Appears in: Proceedings of the 20th International Symposium on Real-Time Distributed Computing (ISORC 2017)
Toronto, Canada, 16–18 May 2017

An End-To-End Toolchain: From Automated Cost Modeling to Static WCET and WCEC Analysis

Volkmar Sieh, Robert Burlacu[1], Timo Hönig, Heiko Janker, Phillip Raffeck,
Peter Wägemann, and Wolfgang Schröder-Preikschat

Department of Computer Science, Distributed Systems and Operating Systems
[1]Department of Mathematics, Economics · Discrete Optimization · Mathematics

Friedrich-Alexander University Erlangen-Nürnberg (FAU)

***Abstract*—Reliable and fine-grained cost-models are fundamental for real-time systems to statically predict worst-case execution time (WCET) estimates of program code in order to guarantee timeliness. Analogous considerations hold for energy-constrained systems where worst-case energy consumption (WCEC) values are mandatory to ensure meeting predefined energy budgets. These cost models are generally unavailable for commercial off-the-shelf (COTS) hardware platforms, although static worst-case analysis tools require those models in order to predict the WCET as well as the WCEC of program code.**

To solve this problem, we present NEO, an end-to-end toolchain to automate cost-model generation for both WCET and WCEC analyses. NEO exploits automatically generated benchmarks, which are input for 1) an instruction-level emulation and 2) automatically conducted execution-time and energy-consumption measurements on the target platform. The gathered values (i.e., occurrences per instruction, execution-time and energy-consumption per benchmark) are combined as mathematical optimization problems. The solutions to the formulated problems, which are designed to reveal the worst-case behavior, yield the respective cost models. To statically determine upper bounds of benchmarks, we integrated the cost models into the state-of-the-art WCET analyzer PLATIN. Our evaluations on COTS hardware reveal that our open-source, end-to-end toolchain NEO yields accurate worst-case bounds.

I. Introduction & Problem Statement

Time and energy are prime resources for embedded computing systems. On the one hand, embedded real-time systems need to meet deadlines in order to operate accordingly to specifications. On the other hand, today's embedded systems additionally have energy requirements that need to be satisfied.

It requires detailed knowledge on the runtime behavior of the system software to implement efficient resource management for embedded systems. To analyze system software components for their resource requirements, developers have to carry out various analyses. This includes but is not limited to exhaustive timing- and energy-consumption analyses. Determination of time and energy requirements of program code therefore is an expensive and difficult task.

To facilitate the work required for the resource-requirements analysis of system software, time and energy models are commonly used to circumvent time-consuming measurements. However, profound tooling support to generate time and energy models of embedded systems is yet missing.

Execution-Time Analysis. A challenging task for developing real-time systems is the prediction of the worst-case execution time (WCET) of programs. Data about the WCET of program tasks is inevitable in such systems in order to guarantee the schedulability of all program tasks, which means that no task exceeds its deadline.

A widely used technique to determine execution bounds of program code in real-time systems is the Implicit Path Enumeration Technique (IPET) [1]. Its core idea is the formulation of an integer linear programming (ILP) problem from the control-flow graph of the analyzed program. The solution of the ILP yields the maximum possible flow through the graph, which is an upper bound on execution frequencies of basic blocks. For the low-level target-dependent hardware analysis, the IPET requires accurate information on upper execution times for each basic block, which needs to be provided by a fine-grained (i.e., instruction-level) cost model.

Usually, documentation on the timing behavior of processors is not available for commercial off-the-shelf (COTS) hardware platforms. However, even if an instruction-level execution model is provided by the platform's reference manual, this documentation can be incorrect: Atanassov et al. [2] have proven that documented execution times of instructions may include underestimations and demonstrated that for an embedded hardware platform.

Energy-Consumption Analysis. In analogy to WCET analysis, worst-case energy consumption (WCEC) analysis of program code, requires energy-cost models for static analyses in order to guarantee the execution of programs within predefined energy bounds [3]. This is especially crucial for energy-constrained devices for the Internet of Things (IoT).

For these static WCEC analyses, techniques from the domain of real-time systems, such as the IPET, can be adapted. However, a safe determination of WCEC estimates is not possible from an existing WCET estimation by multiplying the execution time with an average power consumption of the system [4]. As a consequence, specialized cost models on energy-consumption are inevitable to precisely predict worst-case energy consumption of tasks.

Documentation on energy-consumption costs is even less available than it is on the timing behavior of embedded target platforms. This is also due to the fact that power consumption

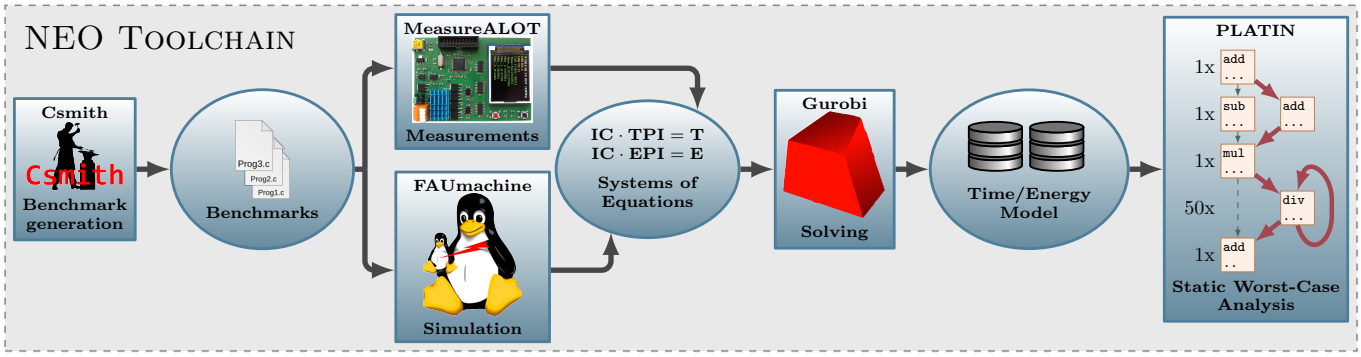

Fig. 1: NEO uses CSMITH to automatically generate benchmarks. The programs are executed on the target platform while time and energy are measured by the MEASUREALOT tool. The binaries are simulated on the virtual machine FAUMACHINE to gather the concretely executed instructions. These counters and the measurements are considered to solve the optimization problem yielding time and energy models of the target platform. These models are used in the worst-case analyzer PLATIN.

is heavily influenced by the actual wiring of each component in the system. The energy-consumption models depend on the static power consumption of peripherals (i.e., sensors) and thus demand for automated measurement-based cost modeling.

Time- and Energy-Model Generation. Exact time- and energy-cost models are the fundamental basis for both static WCET and WCEC analyses. Since manually determining reliable time and energy models is labor-intensive and error-prone, an approach to generate these numbers for time- and energy-consumption analyses of target platforms is necessary.

