Automated Sensor Firmware Development - Generation, Optimization, and Analysis

Jens Rudolf *, Manuel Strobel †, Joscha Benz ‡, Christian Haubelt *, Martin Radetzki †, and Oliver Bringmann ‡
Institute of Applied Microelectronics and Computer Engineering, University of Rostock, 18051 Rostock, Germany*
{jens.rudolf, christian.haubelt}@uni-rostock.de
Chair of Embedded Systems, University of Stuttgart, 70569 Stuttgart, Germany†
{manuel.strobel, martin.radetzki}@informatik.uni-stuttgart.de
Embedded Systems Department, University of Tübingen, 72076 Tübingen, Germany‡
{joscha-joel.benz, oliver.bringmann}@uni-tuebingen.de

Kurzfassung

Der Entwurf Eingebetteter Systeme ist eine wachsende Herausforderung, nicht zuletzt aufgrund stetig steigender Komplexität der Systeme, Anforderungen wie Echtzeitverarbeitung und hohe Zuverlässigkeit bei gleichzeitig begrenzten Speicherkapazitäten, starken Beschränkungen der Leistungsaufnahme sowie kurzen Entwicklungszyklen. In der jüngeren Vergangenheit finden sich daher viele Untersuchungen neuer Verfahren zum automatisierten Erzeugen, Analysieren sowie Optimieren in verschiedenen Phasen des Entwurfsablaufes. Die Bewertung dieser Methoden konzentriert sich jedoch häufig auf das jeweilige Forschungsgebiet und berücksichtigt selten die übergreifenden Effekte, welche sich aus der Interaktion verschiedener Entwurfsmethoden ergeben. In dieser Arbeit stellen wir einen vollständigen, automatisierten Entwurfsablauf vor, der die Aspekte modellbasierte Generierung, Analyse und Optimierung von Firmware für Eingebette Systeme abdeckt und demonstrieren diesen an einem virtuellen Systemprototyp eines typischen inertialen Sensorknotens. Wir zeigen die Integration verschiedener Entwurfs- und Bewertungsmethoden an einem realistischen Beispiel und motivieren perspektivisch, wie mit den gewonnenen Informationen der Systementwurf verbessert werden kann.

Abstract

Embedded systems design has lately become particularly challenging due to fast increasing system complexities, real-time demands and reliability requirements. At the same time, designs are constrained by stringent power budgets, limited memory capacity and short time-to-markets. Recently, various methods to automatically generate, analyze, and optimize different stages of the design flow have been investigated. The evaluation of these methods, however, often focuses on a single associated field of research and thus may not consider the domain-crossing effects that stem from the interaction of different design methods. This paper presents a complete and fully automated workflow that covers model-based generation, analysis, and optimization aspects for embedded firmware and demonstrates it on a virtual system prototype of a typical inertial sensor node. We illustrate the integration of different design and evaluation methods on a realistic example and show potential opportunities for the application of gathered information in order to improve the design.

1 Introduction

The design of embedded systems is subject to steadily increasing complexity. Higher functionality and performance are in direct contrast to given power consumption and timing budgets of an existing system design. For non-stationary embedded devices, the most severe design restriction is certainly power consumption. On the other hand, more and more data is generated by today's applications and even (pre-)processed in place on the embedded device. This increases the system load and affects the timing characteristics negatively. In sum, these facts lead to contradictory constraints as well as large and non-trivial design spaces. To face these challenges, automated design methods for constraint-aware hardware/software co-design are essential in order to keep the system design process manageable at reasonable time and effort.

1.1 Motivation

In this context, different research fields have evolved in the domain of electronic design automation over the years. This includes e.g. model-based development and generation, automated optimization methods, or fast yet accurate system analysis and simulation to name just a few. The evaluation of recent methods is, however, often limited to the scope of the associated research field and rarely goes beyond. The impact of single components, but moreover their interaction within a hardware/software co-design workflow, remains open after all. As a consequence, positive as well as negative aspects that originate from the interaction of different design methods are not actively studied. This leads to unused potential and possibilities in the worst case, which motivates the work presented in the following.

ISBN 978-3-8007-4945-4

1.2 Contribution

On the example of a smart sensor node, this paper introduces a complete and fully automated embedded system design workflow, including generation, optimization, and analysis aspects. Beyond addressing the single components of this flow, the main focus is further put on the interaction of the individual stages, which are defined as follows:

1. Model-based generation of firmware code for a sensor node with integrated processing unit.
2. System simulation with power and timing analysis by using a virtual system prototype.
3. Automated optimization of the memory subsystem.
4. Post-optimization simulation coupled with a second power and timing analysis step.

Beyond that, a virtual system prototype of a sensor hub system and its integration with the above methods and corresponding implementations is presented. With that, the combined evaluation of otherwise isolated development steps is demonstrated and discussed. Finally, the possibility for a step-wise refinement of a system design at hand through interaction of the above stages in a feedback loop is outlined.

1.3 Document Structure

State of the art in relevant areas of model-based sensor firmware development, embedded system memory optimization methods, and highly accurate timing simulation is discussed in Section 2. Afterwards, Section 3 introduces the overall system design workflow, followed by a description of the individual building blocks. The ARM-based demonstration platform is presented in Section 4, followed by an outlook on potential extensions and future work in Section 5. Results and evaluation for exemplary use cases are given in Section 6. Section 7 concludes this paper.

2 Background and Related Work

The electronic design automation fields that are combined in this work, i.e. generation, optimization, and analysis, are non-overlapping in terms of related work. The following section therefore categorizes and discusses the state of the art for the individual workflow blocks separately. Approaches that deal with a combined consideration, comparable to what is presented in this paper, have not been discussed in recent literature to this day to the best of our knowledge.

2.1 Model-based Sensor Firmware Development and Generation

Firmware, running on modern embedded sensor subsystems has recently become increasingly complex due to a rising number of tasks carried out directly on the integrated processing unit [1]. Its development thus requires a model-based approach in order to meet today's constrained budgets and time-to-market.

Different dataflow based models of computation (MoC), e.g. synchronous dataflow (SDF) by Lee et al. [2], cyclo-static dataflow (CSDF) by Bilsen et al. [3], or scenario-aware dataflow (SADF) by Theelen et al. [4] have been studied extensively and proven suitable for design and optimization of digital signal processing applications (DSP) [5] in the past. These MoC provide useful information on important aspects of the system under design, e.g. consistency, deadlock-freeness as well as throughput, allow for optimization of for example latency [6] and memory usage [7] and enable the automatic synthesis of a periodic admissible sequential schedule (PASS) [8].

Tools and frameworks as for example Ptolemy II [9] and others have emerged, enabling developers to compose, evaluate and visualize such models. These tools provide sophisticated algorithms for model transformation and optimization and software synthesis for DSP targets [10] as well as synthesis of parallel hardware implementations [11] has been conducted successfully.

