A Workload Extraction Framework for Software Performance Model Generation

Philipp Ittershagen, Philipp A. Hartmann, Kim Grüttner, Wolfgang Nebel
OFFIS – Institute for Information Technology
Oldenburg, Germany
{ittershagen,hartmann,gruettner,nebel}@offis.de

ABSTRACT

The early performance evaluation of complex platforms and software stacks requires fast and sufficiently accurate workload representations. In the literature, two different approaches have been proposed: *Host-based simulation* with abstract performance annotations, enabling fast and functional simulations with limited architectural accuracy, and *abstract workload models* (or traffic generators) with more detailed platform resource usage patterns.

In this work, we present an approach for automatic workload extraction from functional application code, combining the benefits of both approaches. First, the algorithmic behaviour of the embedded software is characterised statically both in terms of target processor usage *and* target memory access patterns, resulting in an abstracted, control flow-aware workload model. Secondly, this model can be used on the target architecture itself as well as within a host-based simulation environment. We demonstrate the effectiveness of our approach by running our performance model on a virtual platform with and without a target Instruction Set Simulator (ISS) and comparing the simulation traces with the unaltered target processor binary execution.

1. INTRODUCTION

In today's Multiprocessor System-on-a-Chip (MPSoC) designs, a key concept for obtaining a high feature density is the utilization of shared resources. Different actors of a system often share a single interconnect and peripherals due to cost, power or space constraints.

However, the estimation of the application's performance behaviour on an MPSoC becomes increasingly difficult due to this interfering usage of shared resources. Therefore, the designer needs to be able to evaluate an application in terms of its *observable shared resource usage*, i.e. activities generated from actors in the system which are visible to other actors and might influence their temporal behaviour. Analytical as well as state-based formal approaches, trying to derive resource usage patterns of shared peripherals (e.g. Fig. 1) are practically of limited use, either due to their over-abstraction or the lack of scalability, leading to a state-explosion. Targeting state-of-the-art MPSoCs, considering data-dependent application scenarios, we have chosen a simulation-based approach for estimating the application's observable shared resources usage instead.

RAPIDO'15, January 19 - 21 2015, Amsterdam, Netherlands

ACM 978-1-60558-699-1/15/01 …$15.00
http://dx.doi.org/10.1145/2693433.2693436.

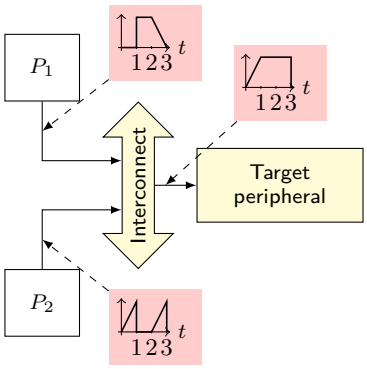

Figure 1: Shared resource usage in an MPSoC scenario.

To evaluate the chosen platform components, the software stack and its configurations, accurate characteristic workloads of the applications to be deployed on the platform have to be available. A possible view on the resource usage patterns of the application is enabled by cross-compiling and simulating the application binary on a virtual platform containing an Instruction-Set Simulator (ISS) of the target processor. This simulation approach precisely represents the functional behavior, is capable of providing a high temporal accuracy and even represents implicit shared resource usage, such as access to a shared memory controller. But at the same time, it requires a complex simulation environment containing many of the target platform properties and has a moderate simulation performance.

In many cases, the precise functional behaviour of the application code is of limited interest when exploring the impact of mapping and platform configurations on shared resource contention influencing throughput, utilization and quality of service. In early stages of the design flow, it is sufficient to use a *workload model* to perform shared resource usage estimation and analysis. Such a model can be constructed by statically analysing the application's behaviour and replacing parts of the application's functionality with a representation, while still preserving the data-dependent control flow of the application. Furthermore, *host-based* simulation techniques enabling fast and sufficiently accurate es-

timations are often used to gain initial resource usage traces from a functional application model running on abstracted resources of the target platform. For this purpose, the application is instrumented with estimated execution times and shared resource access behaviour, natively executed on the designer's host machine. The annotated time is consumed in a simulation model of the platform.

