple yet reusable approach, that enables measuring the duration of a performance relevant event directly from C/C++ software running on the embedded processor. The main idea behind this technique is to use memory write accesses to a predefined memory location to mark the beginning and the end of the event. The testbench then detects write access to that location (e.g. by monitoring the memory I/O signals), identifying the events by the value written, and using the simulation time to measure the delta between them. This is achieved by marking the software using special pre-processor macros, that expand to a data write that uniquely identifies that position in the software, and an inert section of code (such as a comment) that contains the information about the underlying code. The latter can then be parsed from pre-processor output and passed to the testbench monitor to associate data writes with events.

During RTL simulation, when a performance critical section of code is executed, a memory access is produced to the location monitored by the testbench, thus allowing it to measure the simulation time delta between beginning and end of the section. These measurements can then be used in self-checking performance tests in regressions, compiled into reports to be reviewed with stakeholders or post-processed using a tool such as NumPy [11] to perform more refined statistical analysis.

The overall flow is presented in fig. 1 and is as follows: (1) software is developed and performance critical sections are marked using the measurement macros, (2) code is compiled saving pre-processor output (e.g. using -save-temps in gcc), (3) pre-processor output is parsed to extract all the measurements and compile a look-up table (LUT) that associates the unique data write value to a certain measurement, (4) the LUT is passed to the testbench to allow to associate memory accesses during simulation to a specific measurement.

```
#define MEASURE(x) \
    // MEASUREMENT: __LINE__ = x \
    *((volatile uint32_t*) MONITOR_ADDR = __LINE__
```

Figure 2 Example of measurement macro.

```
void interrupt_service_routine(void){
    MEASURE(ISR BEGIN);
    ...
    MEASURE(ISR END);
}
```

Figure 3 Example of macro usage in the source code.

```
void interrupt_service_routine(void){
    // MEASUREMENT: 2 = ISR BEGIN \
    *((volatile uint32_t*) MONITOR_ADDR = 2;
    ...
    // MEASUREMENT: 4 = ISR END \
    *((volatile uint32_t*) MONITOR_ADDR = 4;
}
```

Figure 4 Example of macro expansion after pre-processor parsing.

An example of the measurement macro is presented in fig. 2. In the proposed implementation, the macro takes a single parameter used to identify the event in a human readable form and expands it to a memory write access to the monitored address MONITOR_ADDR; volatile access is used to avoid the write from being removed at high compiler optimization levels. A unique identifier for the measurement is obtained by using the __LINE__ standard predefined macro, which expands to the line number where the macro itself is called. When more than one source code file is used, this must be combined with a user defined file identifier to allow to univocally identify the line. Additionally, a comment is produced to show the association between the human readable identifier and the unique data write value, used to build the LUT passed to the testbench monitor.

The main advantage of this approach is the relative simplicity of its implementation: since simple memory writes are used to track code execution, this flow can be implemented as soon as a "Hello word" program is running successfully on the CPU with little additional framework. The main disadvantage is the overhead introduced by the write to the memory, but this is usually easily quantifiable, or negligible when the memory involved is closely coupled to the CPU.

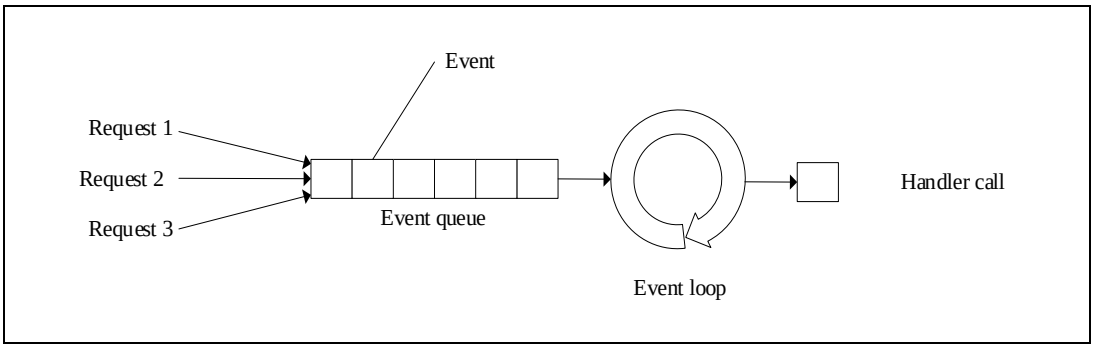

Figure 5 Event-loop software structure.