However, fully automatic firmware synthesis for low resource embedded systems such as microelectromechanical (MEMS) sensor nodes and hubs remains critical due to the stringent requirements to code and data size as well as system overall energy consumption. To this day, available solutions lack the ability to configure the hardware's power modes. Furthermore, they provide no mechanism to automatically integrate pre-existing C or C++ algorithm code and thus do not allow to easily reuse well-tested and platform-optimized implementations without manual integration effort.

For our embedded design workflow, we propose a hybrid approach to automatically generate a complete operational sensor firmware binary from a dataflow model, which configures the sensor to use the optimal available power modes for the given task and leverages software reuse by integrating with existing target platform code.

2.2 Embedded System Memory Optimization Methods

In embedded system design, hardware and software are often developed side by side. This fact allows direct influence on the structure of the memory subsystem, which is of high interest due to the large share that is attributed to it in terms of power consumption. For Static Random-Access Memory (SRAM), still the dominant memory technology in embedded devices, corresponding figures in literature go, depending on the application, far beyond 50% compared to the overall system energy budget [12]. Potential optimization subjects in this connection are memory allocation, i.e. how many memory instances or banks and of which size to use, and application binding, i.e. mapping of application code and data to memory units.

Sensor nodes or hubs can be seen as low-end embedded devices, a class of systems that rarely comes with cache memory. Instead, a widely adopted concept for embedded systems in this class are so-called scratch-pad memo-

ISBN 978-3-8007-4945-4

ries, a static form of caching memory. That is to say, the scratch-pad content is statically determined at system design time, which keeps the run-time overhead that comes from otherwise required dynamic cache coherence protocols at a minimum. The authors of [13] for example present a method based on dynamic programming that assigns the scratchpad content either with focus on performance or energy minimization. Menichelli et al. [12] put the focus on energy minimization only and add the consideration of power-down phases for the main memory to their optimization problem.

For designs without cache or scratch-pad, different methods for memory partitioning have been presented. Benini et al. [14] developed an automated partitioning algorithm for on-chip SRAM based on exhaustive search. Their method searches for the global optimum memory banking with minimum energy consumption. The authors of [15] use a genetic algorithm instead and include the interconnect into their considerations.

A fact, common to all of the above methods is the already set memory allocation. That is, number and size of memory instances is given, hence only the banking allows for different variations, wherefore quite some optimization potential is lost. The authors of [16] address this point and present an optimization model that allows the combined optimization of memory instance allocation and application binding of address ranges to instances.

Subsequent work extends the partitioning concept by the use of memory low-power modes. The approach of Steinfeld et al. [17] can be named here. Unfortunately, it is limited to memory blocks of equal size and only considers simplistic low-power mode activation schemes. The authors of [18] improve on that work by introducing a more fine-grained low-power mode activation model and the possibility to define peak power constraints.

2.3 Timing Simulation

Timing analysis is an important part of embedded system design. Especially during design space exploration it is necessary to be able to evaluate the performance of the system in development to allow informed design decisions. To that end, it is crucial that such an evaluation is as fast as possible in order to keep the exploration process efficient. Furthermore, timing analysis has to yield sufficiently accurate results to be useful as input for design decisions. There are several different approaches to timing analysis that have been developed with distinct requirements in mind. Those that are suited for early performance evaluation during design space exploration are dynamic approaches based on simulation or execution traces.

A very fast, yet accurate approach to timing simulation is source-level timing simulation (SLTS), which has been worked on extensively [19], [20], [21]. These contributions have in common, that it is possible to run the timing simulation on a simulation host, that is usually much faster than the target platform. Similar approaches based on machine learning have been proposed [22], [23], which also allow the timing simulation to be run on a much faster simulation host, while the resulting timing estimate represents the performance on a target system. One major drawback of approaches with differing simulation host and target system is the impossibility to accurately simulate the timing of low-level software such as embedded firmware.

Hence, several techniques have been proposed to handle low-level software as well: hybrid- and binary-level simulation (BLS). Binary-level approaches simulate the timing behaviour during execution of a target binary, usually executed on some kind of virtual prototype (VP) of the target system. There are approaches, which integrate timing simulation into a SystemC-based system-level simulation [24]. Although binary-level simulations are usually slower than source-level simulations, the advantage of binary-level simulation techniques is the flexibility as it allows the timing simulation of any software that can be simulated by the virtual prototype in use. Furthermore, by choosing different kinds of VP, it is possible to control simulation speed and accuracy.

Hybrid simulation methods try to overcome the drawbacks of cross-platform approaches like SLTS and machine learning based techniques by combining those with BLS. Wang et al. proposed a hybrid approach that combines host-compiled source-level simulation with an instruction-level simulation for target dependent software [25]. Another approach that combines SLTS with a SystemC-based VP relies on annotating the target source-code before compiling it for the executing VP [26].

3 System Design Workflow

In this section, we first give a short overview of our system design workflow, followed by a more detailed description of the steps and their interaction with each other and the virtual system prototype (VSP). In addition, we provide an in-depth discussion of the methodologies implemented in each step.

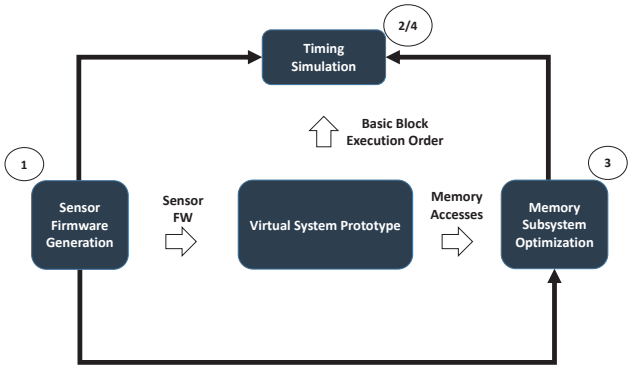


Figure 1 Overview of system design workflow

The system design workflow, as briefly introduced in Section 1, is shown in Figure 1. It starts with a model-based generation of sensor firmware (see Figure 1 - step 1). More specifically, an actor-oriented data-flow model of the software is transformed to C or C++ source code, while actors are scheduled for minimal energy consumption. The resulting firmware is then compiled and run on the VSP, which is used for functional verification of the sensor firmware.

Next, the analysis step is executed (see Figure 1 - step 2/4). This steps consists of timing simulation of the executed firmware as well as power simulation of the memory subsystem. These analyses rely on additional optional input, specified in a JSON-based data exchange format. The latter was designed to allow efficient communication and interaction between the different steps and applied methodologies. In general, it enables the description of all system aspects that are relevant for the different steps of our design workflow, such as:

- Hardware characteristics, for example the memory subsystem.
- Firmware characteristics, for example input source file and function.
- Constraints, for example power and timing constraints.