The contribution of this paper is two-fold. First, a method for statically characterizing and extracting target-specific basic block properties is described. These are derived from the properties of the resulting target code and include local CPU usage as well as memory access patterns. Secondly, the extracted characterization properties are then processed by a back-end, applying the corresponding substitution and instrumentation rules in the final code generation process.

Our analysis and simulation framework enables the analysis of interfering shared resource usage patterns along with local delays using host-based simulation or target binary ISS virtual platforms. We show that it is possible to generate a binary, suitable for a host-based simulation by adding timing and shared resource access instrumentation points to the code, as well as generating a target-binary workload representation by substituting certain parts of the characterized target code with their workload representations regarding its observable shared resource access behaviour on the platform. Our evaluation (Sec. 4) shows, that our generated workload models are very close ($< 6\%$ deviation) to the observed behaviour of the original application code on the target platform, while enabling a significant speedup in the host-based simulation.

2. RELATED WORK

Several approaches for measuring the performance of architectural and platform-related properties have been proposed throughout the literature. The underlying characterization to gather appropriate runtime characteristics is obtained either from a dynamic profiling phase [1, 2, 6, 7, 14, 15, 17, 22], or based on a static code analysis [3, 4, 12, 13, 21] (as done in our work), to extract the required dynamic behaviour of the application.

In the next step, an abstracted performance model is created. The main differences between the resulting models manifest in the abstracted details of the target (micro) architecture and the resulting granularity. For the following performance estimation itself, either target workload clones are generated [1, 2, 7, 14, 22], statistical [6, 12, 17] or trace-based [1, 6, 18] models are simulated, or an enriched functional model is run in a host-based, native simulation. Some approaches support multiple estimation methods.

We propose a hybrid approach, supporting both the generation of abstracted control-flow aware workload models, based on the substitution of replaceable basic blocks as well as functional representations, enriched with workload estimates for the target. The presented framework supports the creation of target binary clones as well as natively executed models, integrated into a SystemC/TLM virtual prototype.

In contrast to the purely non-functional workload simulations, several automatic annotation techniques for timed performance models based on native execution of the functional input model have been proposed [3–5, 8, 13, 15, 19, 21]. Most of these approaches do not include (implicit) memory access pattern and focus on a pure, local delay annotation, without generating observable traffic on the SoC's interconnect model. The resulting simulation performance and the timing accuracy of these approaches is comparable to our model, especially when using the host-based model without explicit memory transactions (Sec. 4).

The approach in [5], which is based on a pure source-level estimation, has been extended in [9] with a cache model based on the resulting host address distribution during native simulation, to obtain memory accesses on the platform. The cache model used in [13] is not utilized for traffic generation and only adjusts the annotated delays based on miss rates. The characterization of the target binary code proposed by Stattelmann et al. [21] especially supports optimized code through a path simulation of the delay model in parallel to the native simulation of the functional code. External memory accesses however are not directly supported.

The underlying annotation technology has been realized based on the LLVM infrastructure [16] in [3, 4, 13]. Similar to our approach, all of these works start on the IR basic block level and provide an integrated characterization step followed by native compilation with back-annotation support. In [4], the focus is on energy characterization at the LLVM IR level and external memory is not explicitly considered.

The approach presented in [3] also focuses on local delay annotations, but has been extended in [8] to include more detailed memory models and in [20] to support additional target memory maps by using virtualization techniques on the simulation host. These approaches are closest to the work presented in this paper. Still, the memory characterization in [8] is performed heuristically at the IR-level, while we analyse the final target assembly.

An early work proposing a similar hybrid approach for instrumenting a native simulation with profiling information from the target architecture (including memory accesses) has been proposed in [15]. Compared to our approach, which is based on a characterization of the final target assembly, the analysis in [15] is performed on a three-address-IR and annotations are added to each step in this representation. This imposes an additional simulation cost and cannot represent implicit operations like function call preambles.

