CMEmu: Synthesizing a Cycle-Exact Model of Program Execution on ARM Cortex-M from In-Code Timing Measurements

Maciej Matraszek, Mateusz Banaszek, Wojciech Ciszewski, Artur Jamro, Wojciech Kordalski, Daniel Gutowski, Michał Siwiński, Bartłomiej Dalak, and Konrad Iwanicki
Faculty of Mathematics, Informatics and Mechanics, *University of Warsaw*, Poland
{m.matraszek, m.banaszek, w.ciszewski, iwanicki}@mimuw.edu.pl

***Abstract*—The last decade witnessed considerable interest in how microarchitectural aspects of processors can impact computer systems, with an increasing focus on dependable low-power embedded systems. Multiple hardening and verification techniques for such systems rely on emulators that faithfully model code execution timings of real microcontrollers. However, in contrast to older ultra-low-power processor families, for the prevalent ARM Cortex-M family, only models derived from hardware sources are able to provide exact timings.**

In this paper, we examine the feasibility of synthesizing a cycle-exact timing model of a Cortex-M3-based microcontroller using solely in-code timing measurements and publicly available documentation. The main artifact of our work is CMEmu, to the best of our knowledge the first emulator of this kind, which provides exact timings for gigabytes of diverse programs from our extensive evaluation suite. We present techniques that we devised to achieve such an accuracy, which involved elaborate research methods to capture the various intricacies of the device microarchitecture, allowing us to even report a previously unknown hardware bug in the processor.

***Index Terms*—ARM Cortex-M; cycle-exact; timing model; emulation; microcontroller simulator; modeling techniques.**

I. Introduction

Models predicting how many processor cycles a given piece of code would execute on a given microcontroller, *microcontroller timing models*, are indispensable instruments for developing software for embedded devices, the devices themselves, as well as for exploring (or exploiting) various aspects of their operation. Usually implemented as emulators or simulators, they enable streamlining those processes, evaluating scenarios difficult to set up or scale in the real world, and obtaining insights normally unavailable with hardware.

A large family of microcontroller timing models are *cycle-accurate*: they approximate the number of processor cycles [1]–[3] or model only on a subset of microarchitectural details, achieving timing errors of a few percent compared to hardware [4]–[8]. In more demanding applications, however, the inaccuracies may be problematic [9], [10]. To illustrate, consider simulating a wireless system employing IEEE 802.15.4-based protocols utilizing concurrent transmissions, such as Glossy [11], which require devices to be synchronized under 0.5 µs. For a 48-MHz microcontroller, this is 24 processor cycles. In such cases, models are necessary whose timings not just approximate but always equal those of the corresponding chips: *cycle-exact* models. In general, apart from faithful simulation of wireless communication [12], [13], application examples of such models include identifying security side channels [14]–[19], firmware fuzzing [20], fault injection scheduling [21], high-fidelity performance analysis [14], power profiling [12], or chip-specific code optimizations [22].

There are two main approaches to building cycle-exact microcontroller models. One is deriving them from chip sources. It guarantees their timings but has a few compelling disadvantages: limited or none public availability due to including the chip manufacturers' intellectual property, the need for obtaining sources of all hardware components comprising a given microcontroller, and a low performance due to simulating the hardware at an ultra-low level [23], [24]. Addressing these issues is thus the main motivation behind the second approach: synthesizing timing models only from empirical observations of program execution on the considered hardware.

However, while empirically synthesized emulators that aim to be cycle-exact have been available for older low-power microcontroller architectures such as MSP430 (MSPSim [25]) and AVR (Avrora [26]), to the best of our knowledge none exists for the modern ARM Cortex-M family, which had over 70 billion chips shipped by 2021 [27]. The general belief [28]–[30] is that cycle-exact emulation of Cortex-M microcontrollers can be accomplished only with hardware-source-based models [31], presumably due to the internal complexity of those microcontrollers (multi-stage pipelines, prefetchers, customizable memories, bus configurations, arbiters, caches, etc.). Although there are multiple works exploring certain aspects of the Cortex-M microarchitecture [15]–[19] and on validation of timing models [9], [32]–[34], they have not resulted in a comprehensive cycle-exact model. To the best our knowledge, the available ones do not aim beyond being cycle-accurate: they exhibit noticeable overall timing errors (e.g. 1.8% [8]) and still contain specification and abstraction errors (following the taxonomy by Black et al. [35]).

This work was supported by the National Science Center (NCN) in Poland under grant no. 2019/33/B/ST6/00448. The evaluation was partially carried out with the support of the Interdisciplinary Center for Mathematical and Computational Modeling at the University of Warsaw (ICM UW).

This is the accepted version of the article:

M. Matraszek, M. Banaszek, W. Ciszewski, A. Jamro, W. Kordalski, D. Gutowski, M. Siwiński, B. Dalak, and K. Iwanicki, "CMEmu: Synthesizing a Cycle-Exact Model of Program Execution on ARM Cortex-M from In-Code Timing Measurements", 2025 International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM), Barcelona, Spain, 2025, pp. 350-359, DOI: 10.1109/MSWiM67937.2025.11308925.

The published version is available at https://ieeexplore.ieee.org/document/11308925.

Research questions. In this light, it is valid to ask: *Can a cycle-exact model of a modern microcontroller be devised without access to its hardware sources?* The question poses unique challenges, especially in contrast to cycle-accurate models, for instance: *How certain one can become that the model does not deviate by a single cycle not only on typical programs but also virtually all valid ones? How to track and fix causes of even single-cycle deviations?*

Studying these questions, we developed a timing model for a microcontroller with a Cortex-M processor. We chose Cortex-M3, as it constitutes a middle ground between the basic, low-cost Cortex-M0 or M0+, and the advanced, high-performance Cortex-M7 or M85, and has more intricate timings of some operations than Cortex-M4. However, to avoid limiting our approach to one particular chip, and thus to answer the question in a general context as well as devise versatile techniques, we restricted our research methods solely to obtaining data from programs run on the microcontroller (in-code timing measurements) and publicly available information about the microcontroller (such as technical documents, errata notices, books, online forums, free educational hardware sources of minor components, and related scientific publications). Moreover, no member of our team was ever affiliated with ARM or any chip manufacturer, and we had no experience in processor design. Likewise, our approach did not require specialized measurement hardware, such as rarely integrated built-in tracing modules (e.g., Embedded Trace Macrocell) or external equipment (e.g., electromagnetic probes). It also did not involve disassembling the modeled chip or otherwise accessing its internals (e.g., microscopic photos).

Contributions. Our contributions are twofold. First, we introduce a cycle-exact timing model of a Cortex-M3-based microcontroller (Texas Instruments CC2650), to the best of our knowledge, the first such model not based on the hardware sources of a modern and relatively complex ARM Cortex-M. We implemented the model as an emulator, CMEmu, which is open sourced,[1] and verified its accuracy on an enormous and diverse test suite containing real-world and synthetic programs. Second, we discuss the techniques we employed to develop the model. In contrast to previous works on timing models, we strove for the cycle exactness from the ground up and thus focused on comprehensively deriving and modeling the microarchitecture as a remedy for the specification and abstraction errors. The scrutiny is highlighted by our discovery of a hardware bug in Cortex-M3 itself. As such, we present completely novel techniques, and techniques inspired by previous works but advanced to achieve the presented accuracy.