However, not all aspects that are covered by the exchange format are relevant to the methodologies presented in this work. Therefore, only relevant points are outlined in the following as part of the individual design step descriptions. The timing analysis (see Figure 1 - step 2/4) interfaces with the VSP to get the binary basic block execution order. Hence, the start address of each executed basic block is passed to the analysis for an online, possibly context-sensitive timing analysis. In addition to the obligatory total simulated execution time of the firmware, timing analysis optionally yields a report in JSON format, containing an evaluation of all specified timing constraints. The memory subsystem power analysis (see Figure 1 - step 2/4), on the other hand, communicates with the VSP to be informed on any memory access. This information is used to trigger a power state machine that simulates the energy and power behavior of the memory subsystem. Thus, power simulation yields total energy consumption and peak power figures. Furthermore, optionally defined power constraints are evaluated, while the corresponding results are provided via the data exchange format.

Next, the optimization step (see Figure 1 - step 3) is executed. For this, all memory accesses during system simulation on the VSP are captured and further used to derive memory access statistics for the individual parts of the firmware, referred to as profiles. This data set is then used to find an energy-optimized partitioning of the memory subsystem. On top of that, the utilization of memory low-power states is evaluated and optimized. Finally, additional instructions for power-mode switching are inserted, followed by a reorganization of the firmware in order to match the optimized memory structure.

In step four, the resulting, optimized binary is executed using the virtual system prototype to re-run the power and timing simulation. The resulting timing and power information can be used to further improve the design in another iteration of generation and optimization step - either manually or automatically. Note that such a feedback loop can be realized using our current tooling environment, since all necessary information can be easily shared and exchanged between the individual stages using the data exchange format. Section 5 contains a detailed discussion of how such an iterative workflow could improve automated sensor firmware design.

In the following, every workflow step is outlined in detail.

3.1 Sensor Firmware Generation

Automatic model-based firmware generation enables developers to handle the increasing complexity of today's implementations. However, code and data size as well as energy consumption must be minimal to meet the stringent requirements of low resource embedded devices as for example sensor nodes. In this section, we introduce a novel methodology to generate energy efficient sensor firmware from dataflow models as a first stage of our automatic embedded design workflow.

Our proposed approach consist of four distinct steps: 1) model composition, 2) actor scheduling, 3) code emission, 4) compile and link. Figure 2 outlines these steps for a simple sensor firmware example that reads acceleration samples from the sensor, applies a detection algorithm for a tap gesture with an input data rate of 200 Hz onto the signal and feeds the result into a system output.

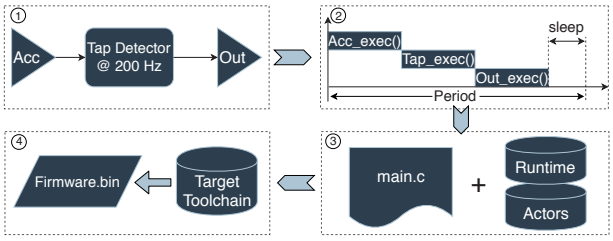


Figure 2 Firmware generation outline: 1) model composition, 2) actor scheduling; 3) code emission; 4) compile and link

The following paragraphs explain the particular steps in more detail.

1) Model composition As a first step, the developer composes the firmware model from signal flow components, so called *actors*. These actors define the data processing operations. They are connected via *channels* to read input data and pass on their computation results. Furthermore, actors can be annotated with attributes, e.g. required input data rate or latency. In order to assist model composition, we provide a Python based framework including a library of selected actors for signal input, output and gesture detection. The functions and classes are inspired by the Ptolemy II project [9]. To extend the functionality, the developer may add new actors by deriving from the available base classes in the library. Listing 1 presents the necessary Python code to compose the model from Figure 2 with our framework available in the *dataflow* module.

2) Actor scheduling In order to create a schedule, the actor based model is transformed into an SDF graph. Actors form nodes and channels the edges. Edge production and consumption rates are derived from data rate requirements of the respective actors. The production rates of source nodes are used to provide the energy optimal sensor con-

figuration. A consistency check is performed by computing the rank of the connection matrix [2] and symbolic execution is conducted to prove deadlock-freeness and create a repetition vector. A simple list-scheduling is applied to generate a periodic sequential schedule [8].

Listing 1 Model input via Python dataflow module

```
from dataflow import (Accelerometer,
    CompositeActor, Stdout, TapDetector)

model = CompositeActor('model')
Acc = Accelerometer('Acc', model)
Tap = TapDetector('Tap', model, rate=200)
Out = Stdout('Out', model)
model.connect(Acc.output, Tap.input)
model.connect(Tap.output, Out.input)
```

3) Code emission In this step, the generated actor schedule as well as the derived sensor configuration is transformed into C code. A single source file is generated, containing the *main()* function that implements the configuration logic, e.g. for setting the correct sensor data rate and schedule execution. Memory buffers are instantiated to enable data exchange between the actors along the channels in the model. The size of these buffers are derived from maximum edge marks in the dataflow graph. Preprocessor statements are inserted to include the headers that contain the necessary declarations to access the sensor modes (Figure 2 - Runtime) and actor firing functions (Figure 2 - Actors). As we do not generate the implementation of these functions out of the dataflow model, they have to pre-exist in source form or as pre-compiled library for the selected target platform. The schedule execution is emitted as an infinite loop, which invokes the actors firing functions according to the sequential schedule. At the end of each period, a sleep statement is inserted to put the processor into a low power mode, what further reduces energy consumption. Here, the source actors (sensors) are responsible for waking up the system once the next sample is available.

4) Compiling and linking In the last step, the generated source code, that contains the sensor configuration and the implementation of the periodic sequential actor schedule is compiled using the corresponding target toolchain. The hardware-specific functions as well as the implementations of the actor firing functions are supplied either in source code form or as pre-compiled libraries during the linking stage. For our demonstration platform, we used the open source GNU ARM EABI GCC toolchain and compiled the code for ARMv6 CPU architecture combined with softfloat ABI.

In the end, we get a fully operational firmware binary that is executable on the target platform. In our proposed workflow, this generated firmware is subject to further analysis and optimization steps as detailed in the next sections.

3.2 Memory Subsystem Optimization

The optimization of the memory subsystem at system design time can be highly relevant in terms of energy consumption reduction and in order to satisfy peak power restrictions (cf. Section 2.2). This section describes the corresponding methodology that has been integrated into the system design workflow according to Figure 1.

An overview of the memory subsystem optimization flow for the sensor hub system design at hand is depicted in Figure 3. It is subdivided in three subsequent steps as follows.

