Automatic Test Coverage Measurements to support Design Space Exploration

Jasmin Jahic[†], Thiyagarajan Purusothaman[*], Markus Damm[*], Thomas Kuhn[†], Peter Liggesmeyer[*], and Christoph Grimm[*]

[*]TU Kaiserslautern, Germany
{liggesmeyer|grimm|damm}@cs.uni-kl.de, purusoth@rhrk.uni-kl.de
[†]Fraunhofer IESE, Kaiserslautern, Germany
{Thomas.Kuhn|Jasmin.Jahic}@iese.fraunhofer.de

Abstract. Functional interferences are a significant challenge when federated architectures are transformed to integrated architectures. While there exist numerous software measures that could be used to resolve or prevent functional interferences, the impact of a particular measure is unknown, because it cannot be tested with current approaches. Therefore, important architecture level decisions are often taken based on opinions instead of facts. Substantiating decisions with facts requires the execution of software components in simulated environments. While it is certainly possible to execute individual software runnables, the creation of contexts that reflects architecture decisions is tricky. We therefore apply our FERAL framework that we have extended to support the collection of test metrics to ensure credibility of obtained results. In this paper, we describe our novel LLVM based approach for the supervised execution of software runnables, and the collection of test coverage metrics. To illustrate the applicability of our approach, we present the supervised execution of a device driver.

Keywords: Design Space Exploration, LLVM, Cyber Physical Systems, Metrics, Coverage Analysis

1 Introduction

Nowadays, software is the main driver for innovations in embedded systems. By controlling most safety relevant systems, embedded software has a major impact on the operation and dependability of these systems. A recent challenge for embedded software developers is the ongoing transition from federated architectures to integrated architectures. Federated architectures execute independent functions (mostly) on independent hardware devices. This prevents unwanted interferences, and supports fault isolation. Due to increasing resources that are available on multi-core processors and potential cost savings, developers however focus more and more on integrated architectures that consolidate independent software functions on shared hardware platforms.

However, consolidation of formerly independent functions may lead to interferences. Existing approaches that address the freedom from interference for software runnables (cf. [9], [2]) mostly focus on temporal properties like execution times. While this is an important aspect, functional interferences are of similar importance in integrated architectures. They occur for example when the same resource, e.g. an actuator, is used concurrently by multiple independent runnables. Software and system architects usually have multiple options for avoiding these functional interferences. The concrete impact of a particular measure to functional and temporal behaviour is however often unknown at this time. Architects therefore have to rely on their experience when taking decisions, when the impacts of decisions are getting evident after they have been implemented, significant changes are often time consuming and expensive. To address this challenge, it is necessary to support architects during design decision activities by predicting the behaviour of systems and the impact of relevant decisions early.

In this paper, we propose the prediction of functional behaviour on system level through supervised execution and simulation techniques. Simulation enables rapid feedback regarding the impact of architecture decisions by evaluating system behaviour in virtual deployment scenarios. This way, possible interferences and the impact of architecture frameworks that define measures for handling potential interferences are evaluated. To ensure significant coverage of runnable behaviour and therefore credibility of simulation results, our approach extracts test coverage metrics. Furthermore, in its current state, our extension enables collection of additional metrics that enable evaluation of the behaviour of software runnables. We present an extension to FERAL that enables the supervised execution of embedded software runnables implemented in C and C++. The runnables are compiled to the assembler-like intermediate representation code of the LLVM compiler infrastructure [14], which is then interpreted under FERAL supervision. Environment components are modelled with the C++ based simulation framework SystemC.

The rest of this paper is organized as follows: Section 2 surveys and discusses technical basics and related work. Section 3 gives an overview on our supervised testing approach. Section 4 provides more details on the LLVM-based implementation of our approach. In Section 5 a proof-of-concept implementation of a PWM (Pulse-Width Modulation) driver is shown with some supervised testing results before concluding in Section 6.

2 Related Work

In this section, we first introduce necessary technical background regarding the FERAL simulation framework, the LLVM compiler infrastructure, and SystemC for the realization of our supervised testing approach. We then discuss related approaches for design space exploration in context of our work.