II. Background

Cortex-M3 implements the ARMv7-M architecture [36] with Thumb instructions, which have to be 2-bytes-aligned and are in 2 sizes: 16-bit, denoted in assembly with suffix `.N`, and 32-bit, denoted with `.W`. The instructions operate on 16 general-purpose registers, `R0`–`R12`, `SP` (stack pointer), `LR` (link register), and `PC` (program counter), and some special-purpose ones, most notably flags. The focus in this paper is on 110 instructions sufficient for executing complete regular user-space programs, which perform arithmetic-logical operations, memory accesses, and branches. We do not discuss instructions associated with kernel-space operations, such as those for interrupt handling (`WFI`) or exclusive memory accesses (`LDREX`). We also restrict ourselves to the functional scope of ARMv7-M and do not explore undefined behavior or behavior noncompliant with the architecture.

[1] https://mimuw-distributed-systems-group.github.io/cmemu/

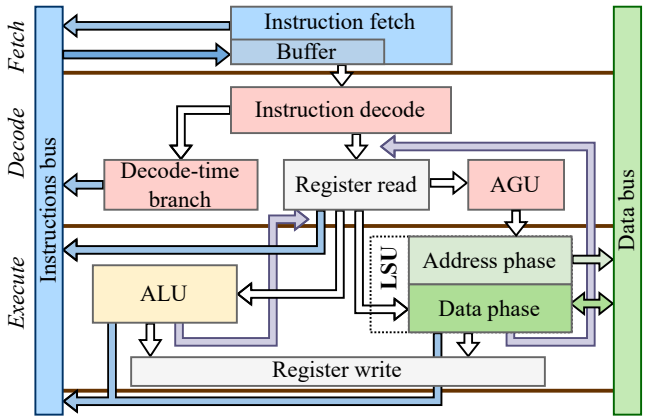


Fig. 1: Simplified overview of the main logical components of Cortex-M3's Core with primary data paths between them.

Cycle-exact modeling requires reflecting in how many processor clock cycles a given chip executes a sequence of instructions. Timings of instructions and other operations of Cortex-M3 are not specified by ARMv7-M, but they depend on the processor's design.

A. Overview of Cortex-M3 Design

In a nutshell, Core is the main component executing a program on Cortex-M3 (see Fig. 1). It features a 3-stage pipeline: *Fetch-Decode-Execute*. Fetch loads program bytes from memory into a 3-word buffer. Decode translates them into control signals for operations and their arguments, performs *decode-time branches*, and generates in the *Address Generation Unit* (*AGU*) addresses for memory operations. Execute performs the operations and writes their results to registers or memory: data manipulation is done by the *Arithmetic and Logic Unit* (*ALU*), memory accesses by the *Load Store Unit* (*LSU*).

Cortex-M3 communicates with memory over the AMBA 3 AHB-Lite protocol [37], which handles each transfer in two phases: an *address phase* (the address of the data is provided), and a *data phase* (the data are transferred). Transfers can be *pipelined*, that is, a bus can serve simultaneously the data phase of one transfer, and the address phase of the next one. Accordingly, load-store instructions in Cortex-M3 are executed in two sub-stages that correspond to the phases and can operate simultaneously for consecutive instructions, resulting in effectively simultaneous execution of *pipelined* instructions.

Debug components available in Cortex-M3 vary between microcontrollers, but almost all feature the *Debug Watchpoint*

and Trace (*DWT*) unit with performance counters. Relevant to our work are: total processor clock cycles (`CYCCNT`), pipeline stall cycles attributed to load-store operations (`LSUCNT`), other stall cycles (`CPICNT`), and cycles during which two instructions are executed simultaneously (`FOLDCNT`).

B. The Challenge: Timings

The above description, extracted mainly from the documentation [38], can support only designing basic abstractions for the timing model. Devising the timings presents challenges.

First, not only is the execution duration of some instructions variable, but also multiple instructions in the pipeline may interact with each other. For example, a `UMULL R0, R1, R2, R3; ADDS R4, R0` sequence involves a *read-after-write* (*RAW*) *hazard*, as the value of `R0`, the lower 32 output bits of the 64-bit multiplication, is used as an input of the next instruction. The hardware may resolve the hazard by *stalling the pipeline* (i.e., suspending its progress) or *value forwarding* to avoid the pipeline stall, or the hazard may not manifest itself with multi-cycle instructions.

Second, in Cortex-M3 there are mechanisms that can be neither controlled nor queried, like fetching instructions from memory. Some decode-time conditional branches, such as `BEQ`, cause the processor to speculatively load the target instruction. This requires special handling that interrupts the normal Fetch's operation. Furthermore, if the branch's condition is not satisfied, restoring the fetching of subsequent instructions may incur additional pipeline stalls, depending on the Fetch's state the moment the condition is determined.

Third, instruction fetches and load-store operations access memory, and thus their timings depend on the microcontroller's memory subsystem. In particular, program code is typically stored in NOR Flash memory that is slower than the processor, so accesses require suspending the pipeline progress, thereby introducing *wait states*. To limit wait states, chips feature buffers and caches, and the processor communicates with memory modules via multiple buses (e.g., separate buses for instructions and for data). Simultaneous transfers on the buses may lead to *bus contentions* (e.g., `LDR` loading a constant from a literal pool `.LTORG`), and the arbitration depends on the exact cycles in which Core issues the transfers.

Intuitively, the smaller the tolerated error, the more phenomena have to be accounted for, which for cycle-exact emulation implies (virtually) all of them: even small errors can be amplified, for instance, through repeated execution (e.g., a for-loop), or by causing a divergence in the cache's state (whose logic features timing-sensitive corner cases). Moreover, Cortex-M3 performs several operations concurrently and many involve multiple hardware components. This severely complicates the reasoning as one cannot look at the features of Cortex-M3 in isolation, and they can mutually obscure their effects.

C. Implications for Cycle-Exact Modeling

To summarize, cycle-exact emulation requires modeling various *design-level aspects* of the microcontroller: its microarchitecture. As a consequence, while large portions of our model ultimately describe any Cortex-M3 microcontroller, one has to tune it to a specific device. We target Texas Instruments CC2650 [39] (see also Fig. 2). It features a Cortex-M3 (rev. *r2p1*) clocked at 48 MHz, 20 KiB of system RAM (*SRAM*), and the *Versatile Instruction Memory System* (*VIMS*), combining 128 KiB of *Flash* and its line buffer with 8 KiB of static RAM, which can operate as a random replacement cache (*Cache*) or a general-purpose RAM (*GPRAM*). TI CC2650 is relatively popular and available in some large testbeds [40], has no published hardware sources, and, like typical similar chips, lacks advanced tracing components. Compared to most of them, however, it has a sophisticated memory subsystem, which we model almost in full: we focused on programs with code in Flash or GPRAM, data in all memories, the line buffer and Cache on or off, and the in-Core store buffer disabled.

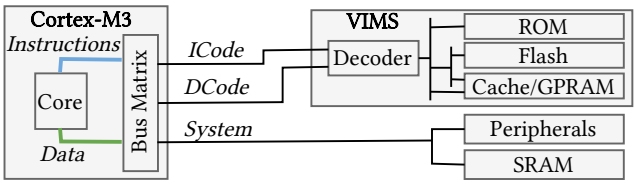


Fig. 2: Simplified overview of the TI CC2650 microcontroller: the main components and memory buses (adapted from [39]).

Following our objective of in-code measurements, the built-in performance counters are the only way we inspect the microarchitecture. Crucially, our goal is *not* to reverse engineer the hardware but to build *abstractions* that mimic its timing: a model that given a program outputs its execution time.