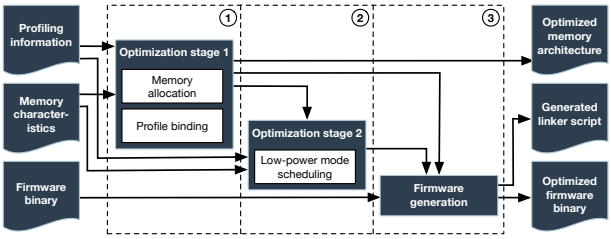


Figure 3 Overview of memory subsystem optimization

In a first step, referred to as optimization stage 1, memory allocation and profile binding are determined. In this context, each profile denotes a single function or data block of the firmware application. The allocation describes the set of memories to use, including cell technology, storage capacity, and banking. The binding part defines the mapping of every profile, which is part of the application, to the set of allocated memories. Main input for this optimization step is a collection of profiling information, i.e. detailed memory access statistics, as obtained for the investigated firmware from simulation using the virtual system prototype (VSP). This set of information is represented in the previously discussed JSON-based data exchange format and includes the following values per profile:

- Profile type, distinguishes code and data
- Profile name or label respectively
- Address range and size
- Duty cycle, i.e. number of cycles this profile is in use
- Number of read accesses
- Number of write accesses
- Dependency vector, reflects the accumulated dependencies of this profile to every other profile in code and data flow graph

Beyond that, a set of memory characteristics is required as input. The representation of this data set is again based on the data exchange format and comprises energy and power figures for all memory components that are available to the system design engineer.

The optimization potential that motivates for this step originates from the fact that, in terms of energy efficiency, it is cheaper to access a small SRAM instance. Main reason is the tremendous impact of the memory periphery, i.e. address logic, amplifiers, or pre-charging units, that considerably increases with memory size. Hence, it is beneficial to have frequently accessed application profiles in possibly small memory instances in order to reduce the energy consumption altogether. For the solution of this optimization

problem, different methods and corresponding implementations are available and integrated into the workflow. This includes, on the one hand, the combined optimization of allocation and binding with ensured global energy minimum for the given set of input data using integer linear programming [16]. On the other hand, different heuristics for efficient clustering of inter-dependent profiles, e.g. modularity or min-cut, are available for this purpose [18].

The following second step (cf. optimization stage 2) is directly based on allocation and binding but also relies on the previously discussed input information of stage 1 (cf. Figure 3). It is only executed if memory low-power modes are available. If so, this optimization method allows the determination of an optimal schedule for the set of available operation modes.

In case of SRAM memories, static power consumption basically originates from leakage currents, whose reduction can be achieved for example by using low-power modes based on biasing methods and/or power gating. Another optimization goal is consequently static power consumption minimization for the set of allocated memories. The used optimization model captures the mentioned inputs and optimization goal and is implemented as mixed-integer quadratic program [18]. After being solved, it yields a low-power mode configuration vector for every function within the set of application profiles. This vector consists of one element per allocated memory, while every element again is part of the set of available low-power modes.

In the final step, the results of both optimization stages are fed back to the firmware application level in a transparent and fully automated way. On the one hand, this includes the generation of a linker script, which reflects the determined memory allocation. On the other hand, several code modifications are applied on the assembly level. These are:

- The placement of application profiles according to the binding with respect to memory allocation and the generated linker script.
- The insertion of code snippets for the controlled activation and deactivation of determined memory low-power modes on relevant transitions between single (code) profiles, based on the result of optimization stage 2.

Finally, the modified assembly and linker script are fed to the assembler and linker of the cross-compilation toolchain again, in order to generate the final firmware binary that reflects all optimization results.

3.3 Timing Simulation

In this section we first give a detailed explanation of the simulation approach in use as well as the underlying timing model. In addition, we discuss the online timing constraint evaluation that is executed along with the timing simulation.

Figure 4 shows an overview of the timing analysis approach we integrated in our VSP, while the former is based on work proposed in [24]. Since we use a binary-level approach, only the target binary is required for the simulation

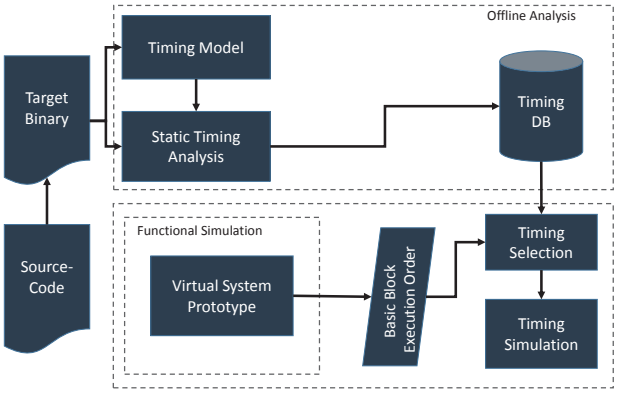


Figure 4 Overview of binary-level timing simulation

to work. The original source code can simplify the specification of timing constraints, which is discussed later in this section. First, the target binary is used to do a static timing analysis based on a timing model for the target hardware. In this case, it models a platform similar to our VSP. More specifically, the modeled target consists of an ARMv6 microcontroller, memory, and no caches. As this microcontroller consists of a simple two-stage pipeline and uses a static branch prediction (predicts always not taken), we do not model the pipeline explicitly. We rather use an instruction-latency based model, which considers branches to account for penalties due to mispredictions. Hence, in contrast to [24], a context-sensitive timing analysis or simulation is not necessary or done in this work.

As a result, the static timing analysis yields a so-called *TimingDB* or timing database, which is a data structure that allows analysis-agnostic storage of information required for fast and accurate timing simulation on binary- and source-level [27] [28]. More specifically, a TimingDB can store context-sensitive basic block timings as well as information necessary to reconstruct the binary-level control-flow. Note that all timings are stored in terms of cycles to allow execution frequency independent simulation.

During execution of the target binary, the timing simulation is triggered on each executed basic block and receives the start address of that basic block as well as the last address of the preceding basic block (see Figure 4 - Functional Simulation). Based on the information contained in the previously generated timing database, it is possible to derive the corresponding execution time as well as potentially occurring delays due to branch mispredictions. Using this information, the total execution time is accumulated during functional simulation and returned before returning from `sc_main`.

An optional step of the timing simulation process is the evaluation of timing constraints. To that end, parts of the data exchange format are designed specifically to model various constraints at different granularity. Note that Section 4.3 contains a more detailed discussion of possible constraints, all of which contain one or more attributes used to describe the program entity to be constrained. Supported are both binary- and source-level entities, while the most fine-grained are binary-level basic blocks. The latter are identified by address, while source-level entities are

identified by source file and line.

As timing simulation is done on binary level it is necessary to map the latter to binary basic blocks. That way, it is possible to evaluate a constraint each time a corresponding basic block is executed. Lets assume we define a constraint to restrict the WCET for a function called *handle_interrupt*. To be able to evaluate that constraint, we need to measure the simulated execution time between the first and the last executed basic block of that function. Since a binary-level function may have multiple entry and exit points, we use each pair of caller and return-to basic block of a function instead. Hence, the mapping between entities (and therefore constraints) and binary basic blocks also works with complex control-flow and run-time calculated call targets. To be able to map information about source-level entities to binary basic blocks, we use a matching algorithm based on [29].

Once all timing constraints can be mapped to binary basic blocks, we create an annotated binary-level control-flow graph. In this CFG, we annotate each basic block with one or more timing constraint references. Thus, each time that basic block is entered during functional simulation the simulated timing is recorded and passed to each of the annotated constraints. Internally, each constraint is implemented using a finite-state machine (FSM), which decodes consecutive invocations of a constraint.