2.1 The FERAL framework

FERAL is our Framework for Efficient simulator coupling on Requirements and Architecture Level [13]. It enables the integration of specialized and focused simulators with heterogeneous models of computation and communication into one holistic simulation scenario. Individual simulators are integrated as simulation components and placed under the control of a hierarchical tree of directors that define the semantics of the integrated simulation. These components represent for example environmental simulators, network simulators, or behaviour simulation. The LLVM based supervision infrastructure presented in this paper is added as a new simulation component for FERAL to enable the early and supervised execution of C/C++ runnables within the framework.

2.2 Analysis approaches using LLVM

LLVM [14] is a compiler infrastructure that implements an abstract assembler-like language known as the *LLVM Intermediate Representation* (LLVM IR). Code written in an arbitrary programming language is first transformed by an appropriate front-end into LLVM IR, which in turn is then transformed to the assembly code of the target processor by a suitable back-end.

LLVM IR is assembler-like in the sense that it has the typical, simple instruction set of an assembly language. In fact, the instruction set resembles a RISC system with load/store architecture. These simple instructions, however, operate on an unbounded set of *typed* registers resembling more to variables in a higher programming language. I.e. present are basic types like integers and floats, as well as derived types like pointers, arrays or structures.

The LLVM infrastructure has several advantages. For example, supporting a new target processor just requires writing a new back-end, since LLVM front-ends are target-independent. The LLVM IR uses Static Single Assignment (SSA) which is particularly well suited for optimization. Applications build from sources in several higher level languages can be optimized as a whole on the LLVM IR level. LLVM can also be used as a platform-independent virtual machine by means of interpretation or Just-In-Time compilation (JIT).

Apart from these features the open structure of LLVM has attracted scientists to find other uses of the LLVM infrastructure besides code compilation and optimization. Some of these applications are in the field of system modelling. In [6], an LLVM backend is used to generate transaction-level SystemC models out of C code. These models represent the execution of the corresponding program on a given target architecture, e.g. by the inclusion of wait-statements. In [12] the LLVM IR is used to analyse SystemC models with mixed register-transfer and transaction level components. System models using the Communicating Sequential Processes (CSP) model of computation are verified in [4] using a transformation to LLVM IR.

2.3 SystemC and SystemC AMS

SystemC is an industry-standard [1] language based on C++ for *electronic system-level* (ESL) design, modeling and simulation. It enables the creation of abstract architectural descriptions of digital hardware/software (HW/SW) systems, also known as *virtual prototypes*. An important concept in SystemC, especially regarding virtual prototypes, is *Transaction Level Modeling* (TLM) [5], which provides a high abstraction level for communication modeling. When simulating the communication over a bus, for example, not every individual bit is processed. Instead, *transactions* are used which capture the transfer of arbitrary large chunks of data at once via single method calls. Timing is only coarsely modeled. It is possible to decouple different parts of the model in time and only synchronize at certain points.

The modeling of embedded systems with mixed physical components are facilitated by SystemC *Analog/Mixed-Signal* (AMS) extensions [10]. To comply with demanding requirements and use cases (e.g. in automotive applications), new execution semantics and language constructs are being defined to model a more reactive and dynamic behavior. The different Models of Computation (MoC) of SystemC AMS are used to enable a tighter time-accurate interaction between the AMS signal processing and control domain while keeping the modeling and communication abstract and efficient.

2.4 Related Design Space Exploration Approaches

Astrée [7] is a static code analysis tool from AbsInt that is able to perform static analysis based on software that was developed using the C programming language to ensure a correct implementation and quality of delivered code. It highlights potential faults like division by zero errors, array bound violations, numeric overflows, invalid pointer operations, unreachable code, and uninitialized variables. In contrast to test based approaches, Astre performs static symbolic execution. It requires the annotation and changing of source code in some cases, for example when recursive algorithms are used. It is also able to prove validity of conditions, if the state space is fully covered by the analysis algorithm, which is not possible with test based approaches.

The aiT [8] tool from AbsInt enables worst case execution time (WCET) analysis. To achieve credible results, aiT requires the fulfilment of several preconditions. Analysed runnables need to be executed in an undisrupted environment - other runnables that are executed on the same or other processor cores must not change the state of the analysed runnable. This often requires a segregated execution of runnables, i.e. an execution that is independent of any possible influence from other runnables that are possibly (concurrently) executed on the same platform. Our supervised testing approach is able to check if the preconditions for aiT are provided by a particular HW/SW architecture through analysis of bus, resource, and memory accesses, and thus complements the use of this tool.