To enable rapid software development, we propose using an event-loop [12]. In traditional software, the structure of the code is linear and based on the flow of execution: this can make development arduous when the program must handle many asynchronous events or multiple tasks in parallel. While this issue can be addressed with the use of a Real-Time Operating System (RTOS), popular solutions such as FreeRTOS [13] tend to have high memory footprint, making them unusable on many devices.

Event-loop based software represents an interesting trade-off between functionality and requirements. In this approach the program is composed of two main components: an event queue and an event loop. When an asynchronous event occurs (e.g. an interrupt request) an event is added to the queue, with an associated event handler to perform the required task. In the main loop of the program, the queue is polled by the event loop, that calls the handler to react to the event.

IV. CASE STUDY

The proposed flow was used to characterize the performance of a cryptographical hardware accelerator system-on-chip (SoC) subsystem. The device under test, presented in fig. 5, is composed by a RISC-V processor, 3 cryptographical engines (CE), a direct memory access unit (DMA) and a AMBA AXI interface. The AXI interface is used to access an external random access memory (RAM) shared with a data provider (DP): using this shared memory as I/O interface, the subsystem provides cryptographical acceleration to DP, controlling the load on the

CEs to achieve maximum throughput at all time. While a CE can store in its internal memory datasets for more than one cryptographical operation, it cannot automatically start the next one without control commands issued by a third-party. Therefore, to ensure a quick turnaround time between operations, a scheduler FSM is added, to allow control commands to be pipelined and automatically issued when processing is complete.

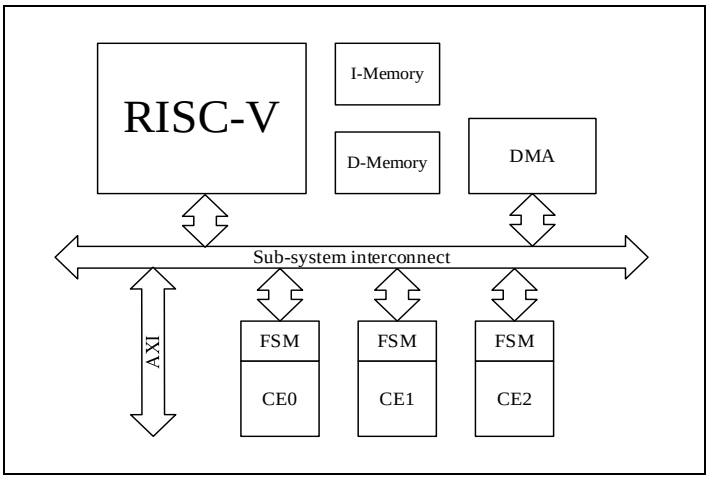

Figure 5 Case study system architecture

The device under test primarily operates as follows: (1) data is written by DP in the shared memory, (2) an interrupt is issued by DP to the RISC-V core to signal the presence of a new dataset, (3) the core programs the DMA to transfer the data from the shared RAM to the selected CE, (4) the core programs CE to perform the cryptographical operation, (5) CE interrupts the core once the operations is complete, (6) the result is transferred using DMA from CE to the shared memory and (7) the core issues an interrupt to DP to signal the presence of a result. To increase throughput, as soon as new dataset is available in the shared memory, the RISC-V core transfers it in the least busy CE, and programs the scheduler FSM, allowing the next operation to start as soon as the previous finishes.

Using the framework proposed in Section 3, software was developed to implement this sequence. Due to the design development schedule, performance analysis was started before the development and integration of the scheduler FSM, instead relying exclusively on the RISC-V processor for handling the CE configuration; to limit the impact over the performance, this operation was given the highest priority in software.

V. RESULTS OBTAINED

All time measurements reported are normalized by maximum allowed dataset processing time required to achieve the system target throughput, multiplied by 100 for readability.

To analyze the performance of the subsystem the following measurements were collected: (1) dataset read time, (2) CE configuration time, (3) CE processing time and (4) result write time. To estimate the benefit introduced by the DMA over the system performance, measurements were collected both using only the RISC-V core to transfer the data and with the aid of the DMA engine. Delay over the AXI interconnect was modelled using the average delay extracted from a previous generation SoC, assuming a gaussian distribution around the average value measured. Results are presented in table I.

Table I Initial measurements.