After simulation, a result file is generated that satisfies the data exchange format and contains evaluation information of all constraints. This file lists each defined constraint including the actual execution time determined during simulation and a boolean field that signals the violation of a constraint.

4 Demonstration Platform

To demonstrate our proposed workflow, we implemented a SystemC virtual system prototype (VSP) for a microelectromechanical (MEMS) system based sensor hub, which shows some similarities to a Bosch Sensortec BMF055 [30]. We instrumented it with interfaces for a memory profiler and a timing model to enable timing simulation. The next three subsections present these particular components in more detail.

4.1 Sensor Hub Virtual System Prototype

The SystemC virtual system prototype is shown in Figure 5 and consists of three parts which are based on components available from the open source SoCLib project [31]:

- A 32 bit ARMv6 processor model with support for ARM11 and Thumb instruction set running at 100 MHz;
- A configurable memory model that serves as system main memory;
- A simplified model of a triaxial MEMS acceleration sensor that provides a three-dimensional acceleration signal with 16 bit resolution per axis sampled at 1600 Hz.

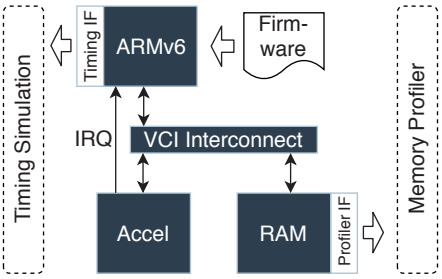


Figure 5 Virtual prototype of a MEMS sensor hub

An m-to-n interconnect implementing the TLM 2.0 based virtual component interface (VCI) connects the three components and provides memory mapped read and write accesses to the firmware executing on the processor.

The sensor component has a single interrupt request (IRQ) line that directly connects to the first IRQ input of the processor model. It operates at a constant output rate of 1600 Hz and raises an IRQ once a new sample is available in its data registers. Thereby it reads the actual acceleration samples from a given CSV file containing a pre-recorded signal from a real sensor.

The processor model implements an interpreting instruction set simulator for both ARM11 and Thumb instructions. It features classic ARM exception handling and debugger interface to connect to the GNU Debugger. We extended it with a mechanism to interface with the timing simulation explained in Section 3.3 that traces the taken branches and jumps during program execution within a distinct timing model.

The memory component models a classic static random-access memory (SRAM) of configurable size and structure. Additionally, it is equipped with an interface to interact with the memory profiling mechanism as presented in the next subsection.

4.2 Memory Profiler

The memory profiler consists of two parts. This is, on the one hand, an access measurement unit (AMU) that captures and resolves memory accesses, which are further processed to detailed access statistics. On the other hand, a power state machine (PSM) that is triggered on occurrence of relevant events. This allows the generation of detailed power and energy figures for the memory subsystem and its individual components.

According to Figure 5, the profiler is directly attached to the memory model of the virtual system prototype. This is realized via the following basic interface (cf. Listing 2).

Listing 2 Memory profiler interface

```
unsigned read  (unsigned address);
void     write (unsigned address,
                unsigned data,
                unsigned mask);
```

Based on the passed address parameter, the AMU is able to resolve the affected memory unit and application profile. The collection of all measurement points results in the memory access statistics that are used for optimization as discussed in Section 3.2.

ISBN 978-3-8007-4945-4

Beyond sampling of accesses, the PSM models the behavior of the memory in terms of energy and power consumption. It is triggered whenever:

- An instruction fetch occurs
- Memory is accessed for a read/write
- The memory-mapped power-mode configuration register of the memory subsystem is modified

Each of the above events leads to a change within the state space as illustrated in Figure 6. The implemented memory power model supports two SRAM low-power modes. These are light sleep, based on source biasing, and deep sleep, which adds power gating of the memory periphery on top of the light sleep state.

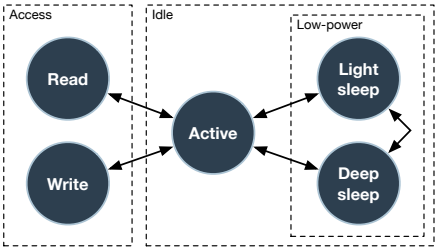


Figure 6 Memory PSM state space

The applied actual energy and power figures are based on memory simulation with the tool CACTI [32] and follow information as provided by the author of [33]. Overall, the following analysis results are provided by the power state machine when attached to a simulation run of the virtual system prototype:

- Energy consumption per memory instance from reading/writing
- Passive power consumption per memory instance (depending on idle mode periods)
- Energy and timing penalty that originates from low-power mode changes
- Average power consumption based on the system frequency
- Peak power consumption of the memory subsystem within the simulated period

4.3 Timing Simulation

As mentioned in Section 3.3, the timing analysis is responsible for two tasks. First, simulating the timing in cycles according to the underlying timing model based on the basic block execution order provided by the VSP. Second, evaluating optionally defined timing constraints, which allows to generate fine-grained timing information.

To support these tasks, our timing simulation framework is attached to the VSP using a simple interface, as shown in Listing 3. Based on this information, the timing simulation is able to reconstruct the number of simulated cycles during the execution of the VSP.

Listing 3 Interface to timing simulation

```
uint64_t simulate_bb(addr_t branch_inst,
                     addr_t target_bb);
```

In addition, the simulated timings are used for an online evaluation of timing constraints. As the methodology behind this process is already discussed in Section 3.3, this section focuses on outlining the possible constraints and how they can be described.

There is a basic constraint, namely point-to-point delay, the definition of which is shown in Listing 4. Thus, a point-to-point delay can be used to check the number of executed cycles between to entities of a program. More specifically, these entities may be:

- binary-level basic blocks;
- binary-/source-level functions;
- source-line information;

while source-line information consist of a file name and a line number.

Listing 4 Point-to-point delay constraint

```
{
  "type" : "p2p-delay",
  "entity_from" : "entity-id",
  "entity_to" : "entity-id",
  "delay" : time_value
}
```

In summary, the timing simulation can provide very fine-grained timing information based on a single run of the virtual system prototype. For example, it is possible to get timing information for each function of a firmware individually. Moreover, a point-to-point delay constraint can also be used to generate timing information for a loop or a sub-loop part of a function.