SymTA/S [9] from Symtavision is a timing analysis tool that enables analysis of relevant bus and processor timing in early development stages to support design space exploration in deployment scenarios. The tools chronSIM and

chronVAL [2] from Inchron serve a similar purpose. These tools support analysis of timing properties, but do not support early evaluation of freedom from interference with respect to functional properties.

Virtual prototyping tools like Synopsis platform architect, Simics [15] or OVP [11] enable the execution of code in context of a accurately simulated virtual hardware platforms. These tools are often used for the development of runnables when target hardware is not yet available. The potential of using these solutions for early and rapid evaluation of architecture decisions is limited, because hardware platforms need to be modeled in a much greater level of detail than it is necessary for FERAL. Furthermore, platform specific software implementations need to be available, functional models are not sufficient in this case.

Summarizing the findings of the discussion, one can conclude that tools and related approaches exist for design space exploration that focus either on the prediction of scheduling, communication, and execution timing. While timing is certainly an important property of systems and therefore of high importance to software and system architects, the impact of architecture decisions to functional behaviour is at least as significant, and needs to be considered as well during design space exploration. Existing virtual prototyping approaches are either bound to specific system architecture or require a complete implementation for testing that is far beyond the functional models that are available at the time at which architecture decisions take place.

3 Supervised Testing Approach

Our supervised execution approach relies on the interpretation of LLVM intermediate representation (IR) code in a simulated environment. Compared to other instruction set simulators, the LLVM interpreter has significant benefits: LLVM IR retains information about the source code through the compilation process that enables traceability between LLVM IR code and source code statements. Even when accessing variables via pointers during program execution, it is possible to obtain the target variable of the operation including not only its memory location, but also relevant type information and its full qualified name. This turns the LLVM into a promising candidate for supervised program execution. A second reason for choosing LLVM is its ability to execute C code on abstract platforms. Therefore, no sophisticated platform software or operating system code is required to execute runnable code in a virtual environment.

3.1 LLVM IR structure and LLVM Interpreter

The logical root of a program in LLVM IR is the *module*, which is one translation unit of the input program. LLVM Modules are composed of functions, global variables and symbol table entries. Each function contains a number of connected basic blocks that form the Control Flow Graph (CFG) as shown in Figure 1. Each basic block represents a single entry/single exit-section of the code. After processing of one block is finished, execution continues with one of the next

linked blocks depending on a given condition. Blocks consist of instructions. For each instruction, LLVM IR allows metadata to be attached to it by the language front-end, e.g. source-level debug information (descriptors for types, variables, functions, etc.). Source-level information can be used for various purposes like optimization or execution tracking. LLVM IR code is executed by the LLVM