III. Preliminary Techniques

A common way to bootstrap a processor model is to review publicly available documentation [6], [7], [15], [17]–[19], [32]. The documents outline facts essential for developing abstractions of the device's internal architecture and include details on certain device operations. Nonetheless, they offer limited information required to accurately determine the timings of even basic features. For instance, `UMLAL` (multiply & accumulate) is documented to execute in 4–7 cycles, but without precise rules [38]. Moreover, the documentation may contain errors: we found values for which `UMLAL` executes in 3 cycles.

Therefore, it is necessary to empirically measure executions on a real device. For timing models, the most popular approach is to use built-in performance counters [8], [14], [18], [33]. Usually, model development involves measuring individual instructions and handcrafted programs [16], [18], although the process can be supported with tools to generate random ones (see Section IV-A). Execution on the device requires a systematic approach to ensure valid results. To this end, software-hardware tools are developed that automate setting up and priming the device (for a consistent initial state) and executing the measurements (incl. recording counter values). However, as we repeatedly demonstrate in this paper, measuring timings without sophisticated reasoning is highly insufficient.

The process of developing a model commonly involves progressively implementing it as an emulator, and repeatedly validating it against the hardware as a step of the development methodology [14], [32], [33]. We implement our model in the Rust programming language; the emulator, called *CMEmu*, can run the same program binaries as our target microcontroller. In existing work, mainly programs imitating typical workloads (benchmarks) and exercising particular features (microbenchmark) are employed for the validation [6]–[9], [32]–[34]. However, as we argue in Section IV-A and Section IV-E, such programs exercise only a limited set of features of a processor.

IV. Techniques Enabling Cycle-Exactness

We followed the above techniques, but they fell short of supporting the development of a cycle-exact model. To achieve and ensure the desired accuracy, we devised novel ones.

A. Generating Random Fully-Featured Programs (CMGen)

Handcrafted programs hardly explore features unknown to their authors but present in the processor, whereas typical workloads (benchmarks) execute a nondiversified set of instructions: we traced the Embench benchmarks [41] to learn that they exercise 92 out of the 110 instructions, 12.7% of all unique allowable pairs of succeeding instructions, and 0.4% of all unique allowable triplets. As a consequence, such programs cover only a small subspace of the possible internal states of the microcontroller. There are tools that can increase the coverage by employing random bytestreams and fuzzing [42], [43], generating random instruction sequences [14], [35], or systematically generating individual instructions [44]. We used one such tool in our evaluation (see Section V), but it was unable to verify complex instruction interactions. We argue that for a cycle-exact model, validation against full-length fully-featured programs is essential. A tool by Corno et al. [45] generates programs with complex control flow, but it was implemented for a DLX architecture and employed an evolutionary approach μGP [46] to ensure the coverage of a target. We are aware of no tool dedicated to ARMv7-M capable of generating instructions with peculiar semantics like `IT` and `TBB`, generating possibly diverse programs, and supporting not only the validation, but also the development of an emulator.

To bridge that gap, we introduce CMGen, which generates any-length correct ARMv7-M assembly programs (i.e., they terminate and do not cause exceptions or undefined behavior). The programs are generated in blocks, and each block is generated backwards to facilitate respecting instruction restrictions (e.g., alignment, allowed registers, input values) and ensuring predictable branches and safe memory operations. Moreover, CMGen strives to maximize the randomness of the programs: it selects instructions and their arguments at random, although it promotes some of the latter to induce interactions between instructions. When it is necessary to prime an instruction's arguments to ensure the correctness of the program, CMGen mixes the prerequisites with other preceding instructions. To do so, it tracks registers and their values, and whether they act as a source or destination of instructions. Thanks to that, the programs contain possibly many features, including complex dependencies, conditional execution, and code reuse. Finally, the set of instructions is precisely configured, and always only correct programs are generated, which we argue to be indispensable for gradually managing the scope when incrementally developing a cycle-exact model.

Finally, when our emulator incorrectly mimics the execution time of a generated program, CMGen can automatically identify problematic fragments by gradually reducing the program source (a ca. 1 MB assembly file) and reevaluating it. To this end, CMGen leverages the program structure to efficiently rule out irrelevant instructions while preserving the correctness of the program. The process progresses both at block- and instruction-level, but instead of straightforward removal, code and data are replaced with blank spaces (in ARMv7-M zeros encode `MOVS.N R0, R0`). The sizes of blank spaces are set to their original sizes modulo a parametrized value, to find the relevant alignment and distances (e.g., 0 removes the fragment altogether, 4 preserves word-alignment, and ∞ preserves all memory addresses). Two search strategies guide the program reduction: a bisection is used to determine fragments that must be preserved, and an evolutionary approach is employed to identify fragments that may be jointly eliminated.

To illustrate the importance of CMGen for the development of a cycle-exact model, consider multiplication instructions: `UMULL` and `MLA` (see Listing 2). For some values their Execute lasts 3 and 2 cycles respectively, and 5 cycles for a sequence `UMULL; MLA`. However, a program generated by CMGen highlighted that the sequence is executed 1 cycle faster when there is one particular register dependency—the upper bits of the `UMULL`'s result are the accumulate value for `MLA`—and that the same effect occurs when there is one more instruction with the dependency (i.e., `UMULL; MLA; MLA`). CMGen's automatic identification of problematic fragments supports even more convoluted cases: for instance, CMGen reduced a program to a sequence of five branching instructions resulting in accesses to a particular cache line, which highlighted an issue in an initialization of the cache in some of our programs.

Finally, we envision much broader applicability of CMGen. For instance, by generating programs with relaxed correctness rules and verifying them with CMEmu, which strictly follows the ARMv7-M specification, we found and reported a few bugs in GNU Binutils (bugs no. 27066, 27065, 27099, and 27096). As future work, CMGen could be extended with a satisfiability solver to generate peculiar register dependencies with strict requirements (e.g., as in Listing 2).

B. Observing Unobservable ("Gadgets")

The performance counters of Cortex-M3 provide only cumulative, top-level information about an execution. For instance, it is directly measurable with `CYCCNT` that executing `ADD RA, RB` from the 0-wait-state GPRAM lasts 1 cycle. Executing the instruction from the 2-wait-state Flash with the line buffer and cache disabled lasts 3 cycles. This is due to pipeline stalls, which is confirmed by the `CPICNT` counter; however, it is not directly observable whether Fetch

or Execute of the instructions is prolonged. Additional insights into the processor could be obtained through tracing modules (e.g., ETM [47]) or energy leakage (e.g., with electromagnetic probes [18], [19]). However, this would violate our assumption of only in-code measurements, and even then not all details essential for a cycle-exact model could be observed directly.

To learn about unobservable details we propose *gadgets*: parameterized modifications to the program that help enforce particular conditions and uncover the processor's internal state through observable side effects. In the above example, as the gadget, one can use a 32-bit arithmetic-logical instruction that introduces no dependencies, and measure the execution times of `GADGET; ADD RA, RB`, selecting either an addition, a division, or a multiplication with appropriate values to control how many cycles (n) the gadget instruction itself is executed. By finding that for $n = 1$ to 5 the execution times of the sequence were the same, we can deduce that Fetch of the instructions is prolonged because increasing the cumulative duration of Execute does not increase the overall time.