To solve this problem, we propose the NEO toolchain to automatically generate instruction-level time and energy models for embedded COTS processors where documentation on low-level timing and energy-consumption details is missing. Our approach requires no a priori knowledge on the energetic and temporal behavior of the target platform. The key contributions presented in this paper are fourfold:

1) We present an approach to create accurate instruction-level execution-time and energy-consumption models based on automatic benchmark generation exploiting mathematical optimization methods.
2) We demonstrate the practical applicability of the generated models through an integration of the models into the generic, open-source WCET-analysis tool PLATIN.
3) We present comprehensive evaluations of the generated cost models as well as static worst-case analyses with these models validating the entire toolchain.
4) We provide our toolchain and the modifications on existing tools under an open-source license, which also includes analyzed floating-point libraries.

The paper is organized as follows: In Section II, we present the main approach to cost-model generation. We discuss different optimization methods in Section III. Section IV shows how the automatically generated models are integrated into an open-source WCET analyzer. We evaluate the entire toolchain of NEO in Section V and discuss related work in Section VI. Section VII concludes our work.

II. APPROACH

The core of our contribution is an approach that automatically generates instruction-level time and energy models for a specific hardware platform. Our targeted hardware platforms are small microcontrollers, such as ARM Cortex-M0+ [5], which are widely used in the area of IoT. Compared to platforms with several memory hierarchies, these architectures expose only limited inter-instruction effects and represent a basic processor model for WCET analysis [6]. Nevertheless, we show in our evaluation that the approach also works when using a small amount of cache memory (see Section V).

A. Basic Overview

In the current implementation of NEO, we use the following chain of tools, which is shown in the overview in Figure 1.

1) CSMITH [7] is used to automatically generate a huge number of different C benchmark programs. Executables are created from these benchmarks for the target platform. Nevertheless, arbitrary benchmark generators are usable to create input data (e.g., [8]).
2) These benchmarks are executed in the virtual machine FAUMACHINE [9] in order to gather the concrete number of instruction executions.
3) All benchmarks are concretely executed on the target platform while the execution time and energy consumption is being measured. For precise time and energy measurements, we utilize MEASUREALOT [10].
4) An optimization problem is now formulated using the numbers of executed instructions, the execution times, and energy consumptions. These formulations focus on extracting the worst-case scenarios and eventually yield the instruction-level time and energy model. We use the GUROBI optimizer [11] to solve these problems.
5) To finally yield upper bounds on execution time and energy consumption of arbitrary program code, the automatically generated cost models for execution time and energy consumption are integrated into the open-source static worst-case analysis tool PLATIN [12].

We point out that the used worst-case analysis (i.e., IPET) is sound [1]. However, our utilized cost models may be unsound, due to the measurement-based approach to determine the models. Nevertheless, as our evaluation reveals, with our models for the Cortex-M0+, we are able to determine upper bounds of program code (see Section V-B).

B. Instruction-Level Time & Energy Models

For each kind of instruction i of a given instruction set I a value TPI_i (time per instruction i) must be measured giving the time the instruction needs for execution. Execution time predictions are obtained by counting the instructions to be executed and multiplying these numbers by the TPI model. Instead of using TPI_i for runtime prediction directly, we are able to use the number of clock cycles per instruction (CPI_i) and time per cycle (TC), which satisfy the equation $TPI_i = CPI_i \cdot TC$. Similar to that, a value EPI_i for each instruction forming the energy model is needed for energy-consumption prediction. The values TPI_i and EPI_i are normally measured by writing micro-benchmark programs for each instruction and measuring their runtime and energy consumption [13], [14], [15]. As the runtime and energy consumption of a machine instruction is a very tiny quantity, it is impossible to measure it directly. Instead, the instruction has to be executed many times and the measured value has to be divided by the number of executions. However, executing the same instruction multiple times without any jump instruction in between is very uncommon in code generated by standard compilers. Hence loops have to be written with the instruction under test in the loop bodies. Moreover, the instruction forming the loop influences the measures. Therefore, it is very time consuming and error prone to write such benchmarks as this has to be carried out for each machine instruction and has to be repeated for each architecture [16].

The core idea of NEO here is to just run an arbitrary given set of (generated) benchmarks. The only requirement is that the benchmarks can be executed on the architecture under test in reasonable time. Hence it is possible to use existing standard benchmarks written in high-level programming language. The unique benefit of the NEO's modeling approach is that these benchmark are generated automatically.

Runtime and energy-consumption measurements are performed when running the benchmarks on the target platform. This gives us the runtime (T_b) and energy consumption (E_b) of each benchmark b of the benchmark suite B. The number of each type of executed instruction is counted by a virtual machine, which is represented by $IC_{b,i}$, the instruction count for benchmark b and instruction i.

Using all these numbers we set up the equation system

$$\begin{pmatrix} IC_{1,1} & IC_{1,2} & \cdots & IC_{1,n} \\ IC_{2,1} & IC_{2,2} & \cdots & IC_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ IC_{m,1} & IC_{m,2} & \cdots & IC_{m,n} \end{pmatrix} \begin{pmatrix} TPI_1 \\ TPI_2 \\ \vdots \\ TPI_n \end{pmatrix} = \begin{pmatrix} T_1 \\ T_2 \\ \vdots \\ T_m \end{pmatrix},$$

whereas n is the number of instructions and m the number of benchmarks. This equation system will be referred to as

$$IC \cdot TPI = T. \tag{1}$$

Regarding energy consumption, the corresponding equation system is described as

$$IC \cdot EPI = E. \tag{2}$$

Unfortunately, the following two problems arise:

1) Runtime T_b and energy consumption E_b are measured on the target platform and therefore not exact.
2) The values of TPI_i and EPI_i are usually not constant when considering different benchmarks $b \in B$. On standard commercially available hardware they vary with the parameters of the instruction and with the state of the processor (i.e., the contents of the cache in case of `load` and `store` operations).

Due to these two problems, there is most likely no feasible solution to Equation (1) and Equation (2). In the following section we present mathematical optimization methods dealing with the problems mentioned above.

III. Optimization Methods

The subsequent sections describe the mathematical optimization methods that eventually yield the time and energy models (i.e., TPI_i and EPI_i).

A. Least Squares

Instead of solving Equation (1) directly in order to get a time model TPI_i, we set up the equation system

$$IC \cdot TPI - T = e. \tag{3}$$

The main objective of this section is to minimize the error vector e, such that there is a feasible TPI for Equation (3). Additionally, each TPI_i must be positive. This is done by solving the following Quadratic Program:

$$\begin{aligned} \min \ & \sum_{b \in B} \left(\frac{e_b}{T_b} \right)^2 \\ \text{s.t.} \ & \sum_{i \in I} IC_{b,i} \cdot TPI_i + e_b = T_b, \quad b \in B, \\ & TPI_i \in \mathbb{R}_+, \quad i \in I, \\ & e_b \in \mathbb{R}, \quad b \in B. \end{aligned} \tag{QP}$$

Every error e_b is counted relatively to its runtime T_b, since every benchmark b is treated equally. We also call attention to the fact that some improvements concerning numerical stability might be necessary to solve (QP) properly. For instance, in our case dividing every equation in (QP) by T_b, while T_b itself is given in nanoseconds, is necessary. Since this is a Convex Quadratic Program, it can be solved very fast, for example by the Interior-Point Method [17].