3. WORKLOAD EXTRACTION

An overview of the proposed workload extraction approach is presented in Fig. 2. It consists of two cycles through which the input application will be passed.

We have utilized the LLVM compiler infrastructure [16] to implement the different steps of the proposed approach. A frontend will compile the input high-level code down to the strongly typed, target-independent LLVM *intermediate representation (IR)* language, which serves as the input format for the LLVM compiler and its backend. LLVM allows for a high-level view of the code, including function calls and static type information, while at the same time providing a relatively simple three-address code, closely related to a target implementation. A popular frontend to LLVM targeting C-based languages is Clang. The proposed approach however solely operates on the IR, such that other compiler frontends and input languages are supported as well.

Within LLVM, the code generation step, ranging from IR basic blocks containing the application's behaviour down to target-specific machine code, can be easily observed and extended using the concept of compilation *passes*. The *IR basic block analysis* and the *IR basic block instrumentation*

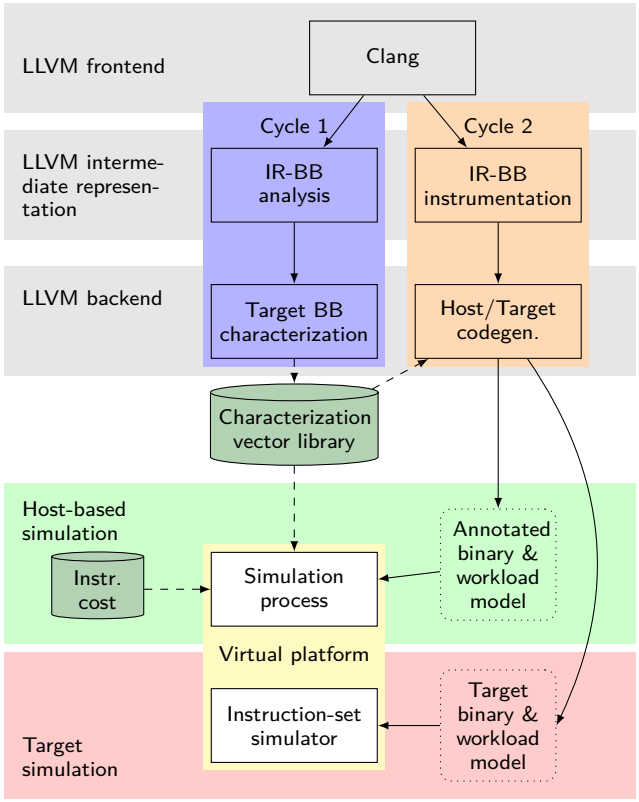

Figure 2: Overview of the proposed workload characterisation approach.

passes both operate on this level of the LLVM compile flow and are able to iterate and modify the IR of the input application. The LLVM backend then controls the code generation, a process of mapping the IR to valid target-specific instructions. In the proposed approach, the *target basic block characterisation* pass iterates over the resulting target-specific basic blocks and performs the characterization. In this stage of the compile flow, it is also possible to define new rules for the code generation process.

In the first cycle, the application is analysed on the IR basic block level and a library of the basic block's platform behaviour in terms of resource usage is created. We introduce the notion of a *characterization vector*, which describes the local CPU usage as well as memory access patterns of a target basic block. Characterization vectors are used either in combination with a look-up table in a host-based simulation scenario, coupled with the target instruction costs, or to create an *abstracted workload model*, that is, a target binary where control flow-irrelevant parts of the application are replaced by *abstracted workload representations* generated from their corresponding characterization vectors.

3.1 IR Basic Block Analysis & Target BB Characterization

In the first step, the input program is statically analysed in order to identify basic blocks which do not change the control flow of the program. In LLVM, a basic block contains a continuous instruction sequence and control flow branches are modelled as edges between these blocks. The analysis is performed during the compilation step on the LLVM IR. Moreover, in this step, the IR code has been transformed to static single assignment (SSA) form which allows for an in-depth analysis of the inter-dependencies between the IR basic blocks.