In our work we employ many kinds of gadgets: `NOP` sequences to offset code alignment, multi-cycle instructions to stall the pipeline and fill the Fetch's buffer, `ADD.W` sequences to drain the buffer, branches to the following instruction to introduce bus conflicts, and `LDR`/`STR` to affect buffers and bus arbiters, to name few. Of particular interest are gadgets that behave as no-ops, but may introduce various side effects facilitating detection of emulation errors. Moreover, not only do the gadgets constitute a reasoning technique themselves, but they also support more sophisticated reasoning methods (Sections IV-C and IV-D) and can enforce a particular state of the processor necessary for further analysis (Section IV-E).

C. Counterfactual Reasoning

The intuitive abductive reasoning—accepting the simplest explanation that matches observations—quickly bootstraps a model, but risks overgeneralized conclusions and accumulating errors with subsequent features. For instance, consider measuring whether two consecutive instructions are pipelined by comparing execution times of these instructions individually and as a sequence (a method used e.g. by Barenghi et al. [18]):

$$CYCCNT(\texttt{I1}) + CYCCNT(\texttt{I2}) \stackrel{?}{=} CYCCNT(\texttt{I1; I2})$$

For the sequence `ADDS RA, RB; ADD RC, RD` both times are always equal for any combination of registers, so one may conclude that any RAW hazard here is resolved by value forwarding to Execute. Executing the sequence `LDR RA, [RB]; LDR RC, [RD]` that loads data from SRAM takes 1 cycle less than the instructions individually unless there is a data-address dependency (`RA=RD`), and it is documented that these instructions are internally pipelined in Execute and their transfers are pipelined on the bus. In contrast, for a similar sequence `STR RA, [RB]; LDR RC, [RD]` both times are always equal. Although inferring that `LDR` cannot be pipelined after `STR` in Execute would initially improve the model's accuracy, for longer sequences this (wrong) conclusion would lead to puzzling contradictions.

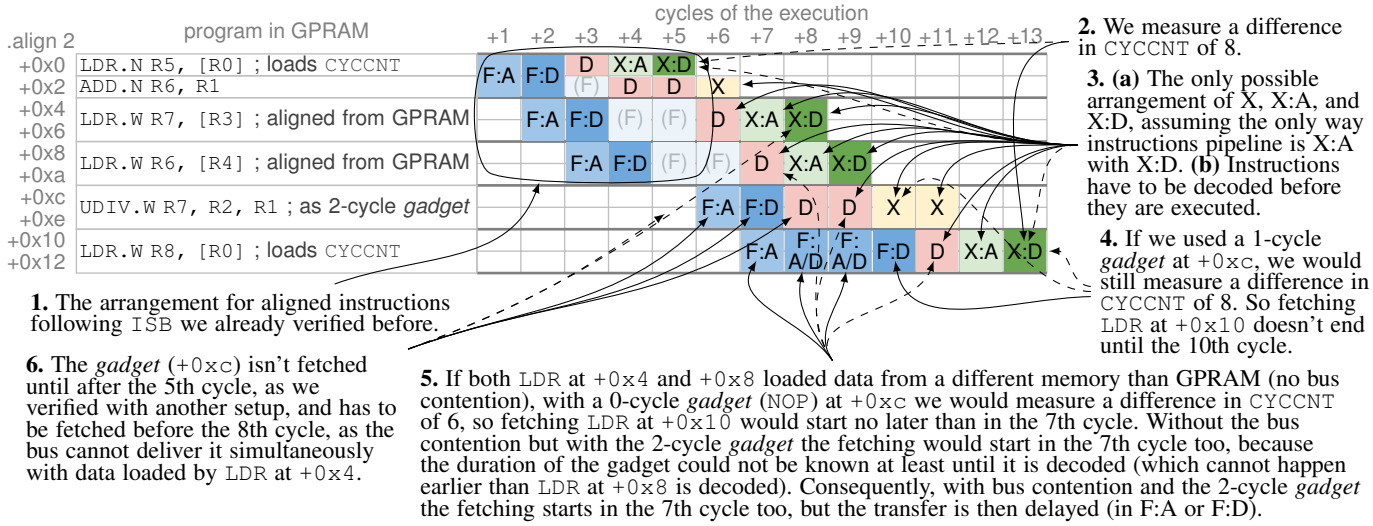
To avoid such pitfalls, we propose moving away from abductive reasoning and instead applying *counterfactual-like reasoning*: for each unknown feature considering all hypotheses to be equally plausible and exploring their consequences to design measurements that refute incorrect ones. These measurements should not involve assumptions about other unverified features; otherwise, the dependency should be noted.

To illustrate, for `STR; LDR`, we considered models where the instructions were pipelined in Execute: always, never, or depending on features such as instruction variants, involved registers, and the pipeline's state. To refute a hypothesis, after the `LDR` we added a decode-time branch as a gadget, so the side effects of the branch would occur when the `LDR` advanced to Execute. For transfers to SRAM we measured different timings of the side effects depending on the variant of `STR` (`STR RA, [RB, #IMM]` vs. `STR RA, [RB, RE]`), indicating that the instructions were pipelined for some variants. Therefore, we ruled out the *never-* hypothesis, and the *always-*hypothesis assuming the memory subsystem cannot identify instruction variants, and thus it cannot block pipelining in Execute for some variants. On Cortex-M3's external buses, on the other hand, in our model loads are never pipelined after stores: for transfers on the same bus it is due to write buffers, on different buses due to bus timing constraints.

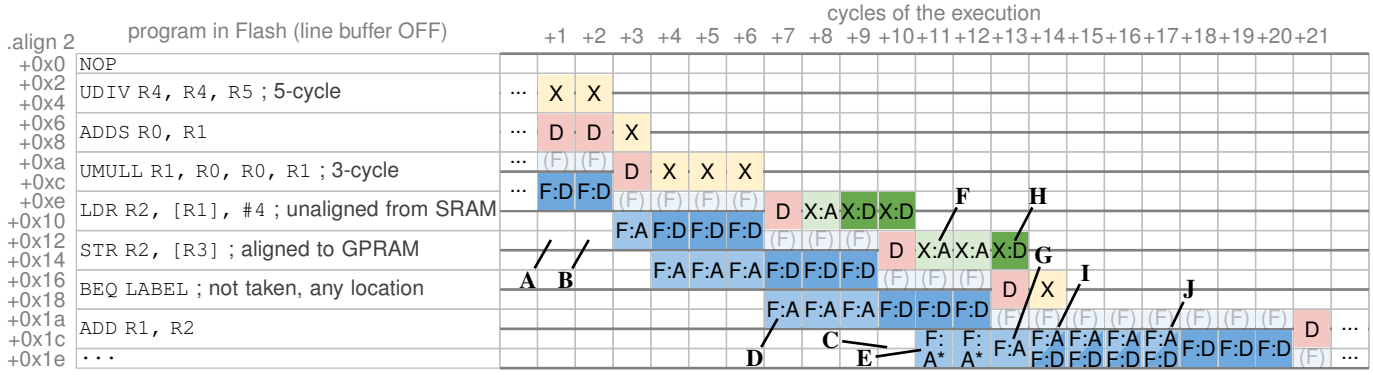
D. Reconstructing Execution With Constraint Solving

When developing a cycle-exact model, it is essential to determine the instruction handled in each cycle by each pipeline stage, but this is not reported by the device. Therefore, we propose a technique to reconstruct the execution of a given program by approaching it as a constraint satisfaction problem. When the execution is visualized on a grid like Fig. 3a, the process is similar to solving a Sudoku puzzle: first, one fills in what stems directly from verified rules of the model (as exemplified in step 1 in Fig. 3a) and direct observation like a difference in `CYCCNT` (step 2); then one tries to fill in the blank spots. To that end, one extracts constraints from all already filled slots and the rules (step 3a), trying to fill more and ultimately find a satisfying arrangement. Importantly, one does not proceed cycle-by-cycle: information on further cycles can be used to reason about previous cycles (step 3b). Usually, the reasoning is supported by adding gadgets (steps 4 and 5) and modifications (steps 5 and 6) to the program to introduce additional constraints.