5 Outlook

In this section, we give another concrete example on the presented methodologies and the interaction with the VSP. On top of that use case, we illustrate how the workflow can be extended to further improve the process of designing and developing embedded sensor firmware. Note that

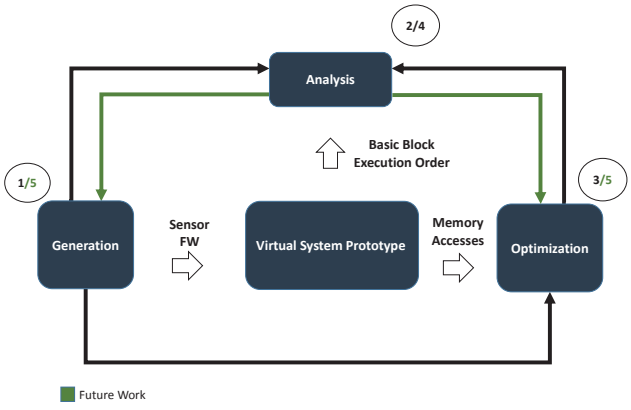
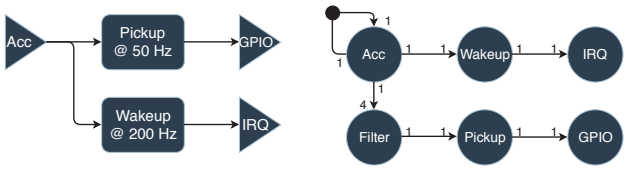


Figure 7 Extended system design workflow

this is a hypothetical example, which we did not actually run trough our workflow. It is rather meant to motivate possible benefits and use cases for future work based on our approach. The possible extensions to our workflow are highlighted in green in Figure 7.

As explained in Section 3, the first step consists of a model-based generation of sensor firmware. More specifically, the software is generated from an actor-based description and contains a power-optimized actor schedule. Thus, we first take a look at the corresponding models, that are the input for our generation step. Figure 8a shows an actor-based data-flow model of the example firmware, while Figure 8b shows the corresponding SDF graph that is created during step one of our proposed flow.



(a) Dataflow model of example firmware

(b) SDF graph of example firmware

Figure 8 Dataflow descriptions of example firmware

The software represented by these models is a simple pickup detection, that may be used in a smartphone to turn on the device's display on such an event. Of course, an application like this would have both power and timing constraints that have to be respected. Especially in case of a mobile device, low-as-possible power consumption is necessary to prolong battery life.

Based on the SDF graph, the generation step creates a cyclic actor schedule that minimizes power consumption. Finally, the actual firmware is generated and the resulting C code is shown in Listing 5.

Listing 5 Example firmware

```
void main() {
    actor_acc_init();
    actor_filter_init();
    actor_wakeup_init();
    actor_pickup_init();
    actor_irq_init();
    actor_gpio_init();

    while (1) {
        for (int i = 0; i < 3; ++i) {
            actor_acc_exec();
            actor_wakeup_exec();
            actor_irq_exec();
        }
        actor_acc_exec();
        actor_wakeup_exec();
        actor_filter_exec();
        actor_pickup_exec();
        actor_irq_exec();
        actor_gpio_exec();
    }
}
```

Next, the firmware is compiled and executed on the VSP, as indicated by Figure 7. During this first simulation run of the virtual system prototype, both timing and power analysis are run. These analyses may yield that all constraints are already met, but there might still be potential to reduce power consumption.

Thus, the next step, memory subsystem power optimization, is executed. As a result, most of the functions shown in Listing 5 are moved to a smaller, more efficient memory. Moreover, the optimization inserts instructions that implement an power optimized handling of memory power modes. Thus, each time a certain memory is not needed, it can be put into a reduced power state.

What follows is another iteration of the analysis step to check if constraints are still met. Again, it is possible that this is the case and at this point the design workflow would end with a valid sensor firmware considering all non-functional requirements. Nevertheless, it is also possible that the previously inserted instructions to handle memory power modes introduce a run-time overhead that causes a violation of timing constraints. In that case, the information provided by the analyses would be fed back into the optimization step (cf. Figure 7). Assume, for example, that the optimization assigned the functions `actor_acc_exec` and `actor_wakeup_exec` to different memories. Hence, additional instructions for power-mode changes would be executed when transitioning between those functions. Additionally, using the per-function timings as provided by the timing simulation, a system designer could decide to merge these two functions. This way, the optimization could be re-run, while making sure that the overhead introduced by placing `actor_acc_exec` and `actor_wakeup_exec` into different memories is eliminated.

Subsequent iterations could allow further design decisions that improve the firmware or lead to a final, power optimized version of it, that also adheres to the specified timing constraints. As already mentioned, the extended workflow described in this section is future work, that should focus on automating the feedback-loop briefly outlined using this example.

6 Results and Evaluation

The following evaluation of the presented system design workflow serves a twofold purpose. On the one hand, it proves the functionality of the automated flow with its components (cf. Section 3) as well as its correct interaction with the virtual system prototype (VSP) (cf. Section 4). On the other hand, resulting analysis figures from different development stages and for individual parts of the design respectively demonstrate the strength of this combined approach and moreover reveal several indicators for a step-

Table 1 General benchmark application details

	benchmark	description	LOC	size [KB]
INFO (1)	bitcount	bit manipulation tests	780	7.9
	crc32	cyclic redundancy check	511	311.7
	dijkstra	shortest path problem	749	281.1
	qsort	quick sort algorithm	549	203.7
	rijndael	AES encryption test	1515	340.8
	sha	160-bit hash generation	726	320.3
	stationary_tap	gesture detection demo	1205	6.6

Table 2 Results of simulation and analysis (baseline)

	benchmark	simulated cycles [#]	execution time [ms]	total energy [μJ]	average power [μW]	peak power [μW]
ANALYSIS (2)	bitcount	82928064	829.2	586.3	707.0	966.9
	crc32	7484813	74.8	930.2	12,428.9	19,884.8
	dijkstra	134494985	1,344.9	13,708.0	10,192.2	19,884.8
	qsort	13063895	130.6	724.7	5,547.9	10,754.6
	rijndael	140125582	1,401.2	13,262.3	9,464.6	19,884.8
	sha	63749858	637.4	5083.3	7,973.8	19,884.8
	stationary_tap	36971921	369.7	188.4	509.6	966.9

Table 3 Results of simulation and analysis (after optimization)

	benchmark	penalty cycles [#]	total cycles [#]	execution time [ms]	penalty energy [nJ]	total energy [μJ]	average power [μW]	peak power [μW]
ANALYSIS (4)	bitcount	48	120504265	1,205.0	23.5×10^{-3}	356.0	295.4	535.3
	crc32	9	7485063	74.8	7.6×10^{-3}	930.1	12,426.5	19,885.7
	dijkstra	381	134532457	1,345.3	9.0	3,371.4	2,506.0	10,767.7
	qsort	5	18367911	183.6	0.2	671.8	3,657.5	10,755.8
	rijndael	58474	141568081	1,415.6	3,530.2	2,826.7	1,996.7	19,890.7
	sha	164	63752902	637.5	7.4	856.2	1,343.0	19,888.8
	stationary_tap	3003	36991685	369.9	1.6	61.3	165.8	535.7

wise improvement of a design at hand (cf. Section 5).