As mentioned before, it is possible that instructions consume more/less time for execution due to different parameters. For example, in a pipeline without branch prediction, conditional jump instructions take more time if the condition is true and

program control flow is transferred to a different location in the program memory. Similar to that, the time to execute `load/store` instructions varies depending on the state of the cache. Furthermore, the time to execute one instruction is an integer multiple of the clock cycle time (TC) of the CPU under test. Accurate clock cycle timers are available in all processors nowadays. Therefore, since a TPI_i computed by solving (QP) is not an integer multiple of TC in general, we can round it down and up obtaining integer lower ($TPI_{i,min}$) and upper ($TPI_{i,max}$) bounds for TPI_i. Now, prediction of the runtime $\widetilde{T}_p$ of any program p is done by

$$\sum_{i \in I} IC_{p,i} \cdot TPI_{i,min} \leq \widetilde{T}_p \leq \sum_{i \in I} IC_{p,i} \cdot TPI_{i,max}. \quad (4)$$

This approach is referred to as *Least Squares FC* (floor and ceiling) in our evaluation (see Section V).

B. Least Sum Of Errors

As the energy consumption of an instruction i varies with different parameters or the state of the processor, a minimal ($EPI_{i,min}$) and a maximal ($EPI_{i,max}$) energy consumption exists for every instruction i. In order to get such values, we propose the following Linear Program

$$\begin{aligned}
&\min \sum_{b \in B} \frac{e_{b,min}}{E_b} + \sum_{b \in B} \frac{e_{b,max}}{E_b} \\
&\text{s.t.} \sum_{i \in I} IC_{b,i} \cdot EPI_{i,min} + e_{b,min} = E_b, \quad b \in B, \\
&\quad \sum_{i \in I} IC_{b,i} \cdot EPI_{i,max} - e_{b,max} = E_b, \quad b \in B, \\
&\qquad EPI_{i,min} - EPI_{i,max} \leq 0, \quad i \in I, \\
&\qquad EPI_{i,min}, EPI_{i,max} \in \mathbb{R}_+, \quad i \in I, \\
&\qquad e_{b,min}, e_{b,max} \in \mathbb{R}_+, \quad b \in B,
\end{aligned} \quad \text{(LP)}$$

which can be solved very fast, for example by the Simplex Algorithm [18]. Again, every error is counted relatively to its energy consumption E_b allowing every benchmark b to have an equal impact. With values calculated by solving (LP), prediction of the energy consumption $\widetilde{E}_p$ of any program p is done as in (4). This method is referred to as *Least Sum* in the evaluation section.

IV. Integrating Models Into PLATIN

The Portable LLVM Annotation and Timing toolkit [12], [19] or short PLATIN, is a generic tooling framework for WCET-aware compilation, which is fundamentally based on the LLVM compiler infrastructure [20]. PLATIN was originally developed for the time-predictable hardware architecture PATMOS [21]. The PLATIN toolkit is a comprehensive framework for research on timing-analysis techniques and is the fundamental basis of recent works in this area [8], [22].

In the following, we first describe the core concepts of PLATIN and the approach of WCET-aware compilation (see Section IV-A). Secondly, we describe how we extended PLATIN and integrated the cost models in order to support the ARMv6-M architecture for the Cortex-M0+ processor, which we use in our evaluation (see Section IV-B).

A. Benefits of WCET-Aware Compilation

A major challenge when performing WCET-analyses is the determination of the control-flow graph (CFG) with its flow facts (e.g., loops and their bounds) on machine-code level. This can be achieved by reconstructing the CFG from the lowest abstraction level of machine code. A further approach to this problem is to perform WCET-aware compilation where flow facts are determined on higher abstraction level and transformed during the lowering process as performed by PLATIN. With this approach, PLATIN avoids the loss of high-level information on program flow during compilation.

A further benefit is the analysis of flow facts on an intermediate representation (i.e., LLVM IR): PLATIN profits from analyses available inside the highly optimizing LLVM compiler framework. For example, LLVM uses sophisticated scalar-evolution analyses [23] to infer costs for loop unrolling. PLATIN reuses this information for loop bounds, which is required to bound the flow in IPET-based WCET analyses. We assume that the exploitation of flow facts drawn from LLVM's existing polyhedral optimizer POLLY [24] will yield further reusable information for the purpose of refinement of the IPET through additional constraints.

However, PLATIN is not limited to LLVM-based analyses. The core of the toolkit is a file format called Program Metainfo Language (PML), which is used to store all meta information about the analyzed program. This generic way of expressing flow facts enables PLATIN to profit from other external analysis tools such as SWEET [25], which are then added to the main PML file. In addition to that, the determination of flow facts using the commercial aiT[1] WCET analyzer is possible, using PLATIN's `ais2pml` conversion functionality from aiT's flow-fact format (i.e., ais) to PML.

Besides flow facts such as loop bounds, PLATIN stores a mapping between the intermediate representation and the machine code of the program. This mapping enables relating IR code with machine code using control-flow relation graphs (CFRGs) [26]. The construction of the CFRGs happens during the lowering process of each backend and is consequently target-dependent. In the following, we describe the required changes we made in order to make PLATIN support the ARMv6-M architecture.

B. Extending PLATIN

As PLATIN only supported the PATMOS architecture, we now provide patches introducing support for the ARMv6-M architecture of the NXP KL46Z board we use for our evaluations (see Section V). The extension of PLATIN for ARM is twofold: First, it requires a compiler capable of creating a PML representation of the functions to analyze. Second, instruction costs and memory models of the ARMv6-M architecture need to be included in PLATIN. To differentiate our extension to PLATIN, we describe our ARMv6-specific implementation as PLATIN$^{\text{ARM}}$ in the following.

[1] `https://www.absint.com/ait/`

1) Adding the ARMv6 Architecture: PLATIN already provides extensions to the CLANG front end, leaving only adjustments to the ARM backend to realize the export to the PML representation during compilation. However, PLATIN comes with a generic implementation of this export, which proved to be highly reusable, requiring only minimal specialization.

Including the details of the ARMv6-M architecture into PLATIN is accomplished by implementing an architecture interface provided by PLATIN. Integrating the calculated instruction cost models consists of annotating these costs to the instruction names used by the ARM backend of the LLVM compiler infrastructure. Due to these generic interfaces of PLATIN for the tool itself and the handling of PML representations in LLVM, porting PLATIN to a new architecture requires minimal effort as already mentioned in [12].