We attempted to automate the reconstruction using Z3 [48], but that required encoding as formulas increasingly many rules and ultimately would have transcribed the entire emulator, thereby being infeasible. In particular, it is unnecessary to reconstruct every execution: we primarily focus on skillfully designed programs to get new insights into operations of Cortex-M3. To illustrate, Fig. 3a is inconsistent with instruction fetches having a strictly higher priority than data loads (cycles +8 and +9), but does not assume that—thus it refutes such a hypothesis. As a variation of this technique, to prove a hypothesis is inconsistent with other rules, we show it leads to no correct reconstruction. For a given hypothesis a program



(a) A reconstruction of processor's pipeline operations illustrating the Sudoku-like reconstruction of the program execution. The numbered comments illustrate the step-by-step reasoning. The dashed arrows point at premises, the solid arrows at conclusions of each step.



(b) A reconstruction of processor's pipeline operations illustrating an exhaustive test and the Fetch behavior.

Fig. 3: Illustrative reconstructions of the processor's pipeline operations in each cycle of program execution. The colored cells denote individual pipeline operations: **F:A** and **F:D** – address and data phases of Fetch, **D** – Decode, **X** – Execute of an arithmetic-logical instruction, **X:A** and **X:D** – address and data phase of Execute of a load-store instruction. **F:A*/X:A*** mean that a transfer has not yet been forwarded to ICode/DCode, **(F)** denotes that an instruction is loaded to the Fetch's buffer.

is designed by considering what execution should occur to support or refute it, and then what program can enforce such an execution, leveraging the already verified rules.

When an execution cannot be reconstructed using gadgets because introducing modifications to the program alters the explored behavior, we propose upgrading the technique: executing the program multiple times, each time scheduling an interrupt more cycles into the program. By reading register values, performance counters and the return address in a custom interrupt handler, one gets snapshots of the processor state. The snapshots, however, do not reveal the execution directly, as the timing of interrupt handling is affected by aspects such as bus congestion and currently executed instructions. Thus, to properly interpret the snapshots, first the reasoning with gadgets is required to determine when the effects of an instruction are visible in an interrupt handler, and when the effects of a handler are visible on an instruction. To illustrate, consider a program with two subsequent load instructions. If scheduling an interrupt in the $i+1$-th cycle instead of the i-th cycle causes the handler to read a smaller value of LSUCNT, this means that the loads are pipelined in the $i+1$-th cycle.

E. Exhaustive Tests

Programs that are handcrafted, randomly generated, or represent real-world workloads have an inherent limitation: they do not exhaustively explore a given aspect of a microcontroller operation. Moreover, even if they reveal an error in the model, fixing it based on only a few observations risks overfitting, particularly when it is related to an extremely rare behavior.

To prevent that, we propose extending the set of programs used to validate a model with *exhaustive tests*: for a problematic case, generating millions of new programs, trying to cover all possible interactions of the underlying feature of the microcontroller. For instance, in order to validate interactions of load-store instructions, we enumerate all their variants (e.g., w/ & w/o writeback, one & multiple registers), encodings, and

arguments (e.g., SP is handled in a special way [36]), we vary the accessed memories and access alignment, and we interleave them with various instructions. Furthermore, we employ gadgets (Section IV-B) to validate the instructions in a magnitude of contexts (e.g., code location, code alignment, Fetch state, conditional execution, features for indirect observations). The scalability of our in-code measurement approach makes the exhaustive tests feasible: in particular, we executed them on dozens of devices of the 1KT IoT testbed [40]. Emulating such a set of programs correctly gives high confidence in the correctness of the corresponding part of the model.

To illustrate the advantage of exhaustive tests, consider a program as in Fig. 3b. It employs a few gadgets: NOP forces instructions not to be word-aligned; UDIV, ADDS, and UMULL control the state of Fetch (new transfers are not started in *A*, *B*, and *C*, because its buffer is full; one is opportunistically started in *D*); LDR executes an unaligned transfer over a different external bus than fetching (see Fig. 2). In effect, a bus congestion occurs due to Core simultaneously prefetching instructions from Flash from the address +0x1c (*E*) and executing the STR to GPRAM (*F*). When the DCode bus proceeds to the 1-cycle data phase (*H*), the prefetch transfer is carried out on ICode (*G*) after a prolonged address phase on the internal bus. In that cycle (+13), a conditional branch triggers an attempt to speculatively fetch the target instruction. Since the data phase of the ongoing transfer has not started yet (*G*), the prefetching logic attempts to substitute the transfer's address with the branch target, but the memory subsystem carries out the transfer on ICode from the original address. In the next cycle that transfer proceeds to the data phase (*I*), and since the branch was not taken, the prefetching logic starts fetching from +0x1c as it previously successfully prefetched from +0x18. It is not aware that the address substitution did not succeed so it issues a new transfer from +0x1c (*I*), whereas the data from the ongoing transfer is disregarded when it finishes (*J*). Consequently, +0x1c is prefetched twice, regardless of the address of the branch target. We hypothesize it is so because full AHB-Lite compliance was introduced as an option only in the latest revision of Cortex-M3 and thus was implemented not in the prefetching logic but in another subcomponent. Even though this behavior surfaces only in particular setups, exhaustive tests were able to capture it.

F. Automatically Characterizing Misemulated States

The exhaustive tests are suitable for revealing errors related to microarchitectural details of Cortex-M3. However, as the details are low-level, their effects are easily obscured by higher-level features. As a result, such errors can manifest themselves as misemulation of programs that seemingly have nothing in common. Walker et al. [33] employ clustering and correlation analysis of values of real and emulated performance counters to identify sources of errors in the model. While inspiring, from the perspective of cycle-exact modeling their method has a few limitations: it evaluates a suite of 65 programs whereas we advocate massive tests (Section IV-E); it benefits from 68 counters of their hardware while Cortex-M3 features 6; it identifies misemulated features but we would like to narrow that down to precise internal states.

```
1 B.W label                 ; goto label
2 ...
3 label:
4 STR.W R1, [R2, R3]        ; *(u32*)(R2+R3) = R1
5 NOP.N                     ; do nothing
```

Listing 1: Program illustrating pecularities of Fetch and IT.

As a consequence, we propose an alternative approach leveraging the design of the exhaustive tests (Section IV-E). For each program, we annotate hardware and emulation results with a list of the alternations that define its state and control flow. We apply to the dataset manual feature engineering based on our understanding of Cortex-M3, and automated associative learning to extract problematic alternations and thereby characterize those cases, ultimately identifying states of the processor in which the model is inaccurate. To this end, we fit gradient boosted trees and use TE2Rules [49] to extract a rule-based model. As it is hardly understandable, we apply post-processing transformations—agglomerative clustering and logic- and data-based simplifications—to obtain a logical formula in the form of an intuitive tree of conditions (conjunctions in nodes, alternatives in subtrees). We did not use explainable artificial intelligence (XAI) toolkits, as they are designed to operate on incomplete and noisy data, therefore producing less precise descriptions.