All experiments have been carried out for a fixed system frequency of 100MHz and using the following tools for simulation, cross-compilation, optimization, and code generation: g++ (GCC) 5.4.1, SystemC 2.3.1 with TLM 2.0.3, arm-none-eabi-gcc (GCC) 8.2.0, GNU Binutils 2.27.51, LLVM version 7.0.0, AMPL version 20111121, and Gurobi 8.1.0. We compare a real world gesture detection sensor firmware example (stationary_tap) to different exemplary benchmarks of the MiBench suite [34]. All considered examples are listed in Table 1 along with a basic description, the extent of the firmware in lines of code (LOC), and the required memory space after compilation with optimization level `-Os`. Please note that even though the focus in the following discussion is put on total values, the provided analysis results are not limited to that. That means, a more fine-grained evaluation in terms of timing and power is possible, even down to the level of single application profiles and memory instances.

Based on the firmware and the resulting binary, as possibly obtained from model-based generation, a baseline simulation and analysis round is carried out on the VSP in first place. The corresponding results are listed in Table 2. Timing simulation yields simulated cycles and accurate execution time values, whereas the memory power state machine provides energy and power figures, characterizing the memory subsystem. Striking, already in this table, is the considerable difference in peak power, which originates from the size of the memory instance that is attached to the VSP. Bitcount and stationary_tap get by with a memory block of 8K, while all other applications require 256K or more. As a consequence and in case of a peak power constraint violation, it can be worth to investigate the possibilities of a firmware memory footprint reduction already in this development stage.

Based on application binary and baseline analysis results, the memory subsystem optimization is carried out (cf. Section 2.2). On the example of the stationary_tap demo, this optimization step yields a memory allocation consisting of 5 memories (cf. Table 4) and the corresponding binding of all code and data blocks to these memories. Further, all memories are assumed to support an active operation mode (A) and two low-power modes, light sleep (L) and deep sleep (D). The corresponding low-power mode schedule for stationary_tap is given in Listing 6.

Table 4 Memory subsystem optimization statistics

	benchmark	memory count	time penalty	energy savings	peak power reduction
OPTIMIZATION (3)	bitcount	7	45.31%	39.28%	44.63%
	crc32	4	0.00%	0.02%	0.0%
	dijkstra	5	0.03%	75.41%	45.85%
	qsort	4	40.60%	7.31%	-0.01%
	rijndael	5	1.03%	78.69%	-0.03%
	sha	5	0.00%	83.16%	-0.02%
	stationary_tap	5	0.05%	67.45%	44.59%

Listing 6 Low-power mode schedule for stationary_tap

```
M1 M2 M3 M4 M5
A  L  A  D  D  __divsi3
A  L  A  D  D  __aeabi_idivmod
A  A  D  D  D  main
A  L  A  D  D  printf
A  L  A  D  D  stap_do_step
A  L  D  A  D  stap_initialize
```

All these results are finally fed back to the firmware assembly level and recompiled with a modified linker script in order to obtain a binary that reflects all optimization aspects. Based on that, a second simulation and analysis round on the VSP is executed, which yields the figures in Table 3 and Table 4. For the stationary_tap sensor application, a considerable energy consumption and peak power reduction at only a minimal timing penalty is achieved.

Altogether, it is clearly visible that energy savings as well as peak power reduction go hand in hand with a timing penalty, which is caused by additionally inserted code. In comparison to the baseline firmware, these code snippets are used for steering of the memory low-power mode configuration. In some cases, the resulting penalty is dispro-

portionately high, e.g. in case of bitcount and qsort benchmark (cf. Table 4). This is striking because the corresponding number of penalty cycles, due to actual mode changes is extremely low. A follow up investigation of these examples reveals that both applications make heavy use of pointer-based function calls, which are not traceable by the post-optimization code generation step. This feedback information can be used to adjust the source code accordingly, hence unneeded power mode management code can be avoided. By that, an improvement into the direction of considerable energy savings at negligible time penalty as for example given for dijkstra or stationary_tap demo becomes possible.

Another noticeable figure is the high number of penalty cycles that accumulated for the rijndael encryption benchmark. A closer co-investigation of optimized memory binding and low-power mode schedule reveals that two highly active but related functions have been separated by the optimization, a use case, comparable to the above presented example in Section 5. One option in order to reduce this penalty can be the feedback of this information to the firmware model, where a combination of these nodes will avoid the later separation. This way, the number of penalty cycles can be reduced, possibly in exchange for a decrease in energy savings. This trade-off can easily be evaluated by iterative workflow executions.

Summarized, these exemplary use cases show, how the set of information that is provided by our combined workflow can be used in a profitable way. Especially in the presence of timing and peak power constraints, this iterative approach is clearly a highly interesting option that allows the handy exploration of the design space in form of different trade-offs.

7 Conclusion

Motivated by the combined evaluation of different aspects in a top-down embedded system design workflow, this paper unites different contributions from model-based generation, memory optimization, and accurate timing simulation fields. The individual methods have been integrated with a virtual system prototype of an ARM-based sensor hub, which allows automated simulation, analysis, and optimization. Experiments for both, a real world application and representative benchmarks, prove the functionality and strength of this approach. Beyond the individual potential of the single workflow parts, it is the combination of analysis figures from different perspectives that allows new conclusions to be drawn. Especially in the presence of strict constraints, e.g. in terms of power or timing, such information is of high interest to the system designer. The continuation of this work will put the focus on the automation of such a design feedback loop for repetitive and step-wise improvement of a system design at hand.

Acknowledgment

This contribution is funded as part of the CONFIRM project (project labels 16ES0567, 16ES0568 and 16ES0569) within the research program ICT 2020 by the German Federal Ministry of Education and Research (BMBF) and supported by the industrial partners Infineon Technologies AG, Robert Bosch GmbH, Intel Deutschland AG, and Mentor Graphics GmbH.

References

[1] J.-P. Wolff, S. Stieber, T. Rankl, and R. Dorsch, "Improving always-on gesture recognition power efficiency for android devices using sensor hubs," in *Computational Science and Engineering (CSE) and IEEE Intl Conference on Embedded and Ubiquitous Computing (EUC) and 15th Intl Symposium on Distributed Computing and Applications for Business Engineering (DCABES), 2016 IEEE Intl Conference on*. IEEE, 2016, pp. 64–67.

[2] E. A. Lee and D. G. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[3] G. Bilsen, M. Engels, R. Lauwereins, and J. Peperstraete, "Cycle-static dataflow," *IEEE Transactions on signal processing*, vol. 44, no. 2, pp. 397–408, 1996.

[4] B. D. Theelen, M. C. Geilen, S. Stuijk, S. V. Gheorghita, T. Basten, J. P. Voeten, and A. H. Ghamarian, "Scenario-aware dataflow," *Technical Report ESR-2008-08*, 2008.

[5] F. Grützmacher, B. Beichler, C. Haubelt, and B. Theelen, "Dataflow-based modeling and performance analysis for online gesture recognition," in *Modelling, Analysis, and Control of Complex CPS (CPS Data), 2016 2nd International Workshop on*. IEEE, 2016, pp. 1–8.