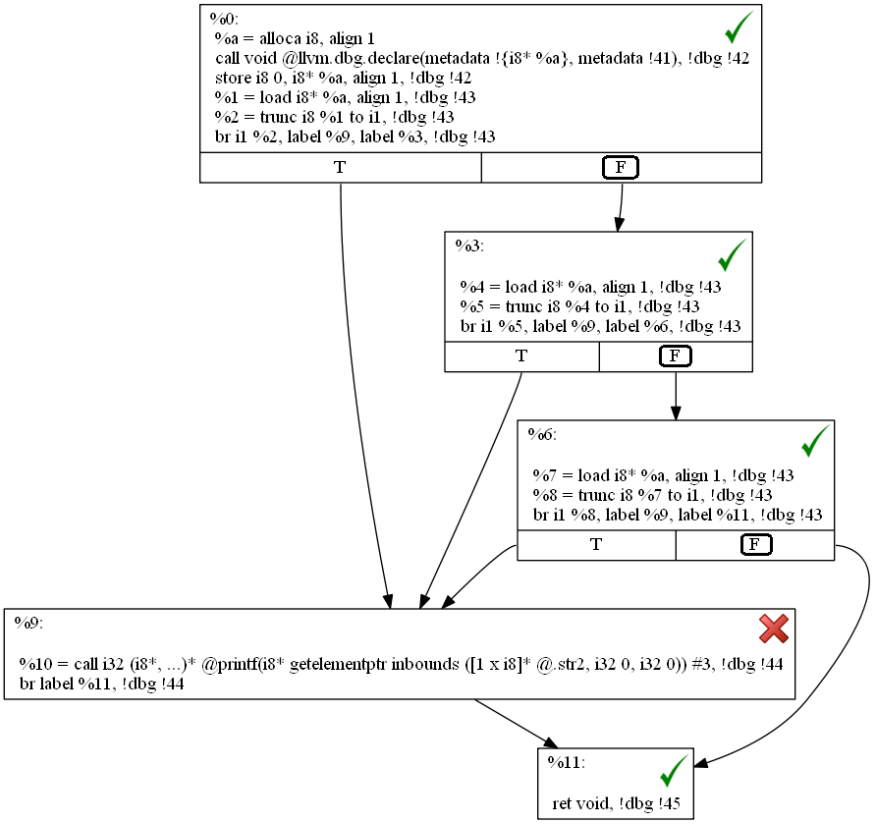

Fig. 1. Example of function structure in LLVM IR

interpreter LLI that is provided by the LLVM project. While interpreting an executable, the LLI dynamically binds functions that are called by the executable to matching functions of the host. We use this capability to connect the LLI to FERAL and SystemC to provide realistic execution environments for executed runnables. An execution environment consists of higher level services like access to simulated networks that are realized by other simulation components, as well as simulated hardware registers. The latter for example simulate low level devices like hardware timers that are required for the implementation of schedulers. Using this approach, micro controller abstraction layers (MCAL) can be quickly

created for given simulation scenarios. The use of an MCAL is common in the embedded systems domain, standards like AUTOSAR [3] define standardized MCALs. This way, runnables may be compiled into LLVM IR, e.g. using the LLVM C/C++ compiler Clang or GCC without having to introduce any changes to the application level source code.

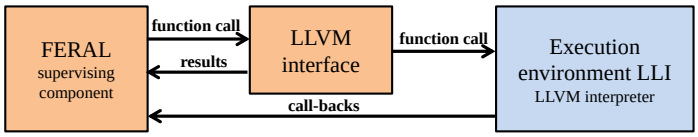

Fig. 2. Components of supervised execution

3.2 Supervised execution

The main components of our supervised execution approach are illustrated in Figure 2. The FERAL simulator is the supervising component. It controls the execution of each LLVM instance according to a time triggered execution model that periodically triggers execution of a given number LLVM instructions per time interval. Communication with the LLVM infrastructure and LLI is realized via an LLVM interface. This interface controls the execution of the LLI interpreter that supervised the execution of the runnable under test. Events from the LLI, such as the end of a time frame, a function that has been called by the runnable, a variable access, or an access to a simulated register, is reported via call-back functions to the FERAL framework.

When the runnable execution is initiated, the LLI interpreter executes the LLVM IR code instruction by instruction, as a real processor would be executing machine code. It also tracks the execution path, variable accesses, and invoked functions. To collect run time information about executed runnables, we have adapted the LLVM interpreter LLI. The main modification of LLI was a change of the main loop that is used for instruction processing. Our extension stores the following information for every instruction that is executed, where part of this information is obtained from debug information that is inserted by the used compiler:

- Statement line number in the source code
- Basic block of the instruction
- Parent function of the instruction

In addition to the common data that is obtained for every instruction type, we collect additional data that depends on the type of an LLI instruction. When a function is invoked, the function parameters are stored in addition to the commonly collected data. For load/store instructions, the variable value read or written is stored, as well as the name and virtual memory address of the

operation target. For normal variable access operations the variable name is known to LLI. Targets of pointer arithmetic need to be resolved using Hashtables. This way, the target of pointer operations are resolved to concrete variables. This information is passed via call-back functions to the FERAL framework. Based on the memory location of an executed load/store instruction, the FERAL framework decides whether a bus access will be necessary or not, and which parts of the processor interconnects will be required to perform this access. By evaluating this information, evidence about variable and bus access is created.

In addition to bus/register accesses and variable/function accesses, additional metrics are derived from the collected data. Based on the data that is obtained by LLI, FERAL evaluates the following:

- Functions, algorithms and loops accessing a particular variable
- Frequency of variable access, executed loops and function calls
- Unused variables of the runnable

This information is interesting for the early tuning of functional behaviour to integrated architectures. For example, it is possible to deploy runnables in a manner that prevents interferences from independent runnables that use the same buses or resources. Real-time scheduling operating systems may be used to implement schedules that use time division techniques to isolate potentially concurrent accesses temporally. Prediction of functional behaviour with FERAL enables the prediction of the impact of these measures on system level.