To illustrate the benefits of this approach, consider a rule that we distilled from a series of exhaustive tests and their analyses: 5 half-words after IT (the if-then instruction) there is a 16-bit NOP (or a skipped instruction) preceded by a load-store instruction, the NOP can be pipelined with the load-store instruction, and the NOP is the only instruction in the Fetch's buffer (e.g., due to slow memory). That rule characterized a processor state when the NOP was pipelined in our model but not in the hardware, and it would be extremely hard to derive it so precisely with solely manual reasoning.

The holistic explanation of that behavior was deduced from a program as in Listing 1 generated by CMGen (Section IV-A). We noticed that a branch there was encoded as a 32-bit instruction whose second half-word was an encoding of IT, so binary-wise the program resembled the distilled rule. Usually the NOP instruction is pipelined with a preceding load-store instruction, except when a 16-bit NOP is followed by a 16-bit IT instruction, and they are decoded together—then the former is no longer pipelined. We hypothesized that in the programs when the instruction following NOP was not fetched yet, the next buffer entry was still occupied by the already executed IT or half of the already executed branch that encoded IT—the entry was logically empty but interacted as if IT was there.

G. Exploration with High-Performance Computing

Components of a microcontroller interact with each other, so it is necessary to tune the model as a whole. To this end, Adileh et al. [32] parametrize the model and apply

```
1 UMULL.W  R0, LR, R1, R2  ; u64 LR:R0 = R1 * R2
2 MLA.W    R3, R4, R5, LR  ; R3 = R4*R5 + LR
3 BX.N     LR              ; goto *LR
```

Listing 2: Code triggering a hardware bug in Cortex-M3.

an optimization algorithm to search for the best parameter combination. We employed this idea in our work to evaluate mutually-dependent hypotheses arising from counterfactual reasoning (Section IV-C): we implemented the hypotheses in our emulator as parameters, and we applied a Bayesian optimization minimizing the number of incorrectly emulated cases.

Exploring the combinations is a computationally intensive task since a model contains dozens of parameters: Adileh et al. restrained themselves to a set of micro-benchmarks that was simulated in subseconds, but the entire exploration still ran for 2 days on a commodity CPU. However, a cycle-exact model requires much more scrutiny (see Section IV-E). Therefore, we evaluated our model against millions of programs of the exhaustive tests. To make that feasible, we employed an HPC cluster (Intel Xeon E5-2697 v3 CPUs providing ca. 500 cores). That made it possible to evaluate the programs in 15 minutes and complete the exploration in a few hours despite the scale. Moreover, we were able to run the exploration repeatedly as the model progressed. Overall, for exploration and evaluation (Section V) we used 30,000 CPU-hours.

H. Errata Notices

Reviewing public documentation is a common technique (see Section III), but it describes few internal details. However, we argue that some insights into the microarchitecture can be obtained from seemingly irrelevant literature. Errata notices for previous Cortex-M3 revisions are an important example, since bugs found after the processor's initial release were likely patched with workarounds instead of being thoroughly corrected (which would require optimizing and verifying the processor again [50]). To illustrate, Errata Notice 377494 [51] lists a bug where decoding `BX LR` (procedure return) after pipelined load-store instructions could start fetching an incorrect instruction due to a RAW hazard. We deduced that the problem was solved using information easily available in the design to detect a relatively simple necessary condition for the RAW hazard, resulting in false positives. That explained why our initial model of `BX LR` based on an intuitive, more precise condition was inconsistent with measurements.

The peculiarity of `BX LR` prompted us to verify RAW hazards introduced also by other instructions. In particular, CMGen had found that `UMULL; MLA` could be pipelined, even though that was not documented (Section IV-A). When analyzing the timings of such instruction sequences, we found that a program like in Listing 2 caused incorrect functional operation of Cortex-M3: not only did it fetch an instruction from a stale address (i.e., the value of `LR` from before `UMULL`), but it also executed that instruction or otherwise entered an inconsistent state. The discovery was reported to ARM, who confirmed the bug in hardware and issued Erratum #3922886.

V. Evaluation

We evaluated our model's accuracy by employing CMEmu to emulate a suite of programs, and for each test case we compared the number of emulated cycles against the hardware. The suite consists of three main categories: short, random, and real-world programs. For robustness of the evaluation [32], we employed the machine learning approach of splitting the suite into two sets: training (development) and testing (which we did not use to identify emulation errors). CMEmu **perfectly** emulates timings of all these programs; several gigabytes of binaries, an order of 10^{12} cycles in total.

1) Development tests: We employed over 500,000 handcrafted test cases, 900,000 lengthy programs generated with CMGen (Section IV-A), and 500,000,000 extensive test cases (Section IV-E). Moreover, we used the CoreMark [52] benchmark and micro-benchmarks targeting Cache.

2) Real-world tests: To estimate the accuracy of a cycle-accurate model usually a few benchmarks [6]–[10], [32], [34] or typical workloads [13], [53] are employed. We followed that idea but selected a wide suite: Embench [41] (a diverse benchmark incl. algorithms, compression, state machines, etc.), STREAM [54], BenchCouncil IoTBench [55], TI MSP430 microbenchmarks [56], and selected libc tests [57]; in addition, cryptographic primitives from embedded libraries: Mbed TLS, OpenSSL, TweetNaCl, wolfSSL, and built-in TI CC2650 ROM. The selection was guided by our platform's capabilities (limited memory, no external probes, etc.), licensing of the benchmark (free/open-source), and our model's scope (no multithreading, no file system), although for some benchmarks we had to set selected parameters (e.g., buffer and key sizes) outside their recommended range to fit the memory. To further diversify the suite, as most of the programs are written in C and thus expose CMEmu to compiler-generated assembly and C programming practices (only some additionally utilize hand-written assembly), we compiled each program with multiple GCC optimization levels: `-Os`, `-O0`, and `-O3`.

In total, the suite consists of 644 test cases—55 unique binaries compiled at 3 optimization levels and executed in 4 memory configurations (16 test cases do not fit in the memory)—each executing from 51 to a billion cycles.

3) Instruction coverage: Real-world programs exercise few features supported by our model (see Section IV-A), so we employed EXAMINER [44] to successfully validate our model against 33,951 single-instruction test cases it generated. Moreover, we used our CMGen (Section IV-A) to generate 460,000 test cases we did not use in the model development. They covered all 110 instructions supported by our model, all unique allowable pairs of succeeding instructions, and 99.97% of unique allowable triplets of succeeding instructions.

4) PC trace correctness: We additionally verified the correctness of an entire *PC trace*—a trace listing the instruction executed in each cycle. We chose OpenSSL AES, and due to the lack of advanced tracing components (ETM) in TI CC2650, we obtained the reference PC trace using an alternative, lengthy technique [47]. The emulated and reference

Programs category	Test cases	Cycles	Speculative Fetch		`STR; LDR`		`UMULL; MLA; MLA`		Phantom `IT`		Final model	
		in total	cases	error	cases	error	cases	error	cases	error	cases	error
Embench	252	1e11	251	2e-2	140	5e-4	0	0	0	0	0	0
other benchmarks	168	7e7	136	2e-2	18	3e-4	0	0	0	0	0	0
libc tests	72	4e8	72	3e-3	60	1e-5	0	0	0	0	0	0
cryptography	152	2e10	152	1e-2	101	9e-5	12†	8e-8	0	0	0	0
CMGen test set	460,000	6e9	459,845	3e-2	137,420	4e-4	2	2e-8	1024	1e-6	0	0

TABLE I: Results of the ablation study. The programs are grouped into categories, but we compared emulated cycles against the hardware for each test case individually. For each variant of the model we report a number of failed test cases (i.e., non-zero error) and a total relative absolute error: $\frac{\sum |emulation-hardware|}{\sum |hardware|}$. †– All cases execute the same built-in ECC code from ROM.

traces matched, demonstrating that in some applications it is feasible to substitute a real device with CMEmu.