[6] A. H. Ghamarian, S. Stuijk, T. Basten, M. Geilen, and B. D. Theelen, "Latency minimization for synchronous data flow graphs," in *Digital System Design Architectures, Methods and Tools, 2007. DSD 2007. 10th Euromicro Conference on*. IEEE, 2007, pp. 189–196.

[7] M. Geilen, T. Basten, and S. Stuijk, "Minimising buffer requirements of synchronous dataflow graphs with model checking," in *Design Automation Conference, 2005. Proceedings. 42nd*. IEEE, 2005, pp. 819–824.

[8] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Transactions on computers*, vol. 100, no. 1, pp. 24–35, 1987.

[9] J. Davis II, M. Goel, C. Hylands, B. Kienhuis, E. A. Lee, J. Liu, X. Liu, L. Muliadi, S. Neuendorffer, J. Reekie *et al.*, "Overview of the ptolemy project," ERL Technical Report UCB/ERL, Tech. Rep., 1999.

[10] J. L. Pino, S. Ha, E. A. Lee, and J. T. Buck, "Software synthesis for dsp using ptolemy," *Journal of VLSI signal processing systems for signal, image and video technology*, vol. 9, no. 1-2, pp. 7–21, 1995.

[11] M. C. Williamson and E. A. Lee, "Synthesis of parallel hardware implementations from synchronous dataflow graph specifications," in *Signals, Systems and Computers, 1996. Conference Record of the Thirtieth Asilomar Conference on*. IEEE, 1996, pp. 1340–1343.

[12] F. Menichelli and M. Olivieri, "Static minimization of total energy consumption in memory subsystem for scratchpad-based systems-on-chips," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 17, no. 2, pp. 161–171, 2009.

[13] F. Angiolini, L. Benini, and A. Caprara, "An efficient profile-based algorithm for scratchpad memory partitioning," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, no. 11, pp. 1660–1676, 2005.

[14] L. Benini, A. Macii, and M. Poncino, "A recursive algorithm for low-power memory partitioning," in *Proc. of the 2000 International Symposium on Low Power Electronics and Design (ISLPED'00)*, 2000, pp. 78–83.

[15] S. Srinivasan, F. Angiolini, M. Ruggiero, L. Benini, and N. Vijaykrishnan, "Simultaneous memory and bus partitioning for soc architectures," in *Proc. of the 2005 IEEE International SOC Conference*. IEEE, 2005, pp. 125–128.

[16] M. Strobel, M. Eggenberger, and M. Radetzki, "Low power memory allocation and mapping for area-constrained systems-on-chips," *EURASIP Journal on Embedded Systems*, vol. 2017, no. 1, 2016.

[17] L. Steinfeld, M. Ritt, F. Silveira, and L. Carro, *Low-Power Processors Require Effective Memory Partitioning*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 73–81. [Online]. Available: https://doi.org/10.1007/978-3-642-38853-8_7

[18] M. Strobel and M. Radetzki, "Design-time optimization techniques for low-power embedded memory subsystems," in *Proc. of the 1st International Workshop on Embedded Software for Industrial IOT (ESIIT)*, 2018.

[19] D. Mueller-Gritschneder, K. Lu, and U. Schlichtmann, "Control-Flow-Driven Source Level Timing Annotation for Embedded Software Models on Transaction Level," *2011 14th Euromicro Conference on Digital System Design*, pp. 600–607, Aug. 2011, publisher: Ieee Citation Key: Mueller-Gritschneder2011a ISBN: 978-1-4577-1048-3.

[20] S. Schulz and O. Bringmann, "Accelerating source-level timing simulation," in *Proceedings of the 2016 Conference on Design, Automation & Test in Europe*, ser. DATE '16. San Jose, CA, USA: EDA Consortium, 2016, pp. 1574–1579.

[21] J. Benz, C. Gerum, and O. Bringmann, "Advancing source-level timing simulation using loop acceleration," in *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. Dresden, Germany: IEEE, Mar. 2018, pp. 1393–1398.

[22] X. Zheng, L. K. John, and A. Gerstlauer, "Accurate phase-level cross-platform power and performance estimation," *Proceedings of the 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2016, iSBN: 9781450342360.

[23] X. Zheng, H. Vikalo, S. Song, L. K. John, and A. Gerstlauer, "Sampling-Based Binary-Level Cross-Platform Performance Estimation," pp. 1709–1714, 2017, iSBN: 9783981537086.

[24] S. Ottlik, S. Stattelmann, A. Viehl, W. Rosenstiel, and O. Bringmann, "Context-sensitive Timing Simulation of Binary Embedded Software," *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, pp. 14:1–14:10, 2014, iSBN: 978-1-4503-3050-3.

[25] Z. Wang and J. Henkel, "HyCoS," *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/-software codesign and system synthesis - CODES+ISSS '12*, p. 133, 2012, iSBN: 9781450314268.

[26] T. Meyerowitz, A. Sangiovanni-Vincentelli, M. Sauermann, and D. Langen, "Source-Level Timing Annotation and Simulation for a Heterogeneous Multiprocessor," pp. 276–279, 2008, iSBN: 9783981080131.

[27] S. Ottlik, J. M. Borrmann, S. Asbach, A. Viehl, W. Rosenstiel, and O. Bringmann, "Trace-based context-sensitive timing simulation considering execution path variations," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. Macao, Macao: IEEE, Jan. 2016, pp. 159–165. [Online]. Available: http://ieeexplore.ieee.org/document/7428005/

[28] S. Ottlik, C. Gerum, A. Viehl, W. Rosenstiel, and O. Bringmann, "Context-sensitive timing automata for fast source level simulation," in *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. Lausanne, Switzerland: IEEE, Mar. 2017, pp. 512–517. [Online]. Available: http://ieeexplore.ieee.org/document/7927042/

[29] S. Stattelmann, O. Bringmann, and W. Rosenstiel, "Fast and accurate source-level simulation of software timing considering complex code optimizations," *Proceedings of the 48th Design Automation Conference*, pp. 486–491, 2011, iSBN: 9781450306362.

[30] Bosch Sensortec GmbH, "BMF055 Application Note," Tech. Rep., 2016. [Online]. Available: https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST-BMF055-DS000.pdf

[31] SoCLib Consortium *et al.*, "The soclib project: An integrated system-on-chip modelling and simulation platform," CNRS, Tech. Rep., 2003, last visited 01/14/2019. [Online]. Available: http://www.soclib.fr

[32] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi, "Cacti 6.0: A tool to model large caches," HP Laboratories, Tech. Rep. HPL-2009-85, 2009, last visited on 01/10/2019. [Online]. Available: http://www.hpl.hp.com/techreports/2009/HPL-2009-85.pdf

[33] L. Minwell, "Advanced power management in embedded memory subsystems," 2011, last visited on 01/20/2019. [Online]. Available: https://www.design-reuse.com/articles/26402/power-management-in-embedded-memory-subsystems.html

[34] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proc. of the 2001 IEEE International Workshop on Workload Characterization*. IEEE, 2001.