2) Integrating Floating-Point Support: A drawback of WCET-aware compilation is that target-specific details can be added to the program representation during the backend's lowering process (i.e., the mapping from target-independent LLVM assembly code to target-specific machine code). For example, when the backend detects that the target platform has no floating-point support, special libraries that emulate floating-point operations in software are inserted into the code. Such functions are necessary for the Cortex-M0+ platform. To solve this issue, we developed these target-specific libraries and integrated them into PLATIN to support divisions and floating-point arithmetic. Besides the NEO toolchain, we provide the flow facts and timings of these open-source libraries.

C. Applicability of NEO: Teaching Real-Time Systems

This adjustability of PLATIN with acceptable effort is also useful in the context of teaching real-time systems and timing analysis. The NEO toolchain makes it possible to practically teach students about real-time analysis and WCET analysis at extremely low costs on a COTS platform. Current WCET analysis relies either on state-of-the-art static analyzers, which usually are closed-source and come with expensive license fees, or specialized hardware measurement tools.

In contrast, NEO provides similar capabilities completely open-source and, with the hardware setup presented in this paper, at the cost of no more than $20.00 per platform. Our experiences with low-cost, well-documented evaluation platforms reveal that motivation increases when students carry out there assignments in class in a practical way[2].

As the hardware platform used in this paper is capable of providing precise execution cycle counts, no additional hardware (e.g., external oscilloscope) is needed. PLATIN's practical usability makes the presented NEO toolchain perfect for experiencing WCET analysis. The validity of such an approach has already been shown, as PLATIN was used in teaching timing analysis for the PATMOS processor[3].

[2]SPiCboard for the course Systems Programming in C: `https://www4.cs.fau.de/Lehre/WS15/V_SPIC/Board/`

[3]`http://ti.tuwien.ac.at/cps/teaching/courses/wcet`

V. EVALUATION

This section presents evaluations demonstrating the applicability of our proposed toolchain. Specifically, we first present the precision and reliability of our cost models (see Section V-A). In Section V-B we show static analyses of a benchmark suite for WCET analyses using PLATIN$^{\text{ARM}}$.

A. Evaluation of Cost Models

The following section presents the setup for evaluating the generated cost models (see Section V-A1) and discusses the results (see Section V-A2). In Section V-A3, we compare generated runtime models with available documentation.

1) Setup: At first, a benchmark suite was used to calculate runtime and energy-consumption models as described before. For checking the quality of the models, programs of another benchmark suite were executed and measured. The programs of the second benchmark suite were not used to calculate or improve the runtime and/or energy-consumption models. Instead, their measured runtime and energy consumptions were compared to the predicted values determined by the first benchmark suite.

All experiments below were executed on an NXP Freedom KL46Z board. The board provides an ARM Cortex-M0+ processor containing several I/O ports, 256 KB of flash memory, and 32 KB of SRAM. The KL46Z board was setup to run the execution pipeline at 48 MHz and the bus speed was 24 MHz. In our evaluation setup, the instruction and data cache (4-way, 4-set program flash memory cache with size of 64 B) were switched on. Although, our current target platform is widely used for low-power electronics (e.g., IoT) and has reduced complexity regarding inter-instruction effects (i.e., caching and pipelining behavior), PLATIN offers the possibility to further refine hardware-specific analyses. However, this is only possible when such low-level details are documented.

The MÄLARDALEN benchmarks [27] are commonly used to evaluate timing-analysis approaches [28], [29]. We used 31 of these benchmarks to validate the accuracy of NEO. Four benchmarks were not considered due to insufficient RAM/ROM on the development board.

We generated 5,000 benchmarks through the CSMITH C-code generator (version 2.1.0). CSMITH was configured with incrementing seed numbers starting with the number 1. We used the compilers GCC (4_8-2014q3) and CLANG (3.5.0-libc_2.13) and three different optimization options (O0, O2, O3) to compile these programs. This results in 30,000 executables ($5{,}000 \cdot 2 \cdot 3$). In fact only 21,984 results were considered because some of the benchmarks timed out during measurements. These generated benchmark programs were executed to get the numbers of executed instructions ($IC_{b,i}$) and to measure their runtime (T_b) and energy consumption (E_b). The resulting measurements were used to calculate the TPI and EPI model as presented before.

The runtime and energy measurements are conducted using our own measurement device, the MEASUREALOT [10]. It offers a high temporal resolution of 6 ns and an integrating energy-measurement approach with a resolution of 55 nJ. This

Instruction Class	Instructions
addsub	`adds/subs rA, rB, #imm`
addsubsp	`add/sub sp, #imm`
alu	`and/eor/lsl/.../mvn rA, rB`
branchcond	`beq/bne/.../ble off`
branchuncond	`b off`
extend	`sxtb/sxth/uxtb/uxth rA, rB`
hireg	`add/cmp/mov/bx rA, rB`
immediate	`mov/cmp/add/sub rA, #imm`
inst32	`bl addr`
lea	`add rA, [pc/sp, #imm]`
memimmediate	`ldr/str rA, [rB, #imm]`
memmultiple	`stm/stmia/stmea rA!, rlist`
mempcrel	`ldr rA, [pc, #imm]`
memreg	`ldr/str rA, [rB, rC]`
memsprel	`ldr/str rA, [sp, #imm]`
pushpop	`push/pop rlist`
shift	`lsl/lsr/asr rA, rB, #imm`

TABLE I: Instruction classes for the ARMv6-M architecture

is in contrast to typical measurement methods, which sample the drawn current in fixed intervals using an analog-to-digital converter (ADC). Activities between the sampling intervals cannot be captured and therefore are lost. Depending on the performance of the ADC and the clock frequency of the microcontroller under test, the accuracy of measurements without an integrating method may therefore be significantly compromised. MEASUREALOT also offers good integration into existing build systems and a high degree of automation. This is important for NEO, as thousands of measurements have to be conducted to build a model. However, our approach is not limited to MEASUREALOT, as any other measurement device with sufficient precision and automation capabilities can be used. Due to the high degree of automation, our device is able to measure all programs autonomously within one day.

For the approach presented in this paper, it is essential to know the exact number of executed instructions per benchmarks. For this task within our evaluations we used the virtual machine FAUMACHINE [9]. FAUMACHINE was mainly developed to be able to preform fault injection into x86-systems. The simulation of FAUMACHINE is fast due to the built-in just-in-time compiler. Hence even huge fault injection campaigns are executed within reasonable time.

Due to the clean structure of the simulator – which is needed for correct fault injection, isolation, etc. – implementing new components is straightforward to accomplish. Thus, we modeled the NXP board in only a couple of hours. Note that we did not add any timing and/or energy consumption details. The model is therefore only able to simulate the CPU according to the instruction set architecture. Instrumenting the simulator to count instruction classes was done within a few minutes.