3.3 Coverage metrics

To assess the credibility of collected results for design space exploration, it is necessary to know the extent of the coverage that has been achieved while executing the runnables. During supervised execution, the supervising FERAL framework receives data about executed instructions via call-backs. This includes the corresponding source code statements, basic blocks, and functions, respectively. Based on this information, executed statements, paths and branches for each runnable are derived. The static structure of each runnable, which serves as a baseline for coverage analysis is provided by the LLVM IR as well. It has a graph structure with branches representing conditional statements in the source code (see Figure 1). Each condition is the beginning of a new basic block, and each block has information about its successors and predecessors.

A block with two successors represents a conditional block, with the first successor corresponding to an evaluation to the value true. Each basic block contains a mapping to a distinct, uninterrupted part of the source code. Therefore we calculate coverage metrics based on basic blocks instead of instructions to reduce overhead. By analysing the code structure and the collected runtime data, we identify basic blocks that were executed and those who were not, as well as their order of execution. Since every invocation is reported, FERAL is also able to trace how often a block has been executed, e.g. as part of a loop, together with the predecessors and successors of each execution. This enables us

to not only trace the execution of individual blocks, but also follow the paths that connect blocks to determine execution paths of the runnable that were covered by the supervised execution. An example for one function is illustrated in (Figure 1), where the structure is visualized using opt and dot tools, which both are included in LLVM. Based on the execution report we ticked executed blocks and encircled the evaluation results of the conditions. A block not executed is marked with an X. By evaluating this information, it is possible to obtain several coverage metrics:

- Functions not called (function coverage)
- Statements not executed (statement coverage)
- Branches not executed (branch coverage)
- Paths not executed (path coverage)

The data gathered can be used to calculate numerous further test metrics. Ongoing work is to derive MC/DC (Modified Condition/Decision Coverage) coverage metrics, which is necessary for the validation of safety critical aviation software as specified by RCTA/DO-178B standard. It is referenced by many other standards for the development of safety relevant software as well.

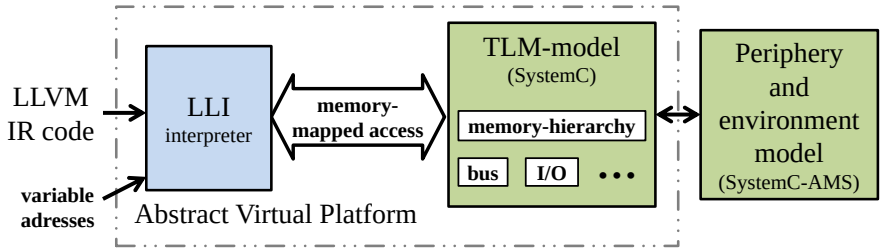

Fig. 3. Connecting LLI with the SystemC model

3.4 Supervised execution with SystemC transaction level models

For a meaningful supervised execution of embedded software, realistic input is needed. Cyber physical systems typically take their inputs from the environment, e.g. from sensors or through communication with other systems. Generated outputs often will influence subsequent inputs; e.g. in control loops. Therefore, for collecting data about the behavior of the runnables, a simulated environment is needed taking all these factors into account.

SystemC has been successfully used in similar contexts in the past to simulate system environments and low-level behavior. One example for simulating cyber physical systems controlling physical processes in a wireless network is provided by [16]. In general, using transaction-level SystemC models together with an instruction set simulator (ISS) to form virtual platforms is common practice

today. Therefore we complement the LLVM-based supervised testing approach presented above with a SystemC model simulating the hardware platform as well as environmental processes (see Figure 3), with LLI used as an ISS.

Since LLVM IR uses a load/store instruction set, the approach of connecting the LLVM interpreter LLI to a SystemC TLM model is to trap load- and store-instructions that access parts of the system that are simulated by SystemC. We forward these accesses to the SystemC simulation model by pairing for example variable names with the corresponding addresses on the TLM side, which can be set deliberately, or may e.g. be extracted from an executable. Each time LLI executes a load- or store instruction on a variable covered in the lookup table, the resulting memory access is therefore directed to the TLM model.

That way it is possible to forward only accesses to selected variables in the LLVM IR code to the TLM model. Most variables that are not related to System C parts of the simulation are handled locally by the LLVM or by other higher level components of the FERAL framework, which enables a very accurate selection of the necessary level of abstraction. This significantly lowers the required simulation time and is a prerequisite for simulation of complex system architectures.