Measurement	***Average Time***	
	RISC-V only	***DMA***
Dataset read	7.53	1.98
CE configuration	0.30	0.30
CE processing	94.78	94.86
Result write	2.57	1.17
TOTAL	105.18	98.31

As expected, using DMA to perform the data transfers, significantly reduces the dataset load and result write time, but it's uncorrelated to the other two metrics. Interestingly, CE configuration takes a very short time, even without the scheduler FSM, hinting that this component might not be need. On the contrary, CE processing time contributes in both cases to more than 90% of the total delay. Statistical distribution of CE processing (figure 6) was studied, showing a gaussian distribution of the delay.

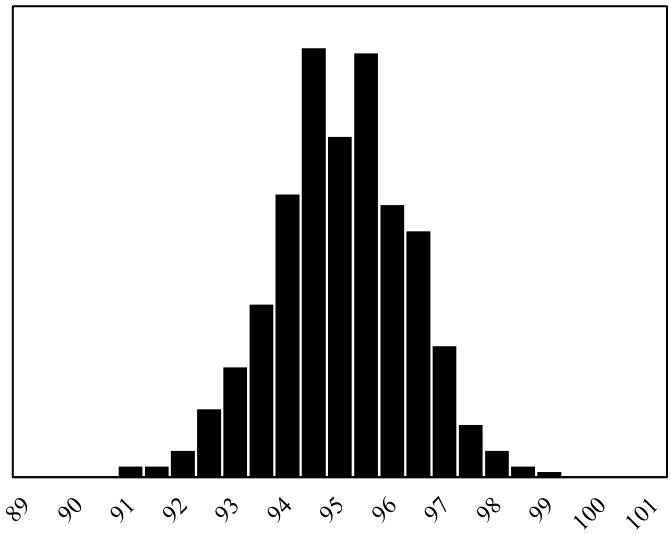

Figure 6 CE processing time distribution

Regardless whether the DMA was used or not, the overall throughput of the subsystem was identical, approximately 5% higher than the target requirement. This can be explained by CE processing time being the main bottleneck of the system. In fact, since

$$t_{data\ read} + t_{result\ write} < \frac{t_{CE\ processing}}{N_{CE}}$$

where $N_{CE} = 3$ is the number of CEs, dataset read and result write delay don't impact the overall system throughput, given that 3 of each could be performed within the time taken to complete a single CE processing. This condition ensures that when a CE finishes, a new dataset is already available in its memory, thus avoiding performance loss due to data starvation. Therefore, the throughput of the subsystem TP_{ss} can be computed as

$$TP_{ss} = \frac{N_{CE}}{t_{CE\ configuration} + t_{CE\ processing}}$$

If software prioritizes starting the next CE operation over other tasks (as it was done for this study), the benefit of a scheduler FSM is negligible, since would only affect the already low CE configuration time. A more effective way to increase performance is to simply increase the clock of CE, to reduce the processing time directly.

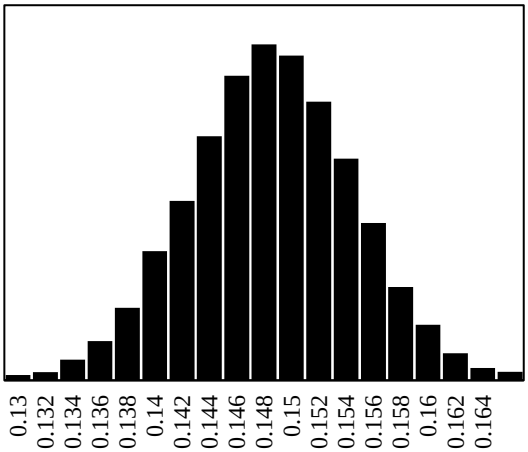

Figure 7 Gaussian read delay distribution.

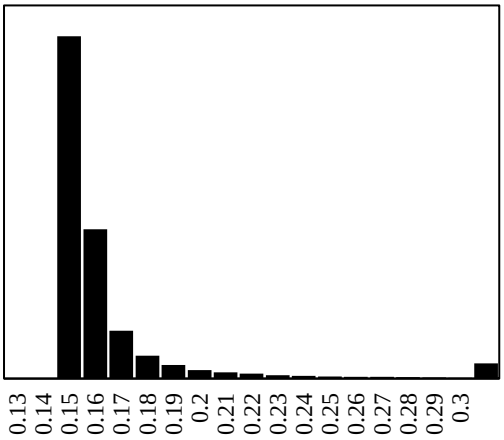

Figure 8 GEV read delay distribution.