5) Performance: Emulation speed was a major consideration in CMEmu development. As a result, although it is cycle-exact, it achieves performance comparable with cycle-accurate emulators [23], [24]: on a PC with an Intel Xeon W-1290 CPU, it emulates CoreMark with a rate of approximately 0.9 million Cortex-M3's clock cycles per real-time second.[2] For reference, ARM's IP Explorer [31] hardware-source-based model emulates at ∼10K cycles/second; gem5 [5] with the Minor CPU model emulates an ARMv8 system (Cortex-M3 is unsupported) at ∼0.7M cycles/second.

A. Ablation Study

For an older low-power microcontroller, ATmega128, there are two major emulators: hardware-source-based AVR Simulator [58] and non-hardware-source-based Avrora [26]. The former is slow but cycle-exact, the latter faster but we measured a timing error of 0.4% on CoreMark. Before applying the techniques of Section IV our model exhibited a comparable error, so to quantitatively assess the impact of our techniques, we performed an ablation study: we selectively impaired the final model and evaluated it against the programs of our testing suite. We considered 4 impairments: missing speculative fetching with single-cycle instructions (related to Section IV-E), a "never pipeline" rule for `STR; LDR` (Section IV-C), no special handling for `UMULL; MLA; MLA` (Section IV-A), and not modeling the "phantom `IT`" (Section IV-F). We evaluated also various impairments of the memory subsystem, but they resulted in regressions comparable with the `STR; LDR`.

Judging from the results (see Table I), we expect the typical methodologies for developing cycle-accurate models of embedded platforms to be inherently limited to approximately 0.1% overall error validated with popular benchmarks. The cycle-exact modeling techniques enable reducing the error to virtually 0%, although it is impossible to be certain that an empirically synthesized emulator is fully cycle-exact. Therefore, striving for cycle-exactness has to be an ongoing effort, and we anticipate our model may be incorrect on some very atypical instruction sequences such as Listing 2, whose systematical verification requires devising further techniques.

[2]At an additional performance cost, CMEmu may produce a per cycle description of a program execution: the pipeline state, values of registers and flags, and ongoing bus transfers. It can also generate a highly detailed microarchitecture-level log and visualize the execution similarly to Fig. 3.

VI. Discussion and Outlook

We showed that, contrary to common beliefs, *it is feasible to devise a cycle-exact model of a modern microcontroller without access to its hardware sources, or even physical access to the device, relying instead solely on in-code timing measurements and publicly available information*. CMEmu, to the best of our knowledge the first implementation of such a model, exhibits *no timing error on trillions of emulated cycles of a diverse test suite*. Although it cannot completely replace hardware-source-based emulators, especially for insights other than execution timing, we believe that CMEmu opens up a range of diverse research opportunities. Furthermore, we are currently modeling other subsystems of the microcontroller and integrating CMEmu with a network simulator to enable research on low-power wireless communication.

While this paper demonstrates modeling a particular microcontroller, we argue that the applicability of our techniques is not limited to that device. By design, none of them are constrained to TI CC2650 or Cortex-M3, and they require no advanced tracing components or invasive instrumentation. In fact, we have already approached modeling other microcontrollers (incl. one with Cortex-M4), and the initial measurements strongly suggest that it is readily feasible. Even a lack of performance counters would not invalidate our techniques, but pose an additional challenge [18], requiring external time measurements and additional reasoning. Lastly, CMEmu itself is organized into modules reflecting typical microarchitectural components, thereby facilitating customization.

Nevertheless, even though the selected processor family is arguably only moderately advanced compared to smartphone processors, the modeling process was a tedious task. Applying the preliminary techniques was a topic of a two-year three-person Master's project, and the following techniques required exponentially more effort. The discovery of a hardware bug in Cortex-M3 and several bugs in GNU Binutils is a good testimony to the meticulousness of the process. Finally, we envision that the demonstrated scalability of measurement acquisition could enable leveraging large ML models to derive a complete emulator rather than only improve its constituents. However, fully automatic exploration of all microcontroller intricacies would necessitate further advancements.