4 Realization

The supervising component has been realized as a new Simulation Component. This is the common approach for adding new simulators to the FERAL framework. The changes that were necessary to integrate the LLVM interpreter LLI are minimal. LLI handles instructions by their type by implementing a specialized handling function. When necessary, it is possible to intercept instructions either in the main LLI loop or in the handling function. Intercepting and reporting *every* instruction of the runnable back to FERAL would be too time consuming. We therefore only report selected instructions:

- Branching - entering a new basic block
- Function calls and returns
- Memory manipulations - load and store instructions

From these instructions, we extract all the information needed for the test metrics. The second change to LLI is related to the development of an interface to the FERAL framework. We have implemented several call-back functions that enable communication with the FERAL framework. The FERAL framework also needs to explicitly invoke the execution of LLI. Here, we have decided for a time triggered approach that resembles the behaviour of real processors. FERAL requests the execution of a number of LLI instructions that resembles a given simulation time based on the MIPS rating of the target platform. Runtime information is reported via call-back functions. Depending on the FERAL director that controls the LLVM domain, execution of other feral components is suspended during that time, or other (independent) components are executed

concurrently. When LLI finishes executing the requested amount of instructions, control is passed back to the FERAL framework.

If a runnable is compiled with debug information, we can tell at runtime which statement of the source code is being executed. However, the mapping between statements, LLVM blocks, LLVM IR instructions, and source lines imposes challenges: Statement line numbers are not present in the basic block data structure. Since the Meta data is bound to instructions, they have to be calculated based on the instructions in the block instead. Consequently, multiple basic blocks of the LLVM IR can share the same line in the source code, and basic blocks can start and end in the same line of the source code as well.

When communicating with developers, it must be considered that one statement of source code may be compiled to numerous LLVM IR instructions and basic blocks. Therefore, the same statement can be the beginning of one basic block as well as the end of other basic blocks. While this is not problematic for the calculation of the current coverage metrics (since those depend on LLVM blocks only) it is important for the presentation of results to the developers. Statements that belong to multiple blocks may have been executed only partially, which is explicitly marked in generated reports.

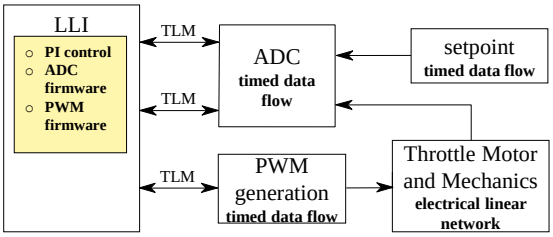

Fig. 4. Example system

5 Application Example

As a proof-of-concept, a throttle valve controller was implemented. A throttle valve controller is an actuator component in the intake manifold of an internal combustion engine. It consists of a DC motor, a gear assembly, potentiometer for position feedback and a closing plate supported by a return spring. It controls the air intake and a mechanism by which the power or engine speed is regulated. The opening and closing of the plate is determined by the PWM input value. Therefore, a PI controller is used to regulate its position. The position of the throttle plate is given by the potentiometer position feedback, and we use the load current as a representative of the system behavior of the throttle valve.

An overview of the model with PWM, ADC (Analog-to-Digital Converter) and PI (Proportional-Integral) control loop is shown in Figure 4. The PWM

and PI control loop was executed with the set point changing at certain points in time. The current position of the throttle valve is sampled from the throttle motor mechanics and passed to the controller software via an ADC. The software then controls the PWM generator via control registers. In the source code, the PWM control registers (`REG_PWM_PRSC`, `REG_PWM_INIT` and `REG_PWM_DUTY` for prescaler, initial counter value and duty cycle, respectively) and the output registers of the ADC (`REG_ADC1` and `REG_ADC2`) are simple variables which are mapped to the corresponding registers in the TLM model with the approach described in Section 3.4. During the execution, the supervising component reports access to these variables, for example:

```
Store REG_ADC1 in 65721072
Store REG_ADC2 in 65721088
Load REG_ADC1 from 65721072
```