The simulator running on a standard of-the-shelf computer is able to simulate the NXP development board in real time. That is, executing all the benchmarks of the CSMITH and MÄLARDALEN benchmark suite took about one day as each of the benchmarks terminates within a few seconds. No human interaction was necessary.

Instructions counted were grouped into the classes shown in Table I according to the ARMv6-M Architecture Reference Manual [30]. Each group consists of instructions with the same encoding scheme and same addressing modes.

For solving the optimization problems, we utilize the solver GUROBI [11]. These calculations only take a few minutes for the time and energy model on an Intel Core i7 (8 cores, 8 GB RAM), which proves NEO's performance.

2) Results and Discussion: The diagrams (see Figure 2 and Figure 3) below show how well the runtime/energy consumptions of each of the benchmark programs are predicted by the methods presented above. The x-axis shows the difference between the real and the predicted values in percentage. On the y-axis the percentage of benchmarks with lower or equal differences (the distribution of the differences) is given. For example, in more than 50 % of all benchmarks, the runtime is overestimated by less than 40 % (see Figure 2).

Runtime: Table II shows the runtime model computed by the *Least Squares FC* method introduced before. The left two columns give the minimum/maximum execution time of the instructions in nanoseconds while the right two columns show the same information in clock cycles.

Instruction Class	min [ns]	max [ns]	min [ticks]	max [ticks]
addsub	20.8	41.7	1	2
addsubsp	0.0	20.8	0	1
alu	20.8	41.7	1	2
branchconditional	0.0	20.8	0	1
branchunconditional	20.8	41.7	1	2
extend	20.8	41.7	1	2
hireg	20.8	41.7	1	2
immediate	20.8	41.7	1	2
inst32	62.5	83.3	3	4
lea	0.0	20.8	0	1
memimmediate	41.7	62.5	2	3
memmultiple	62.5	83.3	3	4
mempcrel	41.7	62.5	2	3
memreg	0.0	20.8	0	1
memsprel	41.7	62.5	2	3
pushpop	83.3	104.2	4	5
shift	20.8	41.7	1	2

TABLE II: Runtime model for instruction classes

Some of the entries in the tables above are 0. This is explained as follows, when considering the standard structure of subroutines for ARM processors:

```
foo:
  push {..., lr}  ; save registers
  sub sp, #...    ; reserve space for vars
  ...
  add sp, #...    ; free space
  pop {..., pc}   ; restore and return

main:
  bl foo          ; call subroutine
```

Because these instructions are always executed together, it is impossible to distinguish which of the instructions needs how much time and energy for execution just by measuring the overall runtime and energy consumption. Therefore, any mathematical method has degrees of freedom when assigning runtime and energy values for these instruction classes.

Additionally, instructions of the `addsubsp`, `lea`, and `memreg` classes are the least used instruction. In fact, less than 2% of all instructions are instructions of these classes. Hence it is difficult to determine their impact on the overall runtime of a benchmark using our approach.

The diagram of Figure 2 shows that the *Least Squares FC* method is able to generate runtime models, which is used to

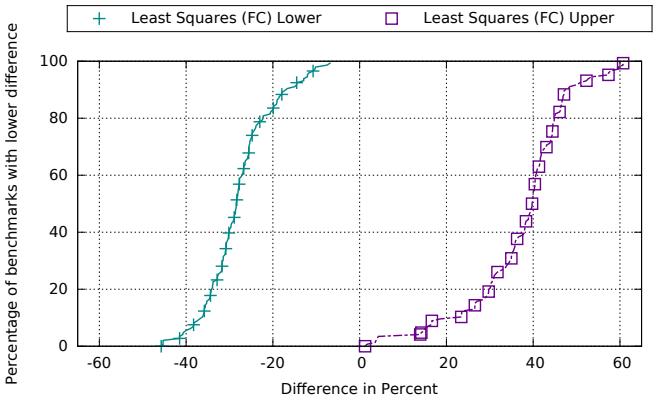

Fig. 2: Runtime Results: The figure shows the deviations of execution-time predictions of the MÄLARDALEN benchmark suite.

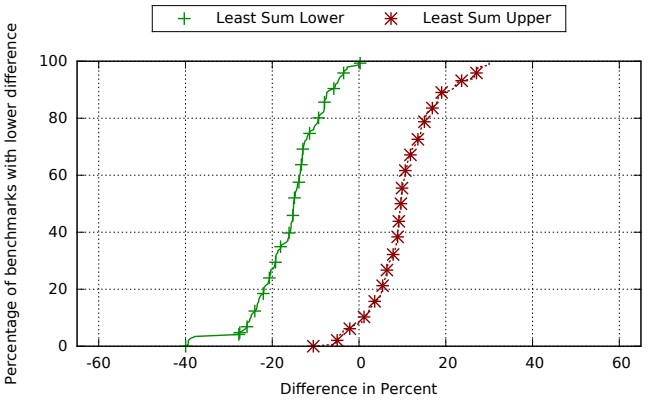

Fig. 3: Energy-Consumption Results: The figure shows the deviations of energy-consumption predictions of the MÄLARDALEN benchmark suite.

Method	Lower	Upper	Errors
Least Squares FC	-28.0 ± 7.8	$+37.0 \pm 11.8$	0.0% / 0.0%

TABLE III: Mean & standard deviation of runtime predictions of the MÄLARDALEN benchmark suite.

Method	Lower	Upper	Errors
Least Sum	-16.0 ± 7.8	$+10.2 \pm 7.6$	0.7% / 8.2%

TABLE V: Mean & standard deviation of energy-consumption predictions of the MÄLARDALEN benchmark suite.

precisely predict the runtime of other programs: All predicted lower/upper runtime bounds of the MÄLARDALEN benchmarks are correct. The results are summarized by Table III. For the lower and upper bound the arithmetic mean μ and standard deviation σ of the predictions are given by the notation $\mu \pm \sigma$.

Using the lower and upper bounds obtained by the *Least Squares FC* method, the predictions of the runtime have an average underestimation of -28.0% with an standard deviation of 7.8 % and an average overestimation of +37.0 % with an standard deviation of 11.8 %. We consider these values as practical in light of the fact that all predictions are correct.

Energy Consumption: The computed instruction-level energy-consumption model is shown by Table IV.

Instruction Class	min [nJ]	max [nJ]
addsub	0.3	0.6
addsubsp	0.0	0.0
alu	0.6	0.6
branchconditional	0.4	0.7
branchunconditional	0.5	0.5
extend	0.5	0.5
hireg	0.4	0.6
immediate	0.3	0.7
inst32	2.3	2.3
lea	0.0	0.0
memimmediate	1.0	1.1
memmultiple	1.2	1.2
mempcrel	0.7	2.1
memreg	1.1	1.1
memsprel	1.1	1.6
pushpop	2.5	2.5
shift	0.6	0.6

TABLE IV: Energy model for instruction classes

The results presented in Figure 3 are summarized by Table V. It outlines the quality of the predictions. Unfortunately, since our approach for energy-consumption predictions is not as conservative as in terms of runtime predictions not all predictions are correct.