References

[1] Antmicro, "Renode." https://renode.io/

[2] "Open Virtual Platforms." https://www.ovpworld.org/

[3] Wind River, "Simics." https://www.windriver.com/products/simics
[4] N. Binkert *et al.*, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, 2011.
[5] J. Lowe-Power *et al.*, "The gem5 Simulator: Version 20.0+," arXiv:2007.03152 [cs], 2020.
[6] A. Gutierrez *et al.*, "Sources of error in full-system simulation," in *2014 IEEE Int. Symp. Perform. Anal. Syst. Softw. ISPASS*. IEEE, 2014.
[7] F. A. Endo, D. Courousse, and H.-P. Charles, "Micro-architectural simulation of in-order and out-of-order ARM microprocessors with gem5," in *2014 Int. Conf. Embed. Comput. Syst. Archit. Model. Simul. SAMOS XIV*. IEEE, 2014.
[8] S. Schreiner, R. Görgen, K. Grüttner, and W. Nebel, "A quasi-cycle accurate timing model for binary translation based instruction set simulators," in *2016 Int. Conf. Embed. Comput. Syst. Archit. Model. Simul. SAMOS*. IEEE, 2016.
[9] R. Desikan, D. Burger, and S. Keckler, "Measuring experimental error in microprocessor simulation," in *Proc. 28th Annu. Int. Symp. Comput. Archit.* ACM, 2001.
[10] A. Akram and L. Sawalha, "A Survey of Computer Architecture Simulation Techniques and Tools," *IEEE Access*, vol. 7, 2019.
[11] F. Ferrari, M. Zimmerling, L. Thiele, and O. Saukh, "Efficient network flooding and time synchronization with Glossy," in *Proc. 10th ACMIEEE Int. Conf. Inf. Process. Sens. Netw.* IEEE, 2011.
[12] J. Eriksson *et al.*, "COOJA/MSPSim: Interoperability testing for wireless sensor networks," in *2nd International ICST Conference on Simulation Tools and Techniques*. ICST, 2010.
[13] K. Roussel, Y.-Q. Song, and O. Zendra, "Using Cooja for WSN Simulations: Some New Uses and Limits," in *Proc. 2016 Int. Conf. Embed. Wirel. Syst. Netw.*, ser. EWSN '16. Junction Publishing, 2016.
[14] J. Bauer and F. Freiling, "Towards Cycle-Accurate Emulation of Cortex-M Code to Detect Timing Side Channels," in *2016 11th Int. Conf. Availab. Reliab. Secur. ARES*. IEEE, 2016.
[15] D. McCann, E. Oswald, and C. Whitnall, "Towards Practical Tools for Side Channel Aware Software Engineering: 'Grey Box' Modelling for Instruction Leakages," in *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, 2017.
[16] M. A. Shelton, N. Samwel, L. Batina, F. Regazzoni, M. Wagner, and Y. Yarom, "Rosita: Towards Automatic Elimination of Power-Analysis Leakage in Ciphers," in *Proc. 2021 Netw. Distrib. Syst. Secur. Symp.* Internet Society, 2021.
[17] Y. Le Corre, J. Großschädl, and D. Dinu, "Micro-architectural Power Simulator for Leakage Assessment of Cryptographic Software on ARM Cortex-M3 Processors," in *Constructive Side-Channel Analysis and Secure Design*, J. Fan and B. Gierlichs, Eds. Springer International Publishing, 2018, vol. 10815.
[18] A. Barenghi, L. Breveglieri, N. Izzo, and G. Pelosi, "Exploring Cortex-M Microarchitectural Side Channel Information Leakage," *IEEE Access*, vol. 9, 2021.
[19] A. de Grandmaison, K. Heydemann, and Q. L. Meunier, "ARMISTICE: Microarchitectural Leakage Modeling for Masked Software Formal Verification," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 41, no. 11, 2022.
[20] T. Scharnowski *et al.*, "Fuzzware: Using Precise MMIO Modeling for Effective Firmware Fuzzing," in *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, 2022.
[21] X. M. Saß, R. Mitev, and A.-R. Sadeghi, "Oops..! I Glitched It Again! How to Multi-Glitch the Glitching-Protections on ARM TrustZone-M," in *32nd USENIX Security Symposium (USENIX Security 23)*. USENIX Association, 2023.
[22] T. Simunic, L. Benini, G. De Micheli, and M. Hans, "Source code optimization and profiling of energy consumption in embedded systems," in *Proc. 13th Int. Symp. Syst. Synth.* IEEE, 2000.
[23] "Running Trusted Firmware-A on gem5 - Research Articles - Research Collaboration and Enablement - Arm Community." https://perma.cc/39RQ-PQMR
[24] B. Neifert and R. Kaye, "High Performance or Cycle Accuracy?" ARM, Tech. Rep., 2012.
[25] J. Eriksson, A. Dunkels, N. Finne, F. Österlind, T. Voigt, and N. Tsiftes, "Demo abstract: MSPsim – an extensible simulator for MSP430-equipped sensor boards," in *5th Eur. Conf. Wirel. Sens. Netw.*, ser. EWSN '08, 2008.
[26] B. Titzer, D. Lee, and J. Palsberg, "Avrora: Scalable sensor network simulation with precise timing," in *IPSN 2005 Fourth Int. Symp. Inf. Process. Sens. Netw. 2005*. IEEE, 2005.
[27] R. Keil, "Arm empowers MCU software developers to capitalize on IoT potential." https://perma.cc/Z7LX-7PU4
[28] "Measuring performance of programs on the FVP - Arm Community." https://perma.cc/CZ77-4LT8
[29] EEVblog Electronics Community Forum, "ARM Cortex series simulator." https://perma.cc/EW38-Y6V7
[30] "Gem5 queries - gem5-users mailing list." https://perma.cc/9JE5-PB3B
[31] "Arm IP Explorer." https://ipexplorer.arm.com/
[32] A. Adileh, C. González-Álvarez, J. Miguel De Haro Ruiz, and L. Eeckhout, "Racing to Hardware-Validated Simulation," in *2019 IEEE Int. Symp. Perform. Anal. Syst. Softw. ISPASS*. IEEE, 2019.
[33] M. Walker, S. Bischoff, S. Diestelhorst, G. Merrett, and B. Al-Hashimi, "Hardware-Validated CPU Performance and Energy Modelling," in *2018 IEEE Int. Symp. Perform. Anal. Syst. Softw. ISPASS*. IEEE, 2018.
[34] A. Butko, R. Garibotti, L. Ost, and G. Sassatelli, "Accuracy evaluation of GEM5 simulator system," in *7th Int. Workshop Reconfigurable Commun.-Centric Syst.–Chip ReCoSoC*. IEEE, 2012.
[35] B. Black and J. Shen, "Calibration of microprocessor performance models," *Computer*, vol. 31, no. 5, 1998.
[36] ARM, "ARMv7-M Architecture Reference Manual, iss. E.d."
[37] ——, "AMBA 3 AHB-Lite Protocol Specification."
[38] "ARM Cortex-M3 Processor Technical Reference Manual, iss. 0201-01."
[39] Texas Instruments, "CC13x0, CC26x0 SimpleLink™ Wireless MCU Technical Reference Manual." https://perma.cc/8FBR-TZWV
[40] M. Banaszek *et al.*, "1KT: A Low-Cost 1000-Node Low-Power Wireless IoT Testbed," in *Proc MSWiM 21*. ACM, 2021.
[41] "Embench: A Modern Embedded Benchmark Suite." https://www.embench.org/
[42] L. Martignoni, R. Paleari, A. Reina, G. F. Roglia, and D. Bruschi, "A methodology for testing CPU emulators," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, 2013.
[43] V. Herdt, D. Grose, H. M. Le, and R. Drechsler, "Verifying Instruction Set Simulators using Coverage-guided Fuzzing," in *2019 Des. Autom. Test Eur. Conf. Exhib. DATE*. IEEE, 2019.
[44] M. Jiang *et al.*, "EXAMINER: Automatically locating inconsistent instructions between real devices and CPU emulators for ARM," in *Proc. 27th ACM Int. Conf. Archit. Support Program. Lang. Oper. Syst.* ACM, 2022.
[45] F. Corno, E. Sanchez, M. S. Reorda, and G. Squillero, "Code Generation for Functional Validation of Pipelined Microprocessors," *Journal of Electronic Testing*, vol. 20, no. 3, 2004.
[46] G. Squillero, "MicroGP—An Evolutionary Assembly Program Generator," *Genet Program Evolvable Mach*, vol. 6, no. 3, 2005.
[47] M. Matraszek, M. Banaszek, W. Ciszewski, and K. Iwanicki, "FrankenTrace: Low-Cost, Cycle-Level, Widely Applicable Program Execution Tracing for ARM Cortex-M SoC," in *Proc. Cyber-Phys. Syst. Internet Things Week 2023*. ACM, 2023.
[48] L. de Moura and N. Bjørner, "Z3: An Efficient SMT Solver," in *Tools Algorithms Constr. Anal. Syst.*, C. R. Ramakrishnan and J. Rehof, Eds. Springer, 2008.
[49] G. R. Lal, X. Chen, and V. Mithal, "TE2Rules: Extracting Rule Lists from Tree Ensembles," arXiv:2206.14359 [cs], 2022.
[50] S. Furber, *ARM System-on-Chip Architecture*, 2nd ed. Addison-Wesley, 2000.
[51] ARM, "Errata Notice for ARM Core Cortex-M3 / Cortex-M3 with ETM (AT420/AT425)." https://www.ti.com/lit/er/spmz091/spmz091.pdf
[52] EEMBC, "CoreMark Benchmark." https://www.eembc.org/coremark/
[53] M. Asri, A. Pedram, L. K. John, and A. Gerstlauer, "Simulator calibration for accelerator-rich architecture studies," in *2016 Int. Conf. Embed. Comput. Syst. Archit. Model. Simul. SAMOS*. IEEE, 2016.
[54] J. D. McCalpin, "Memory Bandwidth and Machine Balance in Current High Performance Computers," *IEEE Comput. Soc. Tech. Comm. Comput. Archit. TCCA Newsl.*, 1995.
[55] "BenchCouncil IoTBench." https://www.benchcouncil.org/iotbench/
[56] W. Goh and K. Venkat, "MSP430 Competitive Benchmarking," Texas Instruments, Tech. Rep., 2006.
[57] R. Felker, "Libc-testsuite: Correctness and quality tests for libc implementations." https://git.musl-libc.org/cgit/libc-testsuite/
[58] "AVR Simulator," Microchip Technology Inc. https://perma.cc/RA2T-X6WJ