The execution report contains the list of functions not called and statistic about the statements, e.g. for this application:

```
0 from 7 functions is not called. Function coverage is: 100.0%
Not executed statements of the runnable per function:
Function: _Z10pidControld
 Statement: 11 NOT EXECUTED.
 Statement: 13 NOT EXECUTED.
 ...
```

The execution path of each function call is recorded. For this application, we reconstruct the execution path of the whole runnable:

```
Order of execution:
 Function: main block: 67675168 START: 68 END: 89
 Function: _Z10pidControld block: 67631296 START: 12 END: 12
 Function: _Z10pidControld block: 67645832 START: 14 END: 14
 Function: main block: 67675216 START: 91 END: 93
 ...
```

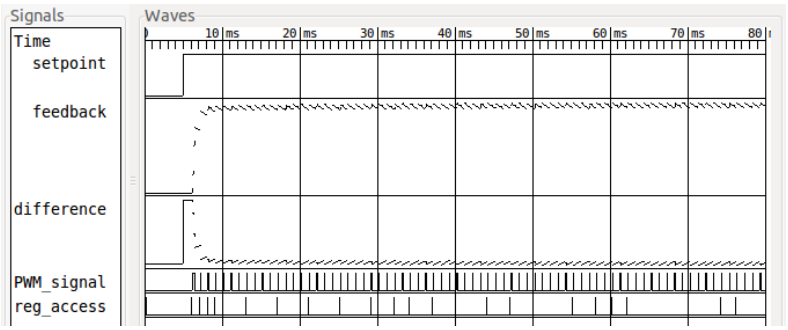

Fig. 5. Example simulation trace

Figure 5 shows the trace of an example simulation run. The software in this specific example is quite simple and essentially consists of the following steps:

1. Read setpoint and PWM-output from `REG_ADC1` and `REG_ADC2`, and compute their difference `P` (the error)
2. Update the integral error `I += P`
3. Compute the controller output `u = P * K_p + I * K_i`, where `K_p` and `K_i` are constants (the proportional and the integral gain, respectively).
4. Saturate `u` between 0 and 1.
5. Compute and write the new value for the (16 bit) duty-cycle register `REG_PWM_DUTY = u * 65535`

This code is executed repeatedly every millisecond, scheduled by the execution supervisor. A standard design space exploration activity here is fine-tuning the controller by finding appropriate values for `K_p` and `K_i`, e.g. to minimize overshoot or optimize response time. With our approach we can now also try to optimize the software to minimize bus access. If we write to `REG_PWM_DUTY` every time the controller code is executed, the bus is utilized 9.62% of the time during the simulation. This includes all accesses to PWM- and ADC registers. We can now store a local copy of the current value of `REG_PWM_DUTY` in a local scratch-pad memory, and only write a new value to `REG_PWM_DUTY` if it differs from the current one by a certain amount `DIFF`. Figure 6 shows the simulation results for two different PWM configurations.

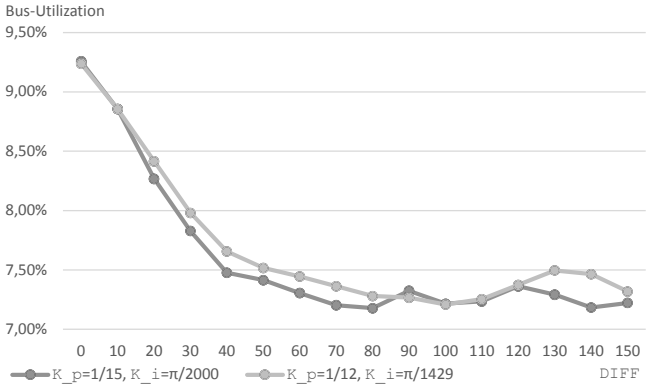

Fig. 6. Different bus utilization percentages depending on `DIFF`

By choosing `DIFF=80` we can lower the bus utilization of the controller to 7.18% while not sacrificing quality. In fact, Figure 5 shows a portion of the simulation run of this setting; at the bottom it is shown when `REG_PWM_DUTY` is accessed. Going beyond the `DIFF=80` margin actually doesn't gain more savings, but it turned out that significant loss of quality occurred only when setting `DIFF` close to 1000, resulting in oscillation. Note since `REG_PWM_DUTY` is an I/O register,

access to it will be independent regardless of the compiler infrastructure actually used for the final software deployment.