To ensure that the requirements for data read and result write time could be achieved in less conservative conditions, measurements where repeated with a different delay model for the AXI interconnection based on a Type-II Generalized Extreme Value (GEV) distribution [14]. Compared to the model used previously (fig. 7), GEV (fig. 8) can model spikes in delay, that could occur in a real SoC when the shared memory interconnect is busy for long period of time. Despite this change, no significant impact on the total throughput was measured.

While these results increased the confidence in the predicted performance, and that it could be achieved without the addition of the scheduler FSM, questions were raised on the acceptability of the margin with the target throughput. The main concern was that CE programming time wasn't a realistic representation of the real software behavior, due to the much simpler implementation required by this study: with a more complex software framework, this component could grow significant enough to lower the performance below the required threshold.

To address these concerns the maximum allowed CE configuration time was studied for different CE clock frequencies using the theoretical model presented before, and then validated in simulation.

Table II Estimated performance at increased CE frequency.

CE freq. scaling	*Performance margin*	*Max. CE config. time*
1.0	5%	5.2
1.1	15%	13.8
1.2	26%	21.0

Additionally, the measurements were repeated using a higher compilation optimization level. Using O3 instead of O0 reduced the average CE configuration time from 0.3 to 0.18, showing that the original measurement wasn't representing an exceptionally optimistic case and that additional improvement was possible.

VI. CONCLUSIONS

The flow proposed in this paper demonstrates quick development of the software stimuli required to implement the case study, characterizing each design component and their impact on the cumulative system performance. A simple, yet accurate model of the system performance was developed and validated through experimentation, allowing characterization of the design space for different corner cases. Initial architecture analysis predicted that the CE programming time would have a significant impact on the subsystem throughput; therefore, recommending the design of a scheduling system. However, performance analysis showed this contribution to be negligible. This

component was subsequently dropped from the final subsystem, saving area, NRE cost of the design, and verification of the new logic.

VII. REFERENCES

[1] W. Wolf, High-Performance Embedded Computing: Architectures, Applications, and Methodologies, Morgan Kaufmann, 2006.

[2] R. Niemann, Hardware/Software Co-Design for Data Flow Dominated Embedded Systems, Springer US, 1998.

[3] H. Posadas, F. Herrera, P. Sánchez, E. Villar and F. Blasco, "System-level performance analysis in SystemC," in *Proceedings of the conference on Design, automation and test in Europe - Volume 1*, 2004.

[4] T. Wild, A. Herkersdorf and R. Ohlendorf, "Performance Evaluation for System-on-Chip Architectures using Trace-based Transaction Level Simulation," in *Proceedings of the Design Automation & Test in Europe Conference*, 2006.

[5] W. Cui and T. Sherwood, "Architectural Risk," *IEEE Micro*, no. 38, pp. 116-125, 2018.

[6] P. Ghosh and K. Chakravarthy, "Use-case based early performance simulation of cryptographic coprocessor," in *SoC Design Conference (ISOCC), 2013 International*, 2013.

[7] J. Holt, J. Dastidar, D. Lindberg, J. Pape and P. Yang, "System-level Performance Verification of Multicore Systems-on-Chip," in *2009 10th International Workshop on Microprocessor Test and Verification*, 2009.

[8] P. Ghosh and S. Rohit, "Case Study: SoC Performance Verification and Static Verification of RTL Parameters," in *2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification (MTV)*, 2019.

[9] R. P. Weicker, "Dhrystone," [Online]. Available: https://en.wikipedia.org/wiki/Dhrystone.

[10] MiBench, "MiBench," [Online]. Available: http://vhosts.eecs.umich.edu/mibench//index.html.

[11] numpy, "numpy," [Online]. Available: https://github.com/numpy/numpy.

[12] Wikipedia, "Event loop," [Online]. Available: https://en.wikipedia.org/wiki/Event_loop.

[13] FreeRTOS, "FreeRTOS," [Online]. Available: https://www.freertos.org/.

[14] wikipedia, "Generalized extreme value distribution," [Online]. Available: https://en.wikipedia.org/wiki/Generalized_extreme_value_distribution.

This template has been prepared and adapted for use in DVCon Europe 2021.