Using lower and upper bounds obtained by the *Least Sum* method, the predictions of the energy consumption have an average underestimation of -16.0% with an standard deviation of 7.8 % and an average overestimation of +10.2 % with an standard deviation of 7.6 %. Therefore, regarding energy-consumption predictions our approach is highly accurate. On the downside, not all predictions are correct. However, we point out that these mispredictions are reasonable small.

Instruction Class	NEO	ARM	Delta
addsub	1	1	0
addsubsp	1	1	0
alu	2	1	1
branchconditional	2	2	0
branchunconditional	2	2	0
extend	2	1	1
hireg	2	1	1
immediate	2	1	1
inst32	3	3	0
lea	1	1	0
memimmediate	3	2	1
memmultiple	3	$1 + N \Rightarrow \{1, \ldots, 9\}$	{+2, …, -6}
mempcrel	2	2	0
memreg	2	2	0
memsprel	3	2	1
pushpop	5	$1 + N \Rightarrow \{1, \ldots, 10\}$	{+4, …, -5}
shift	2	1	1

TABLE VI: Automatically generated cost models are close to the documented cycle costs using the upper bounds of the *Least Squares FC* model. N describes the number of registers.

3) NEO's Models vs. CPU Documentation: Documentation of cycle counts of instructions is rare or might be erroneous [2]. However, for the ARMv6-M architecture used in our evaluation, documentation on execution times is provided by

the reference manual [30]. Unfortunately, the instruction-level timing model from the manual is limited to the assumption of a system with zero wait states. Furthermore, the memory hierarchy including caches is not part of the core and extended by the licensee of the CPU (i.e., NXP) and also not documented on a cycle-accurate level. Consequently, to use the timing information for this evaluations, we performed NEO's automated cost-model generation for a system running on 24 MHz (i.e., CPU and memory both using the same clock) and disabled caches to imitate a system with zero wait states (when accessing memory). The results are summarized in Table VI. The left column depicts the automatically generated runtime model using the upper bounds of the *Least Squares FC* method on 24 MHz. The column in the middle lists the cycles gathered by the documentation and the right column the error delta between both models. The rather pessimistic *Least Squares FC* model generated by NEO yields for about 47 % of the instructions the exact timing costs. In around 41 % cases, the model is one cycle too pessimistic. Consequently, 88 % of the instruction classes are automatically modeled in a safe way.

The instruction classes `memmultiple` and `pushpop` (12 % of the total number of classes) must be treated with special attention: The related instructions perform operations on a variable number of arguments. For example for the `push` instruction from the `pushpop` class, a number of registers up to $N = 9$ can be given. The cycle cost for `push` for pushing registers onto the stack is documented as $1 + N$. Consequently, the instruction consumes up to 10 cycles in the theoretical worst case. However, looking at the generated assembly code of the 30,000 executables produced by CLANG and GCC we used as input set, it turns out that code for pushing registers uses in 99.85 % one to four registers (in detail: 49.12 % one, 15.57 % two, 17.75 % three, 17.41 % four registers). Only in 0.15 % of the `push` instructions, five registers are used. Code using six registers or more is never generated. As already mentioned in Section V-A2, for instructions that have a negligible impact on the overall costs, since they occur very rarely, our approach is unlikely to yield an upper bound for their costs. Reconsidering the example with the `push` instruction, for 99.85 % of the instructions with $N \leq 4$, using the cycle cost of five represents an upper bound, what we consider as practical for automated measurement-based cost-model generation without knowledge of low-level details and documentation about the target hardware.

B. Evaluation of PLATIN^{ARM}

Our worst-case analysis using PLATIN$^{\text{ARM}}$ is currently reduced to WCET analysis. Integrating the energy cost models is straightforward to accomplish since the energy-cost models revealed to be practical as discussed above (see Section V-A2). Our WCET evaluations compare execution traces on the target platform with static predictions. In contrast to the energy-consumption analyses, this setup does not require an external measurement device, since internal CPU timers can be used.

To evaluate the applicability and precision of the entire NEO toolchain, we used the novel suite for benchmarking WCET analysis TACLEBENCH [31]. The suite comprises several challenging programs from the MÄLARDALEN WCET benchmarks, several other benchmarks suites (e.g., MiBench), and further application code. One main advantage of using TACLEBENCH for WCET-analysis benchmarking is the separation of initialization and main workload function. That is, a main entry point for benchmarks is stated as annotations in the code from which the analysis starts.

WCET Analyses with PLATIN$^{\text{ARM}}$*:* In order to evaluate PLATIN$^{\text{ARM}}$, we gathered a trace for each benchmark by measuring the cycles of the main workload function on the actual hardware platform. For the execution-time measurements, we use the internal timers of the Cortex-M0+ platform, which are able to capture execution times on a cycle-accurate level. This setup requires no additional external hardware for measurements and is therefore also ideal for practically teaching WCET analysis (see Section IV-C). As baseline for the static analysis, we use the execution time from a simple trace of the program, considering the input data that is compiled into the self-contained benchmarks of TACLEBENCH. Note that the predefined input data does not necessarily trigger the path with longest runtime and, as a consequence, the measurement usually represents an under-estimation of the actual WCET. The comparisons of execution traces and statically determined WCET values are summarized in Figure 4. The static analysis starts at the main workload function and the start/stop of the measurement is inserted before/after this function.

From the 53 benchmarks of TACLEBENCH (current version 1.9), we are able to evaluate 36 benchmarks: RAM overflows occur inside the processor for 10 benchmarks and thus no trace-based measurement is possible. The remaining 7 benchmarks are not analyzable since they contain recursions, which cannot be handled by PLATIN currently. Nevertheless, we argue, in compliance with the Misra-C standard [32], that the usage of recursions should be avoided in safety-critical real-time systems. Floating-point arithmetic is required for 9 benchmarks, which can be analyzed due the floating-point support we added to PLATIN (see Section IV-B2).

The main observation we draw from the benchmarks is that all traces are upper bounded by the WCET values yielded from PLATIN$^{\text{ARM}}$. For benchmark with less complex flow facts, such as loops with constant bounds and no conditionals inside the bodies (e.g., `rijndael_enc`: 38 %), or small input sizes (e.g., `binarysearch`: 23 %), the estimations are considered as precise. In summary, the over-estimations range from 23 % up to the pessimistic values of 35,557 % (i.e., `fft`). The geometric mean of the normalized values of over estimation is 208 % (i.e., an over-estimation factor of around 3). The over-estimations are due to three problems:

1) The utilized model (i.e., *Least Squares FC*-Upper) leads to the most pessimistic, but most reliable execution-time predictions (up to a maximum error of 61 % with no erroneous too optimistic mispredictions, see Section V-A2), compared to the rather precise modeling technique *Least Sum*.