6 Conclusion and Future Work

In this paper we presented our LLVM based approach for the supervised execution of embedded software. It supports design space exploration by devising runtime information with respect to functional behavior. With our approach, it is possible to execute a software early in a virtual deployment scenario and to predict the impact of architectural decisions to software and system behavior early. To measure the credibility of achieved information, our approach enables the collection of coverage metrics. Furthermore, additional metrics like variable access, bus accesses and device access may be collected. The overall execution is driven by our FERAL simulation framework that enables the synchronization of all system components; hardware devices are simulated by SystemC.

In summary, collecting the coverage measurements without changing the source code was the greatest challenge and corner stone for the contribution. In practice, this meant establishing the communication interface between FERAL and LLVM, which are developed using two different programming languages (Java and C++ respectively). Second practical challenge was extraction of relevant information from the coverage measurement. From huge amount of information gathered during the supervised execution, only those relevant to support parallelization and deployment decisions were extracted and later combined.

We believe that our combination of simulation and execution of software on an abstract LLVM-based ISS could prove to be fruitful in several ways: First quantitative metrics like expected bus traffic can be collected on a high level even before considering a concrete hardware platform. For multicore systems, architectural options like crossbar and memory configuration can be determined based on these metrics. Second, coverage metrics ensure that simulation setups are chosen such that they cover the full behavior of runnables. Last but not least, application behavior can be evaluated early without having an operating system in place since all the important functions of the OS are performed by the FERAL framework.

Future work will concentrate on multicore specific aspects like detecting additional interferences on shared resources as memory, interconnects or peripherals. This would facilitate architectural exploration of the processing platform on a high abstraction level.

7 Acknowledgments

Part of this work was funded within the project ARAMiS by the German Federal Ministry for Education and Research with the funding IDs 01IS11035. The responsibility for the content remains with the authors.

References

1. Accellera Systems Initiative. *SystemC AMS Accellera Standard for Analog/Mixed-signal (AMS) Language Reference Manual, Release 2.0*, 2013.
2. S. Anssi, K. Albers, M. Dörfel, and S. Gérard. chronval/chronsim: A tool suite for timing verification of auto-motive applications. *Proc. Embedded Real-Time Software and Systems, ERTS*, 2012.
3. G. AUTOSAR. Autosar–technical overview v2. 0.1, 2006.
4. B. Bartels and S. Glesner. Verification of distributed embedded real-time systems and their low-level implementations using timed csp. In *Software Engineering Conference (APSEC), 2011 18th Asia Pacific*, pages 195–202, 2011.
5. L. Cai and D. Gajski. Transaction level modeling: an overview. In *Proceedings of the 1st IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 19–24. ACM, 2003.
6. E. Cheung, H. Hsieh, and F. Balarin. Memory Subsystem Simulation in Software TLM/T Models. In *ASP-DAC '09: Proceedings of the 2009 Conference on Asia and South Pacific Design Automation*, pages 811–816, Piscataway, NJ, USA, 2009. IEEE Press.
7. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *Programming Languages and Systems*, pages 21–30. Springer, 2005.
8. C. Ferdinand and R. Heckmann. ait: Worst-case execution time prediction by static program analysis. In *Building the Information Society*, pages 377–383. Springer, 2004.
9. R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst. System level performance analysis–the symta/s approach. *IEE Proceedings-Computers and Digital Techniques*, 152(2):148–166, 2005.
10. IEEE. *IEEE Standard for Standard SystemC™ Language Reference Manual*, 2011. IEEE Std. 1666-2011.
11. Imperas Software Limited. OVPsim and Imperas CPU Manager User Guide, 2010.
12. A. Kaushik and H. D. Patel. Systemc-clang: An open-source framework for analyzing mixed-abstraction systemc models. In *Specification Design Languages (FDL), 2013 Forum on*, pages 1–8, 2013.
13. T. Kuhn, T. Forster, T. Braun, and R. Gotzhein. FERAL - Framework for Simulator Coupling on Requirements and Architecture Level. In *Eleventh ACM-IEEE International Conference on Formal Methods and Models for Codesign*, October 2013.
14. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.
15. P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, 2002.
16. J. Moreno, M. Damm, J. Haase, C. Grimm, and E. Holleis. Unified and comprehensive electronic system level, network and physics simulation for wirelessly networked cyber physical systems. In *Specification and Design Languages (FDL), 2012 Forum on*, pages 68–74. IEEE, 2012.