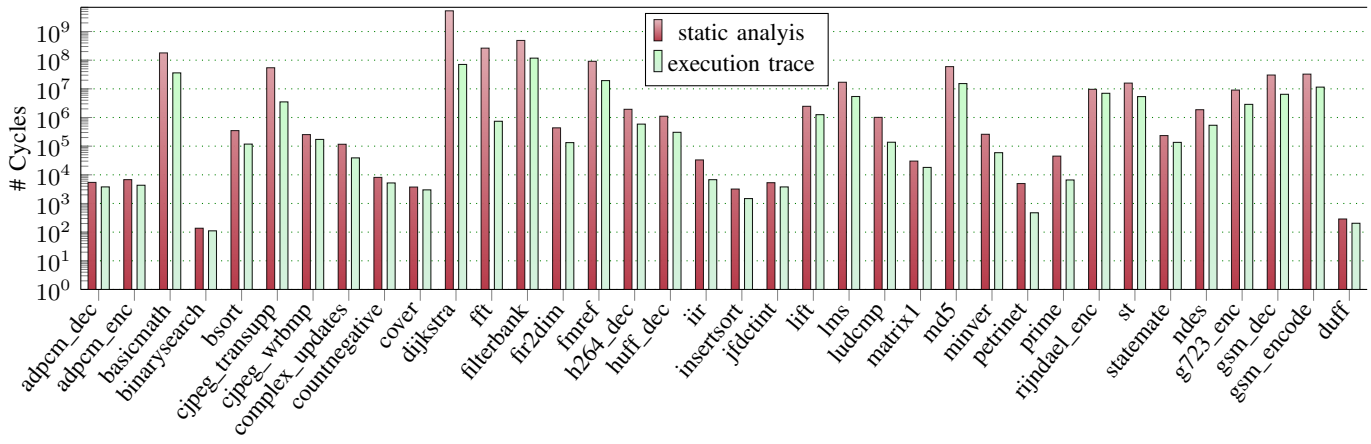

Fig. 4: The comparison (on logarithmic scale) of static WCET analysis (left bar) using the generated models with the execution trace (right bar) demonstrates that PLATIN$^{\text{ARM}}$ is able to yield upper bounds for TACLeBench suite.

2) The annotations provided by the TACLeBench suite are pessimistic upper bounds without considering context-sensitive flow information. This means, in case of `fft` a conditional statement inside a loop over the input array is entered less than half times of the input size. However, the computations are considered in each case of the IPET analysis.
3) The execution trace, which is used as baseline for the WCET analysis, considers an arbitrary input-data set that does not necessarily trigger the target-specific worst-case execution time. In general, this worst-case input-data set is unknown for existing benchmark programs [8].

From a theoretical point of view, input data might exist that leads to a higher execution time. However, practically, larger input data leads to more pessimistic assumptions. In our evaluations, we observed that programs with no input (i.e., `cover`, `matrix1`) resulted in tolerable over-estimations (i.e., 25 %, 67 %). In these benchmarks the execution times from the trace-based analysis are a reliable substitute for the actual WCET. We consider the toolchain and the acceptable degree of over-estimation as practical for many real-time systems.

Gathering Flow Facts: TACLeBench is equipped with annotation of all loop bounds and recursion depths using `#pragma` directives. Per default, PLATIN utilizes these source-code annotations for bound loops forwarded by an extended Clang C frontend, which stores the result to the main PML file. However, exploiting LLVM's scalar-evolution analyses [23], PLATIN also manages to infer required flow-facts for flow restriction of the IPET. To evaluate the effectiveness of this mechanism, we removed all manually stated annotations and observed that PLATIN is able to precisely reveal all essential flow facts for 9 benchmarks, which is about 17 % of entire TACLeBench. We assume that enhancing our setup with the usage of flow facts gathered by SWEET [25], which is already supported by PLATIN, advances our setup towards completely automatic WCET analyses.

VI. Related Work

To our knowledge, Neo is the first approach to automatically generate time and energy models based on benchmark generators and several distinct optimization methods. Our approach is different to other approaches that rely on smaller benchmark suites [33] or manually written micro-benchmarks [13], [14], [15] of single instructions or instruction pairs being executed in loops. Neo does not require knowledge about such instruction-level benchmarks, which, for example, contain non-trivial jump instructions. In contrast to other approaches, Neo relies on automatic benchmark generation and mathematical optimization techniques.

Tiwari et al. presented the power analysis of a micro-controller in [13]. The authors extracted an instruction-level power model without prior knowledge on the low-level implementation of the RISC processor. The presented experiments were carried out manually by current measurements during the execution of micro-benchmarks in infinite loops.

Chang et al. evaluated the energy consumption of a 32-bit processor (i.e., ARM7TDMI) in [14] and revealed that the hamming distance of the processed data between each CPU cycle impacts the energy demand of operations. The integration of data dependences into the model generation process is considered future work of our approach, which is possible through tailoring our benchmark-generation process [8].

Lee et al. derived instruction-level power models by using empirical and statistical methods (regression analysis) [16]. Their work is limited to arithmetic operations on data. Furthermore, their approach requires manually tailored benchmarks.

Shao et al. performed an energy characterization of a many-core processor (i.e., Intel Xeon Phi) in [15] and correspondingly built an energy model for the processor. This approach requires manually generating micro-benchmarks of instruction pairs, which were considered for measurements in a loop. In contrast to this approach, Neo is not restricted to arithmetic operations, since characterizing the time and energy demand of branch instruction is inherently possible through the automated approach of benchmark generation.

Static WCET analyzers, such as Chronos [34], SWEET [25], Heptane [35], or Otawa [36] need to solve the challenge to reconstruct flow facts from an already compiled executable. In contrast to these analyzers, PLATIN profits from the transformation of high-level flow facts through the WCET-aware compilation integrated in the LLVM infrastructure.

The benefits of WCET-aware compilation were introduced by the WCET-aware C Compiler (WCC) [37]. WCC is also able to maintain and profit from high level flow facts throughout the compilation process. Similar to PLATIN's approach, WCC maintains flow facts in a target-independent format along with their mappings to machine code.

VII. Conclusion

Creating reliable time and energy models for embedded systems is an inherently difficult task due to the lack of documentation. In this paper, we presented an optimization-based approach for generating instruction-level time and energy models covering all types of instructions by exploiting automated benchmark generation. Our approach achieves fine-grained cost models without a priori knowledge on the energetic and temporal behavior of the target platform.

To demonstrate the practical applicability of our approach, we integrated the generated models into the existing WCET analyzer PLATIN. Our evaluation on an embedded target platform reveals that the entire NEO toolchain is able to predict worst-case bounds for state-of-the art WCET benchmarks.

Since the entire NEO toolchain and PLATIN WCET toolkit [19] is build upon the widely-used clang/LLVM framework, we believe that this setup can be easily adapted and enhanced in the future. The code of the NEO toolchain, our modifications of existing tools, and our analyzed floating-point and division libraries is available under an open-source license.

```
https://gitlab.cs.fau.de/neo
```

Acknowledgments

We thank Christopher Eibel, Peter Ulbrich, and Manu Kapolke for their insightful comments. This work is supported by the German Research Foundation (DFG), in part by Research Grant no. SCHR 603/9-2, no. SCHR 603/13-1, the CRC/TRR 89 (Project C1), the CRC/TRR 154 (Project B07), and the Bavarian Ministry of State for Economics under grant no. 0704/883 25.

References

[1] P. Puschner and A. Schedl, "Computing maximum task execution times: A graph-based approach," *Real-Time Systems*, vol. 13, pp. 67–91, 1997.
[2] P. Atanassov, R. Kirner, and P. Puschner, "Using real hardware to create an accurate timing model for execution-time analysis," in *Proc. of RTES '01*, 2001.
[3] P. Wägemann, T. Distler, T. Hönig, H. Janker, R. Kapitza, and W. Schröder-Preikschat, "Worst-case energy consumption analysis for energy-constrained embedded systems," in *Proc. of ECRTS '15*, 2015.
[4] R. Jayaseelan, T. Mitra, and X. Li, "Estimating the worst-case energy consumption of embedded software," in *Proc. of RTAS '06*, 2006, pp. 81–90.
[5] ARM Limited, "Cortex-M0+ Tech. Ref. Manual," 2012.
[6] C. Rochange, "WCET Tool Challenge 2014," talk held at WCET '14.
[7] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. of PLDI '11*, 2011, pp. 283–294.
[8] P. Wägemann, T. Distler, C. Eichler, and W. Schröder-Preikschat, "Benchmark generation for timing analysis," in *Proc. of RTAS '17*, 2017.
[9] M. Sand, S. Potyra, and V. Sieh, "Deterministic high-speed simulation of complex systems including fault-injection," in *Proc. of DSN '09*, 2009, pp. 211–216.
[10] T. Hönig, H. Janker, C. Eibel, O. Mihelic, R. Kapitza, and W. Schröder-Preikschat, "Proactive energy-aware programming with PEEK," in *Proc. of TRIOS '14*, 2014, pp. 1–14.
[11] Gurobi Optimization Inc., "Gurobi optimizer reference manual," 2016.
[12] S. Hepp, B. Huber, D. Prokesch, and P. Puschner, "The platin tool kit - the T-CREST approach for compiler and WCET integration," in *Proc. of KPS '15*, 2015.
[13] V. Tiwari and M. T.-C. Lee, "Power analysis of a 32-bit embedded microcontroller," *VLSI Design*, vol. 7, no. 3, pp. 225–242, 1998.
[14] N. Chang, K. Kim, and H. G. Lee, "Cycle-accurate energy consumption measurement and analysis: Case study of ARM7TDMI," in *Proc. of ISLPED '00*, 2000, pp. 185–190.
[15] Y. S. Shao and D. Brooks, "Energy characterization and instruction-level energy model of Intel's Xeon Phi processor," in *Proc. of ISLPED '13*, 2013, pp. 389–394.
[16] S. Lee, A. Ermedahl, S. L. Min, and N. Chang, "An accurate instruction-level energy consumption model for embedded RISC processors," *SIGPLAN Notices*, vol. 36, no. 8, pp. 1–10, Aug. 2001.
[17] S. Boyd and L. Vandenberghe, *Convex Optimization*. Cambridge University Press, 2004.
[18] A. Schrijver, *Theory of linear and integer programming*. John Wiley & Sons, 1998.
[19] P. Puschner, D. Prokesch, B. Huber, J. Knoop, S. Hepp, and G. Gebhard, "The T-CREST approach of compiler and WCET-analysis integration," in *Proc. of SEUS '13*, 2013, pp. 33–40.
[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proc. of CGO '04*, 2004.
[21] M. Schoeberl et al., "T-CREST: Time-predictable multi-core architecture for embedded systems," *Journal of Systems Architecture*, vol. 61, no. 9, pp. 449–471, 2015.
[22] C. Dietrich, P. Wägemann, P. Ulbrich, and D. Lohmann, "Syswcet: Whole-system response-time analysis for fixed-priority real-time systems," in *Proc. of RTAS '17*, 2017.
[23] O. Bachmann, P. S. Wang, and E. V. Zima, "Chains of Recurrences – a method to expedite the evaluation of closed-form functions," in *Proc. of ISSAC '94*, 1994, pp. 1–8.
[24] T. Grosser, H. Zheng, R. Aloor, A. Simbürger, A. Größlinger, and L.-N. Pouchet, "Polly-polyhedral optimization in LLVM," in *Proc. of IMPACT '11*, 2011.
[25] B. Lisper, "SWEET–a tool for WCET flow analysis," in *International Symposium On Leveraging Applications of Formal Methods, Verification and Validation (ISoLA '14)*, 2014, pp. 482–485.
[26] B. Huber, D. Prokesch, and P. Puschner, "Combined WCET analysis of bitcode and machine code using control-flow relation graphs," in *Proc. of LCTES '13*, 2013, pp. 163–172.
[27] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper, "The Mälardalen WCET benchmarks: Past, present and future," in *Proc. of WCET '10*, 2010, pp. 136–146.
[28] B. Blackham, M. Liffiton, and G. Heiser, "Trickle: Automated infeasible path detection using all minimal unsatisfiable subsets," in *Proc. of RTAS '04*, 2014, pp. 169–178.
[29] J. Knoop, L. Kovács, and J. Zwirchmayr, "WCET squeezing: On-demand feasibility refinement for proven precise WCET-bounds," in *Proc. of RTNS '13*, 2013, pp. 161–170.
[30] ARM Limited, "ARMv6-M Arch. Ref. Manual," pp. 65–80, 2010.
[31] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. Sørensen, P. Wägemann, and S. Wegener, "TACLeBench: A benchmark collection to support worst-case execution time research," in *Proc. of WCET '16*, 2016, pp. 1–10.
[32] MIRA Limited, "Guidelines for the use of the C language in critical systems (misra-c:2004)," 10 2004.
[33] B. Lisper and M. Santos, "Model identification for WCET analysis," in *Proc. of RTAS '09*, 2009, pp. 55–64.
[34] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Science of Computer Programming*, vol. 69, no. 1, pp. 56–67, 2007.
[35] A. Colin and I. Puaut, "A modular and retargetable framework for tree-based WCET analysis," in *Proc. of ECRTS '01*, 2001, pp. 37–44.
[36] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "OTAWA: An open toolbox for adaptive WCET analysis," in *Proc. of SEUS '10*. Springer, 2010, pp. 35–46.
[37] H. Falk and P. Lokuciejewski, "A compiler framework for the reduction of worst-case execution times," *Real-Time Systems*, vol. 46, no. 2, pp. 251–300, 2010.