A basic block is considered *replaceable*, iff it does not have any side effects on the control flow of the application. This means that a replaceable basic block does not modify any variables which are later used (directly or indirectly) outside of the block itself in conditional instructions. Another requirement for considering a basic block replaceable is that it does not contain more than one successor. This is required to preserve the control flow logic of the input program. By only marking basic blocks as replaceable which meet the aforementioned requirements, any statements influencing the control-flow of the program, such as loop variables and conditions are preserved.

In the next step, the corresponding target code for *all* IR basic blocks is characterized. During the compilation step in the LLVM backend, an IR basic block is refined to target-specific basic blocks containing the target instructions. These *machine basic blocks (MBB)* are now analysed and a characterization vector for each of these MBBs along with a reference to the originating IR basic block is written to a characterization vector library.

Since we focus on memory access patterns and local CPU usage behaviour, we abstract the behaviour of an MBB to a characterization vector consisting of CPU and memory usage *nodes*. Each instruction of the MBB is analysed according to it's induced local delay and any possibility of emitting a memory transaction from the processor. To capture the local delay, the instruction's cost factor is obtained, e.g. using a static weight for each instruction type. Also, the memory access type of the instruction (none, read, write) is stored in the characterization vector. Thus, an MBB's behaviour is *represented* by its characterization vector, defined as follows:

DEFINITION 1. *A* characterization vector $V = \langle v_1, \ldots, v_n \rangle$ *is a finite sequence of resource usage nodes* $v_i = (\Delta_i, m_i)$ *consisting of a target-specific instruction cost* $\Delta_i \in \mathbb{N}$ *and a memory access attribute* $m_i \in \{N, R, W\}$ *modelling the local delay and the memory access(es) for each instruction (none, read, write).*

To construct a characterization vector from an MBB, the *target basic block characterization* step (see Fig. 2) iterates over all instructions of an MBB and adds an element to the characterization vector according to the current instruction's cost, derived from a target-specific cost function, and its memory access attributes, which can be derived from the instruction type.

3.2 IR Basic Block Instrumentation & Code Generation

After all IR basic blocks are marked and the characterization vectors of their resulting MBBs have been generated, the compilation process is restarted for the next cycle (cycle 2 in Fig. 2). Now, all IR basic blocks are instrumented by the *IR instrumentation* pass with an additional function call. This call serves as marker for the *host / target code generation* step. Due to the separation of the characterization process and the creation of the instrumentation points in the application, generating different workload models originating from the same application code is now possible. For example, ignoring the inserted function calls in the target

```
int arr[5];

void main()
{
  arr[0] = arr[1]
         + arr[2];

  arr[3] = arr[4]
         + arr[5];
}
```

(a) Input source

```
%0 = load i32* ...

%1 = load i32* ...

%add = add nsw i32 %0, %1
store i32 %add, i32* ...
%2 = load i32* ...
%3 = load i32* ...
%add1 = add nsw i32 %2, %3
store i32 %add1, i32* ...
call void @__bb_mark(i32 1)
ret void
```

(b) Instrumented IR basic block

```
(1, R)
(1, R)
(1, R)
(1, R)
(1, N)
(1, W)
(1, R)
(1, R)
(1, N)
(1, W)

(1, N)
```

(c) V_1

```
ldr r0, [pc, #36]
ldr r0, [r0]
ldr r1, [r0, #4]
ldr r2, [r0, #8]
add r1, r1, r2
str r1, [r0]
ldr r1, [r0, #16]
ldr r2, [r0, #20]
add r1, r1, r2
str r1, [r0, #12]

bx  lr
```

(d) Target binary

```
mov    $0x1,%eax ; BB id 1
mov    0x4,%ecx
add    0x8,%ecx
mov    %ecx,0x0
mov    0x10,%ecx
add    0x14,%ecx
mov    %ecx,0xc
movl   $0x0,(%esp)
mov    %eax,-0x4(%ebp)

call   3a ; call __bb_main(1)
add    $0x8,%esp
pop    %ebp
ret
```

(e) Host binary with annotation

Figure 3: Characterization process, resulting in the characterization vector V_1

code generation pass leads to the output of a (target) binary which can be executed on an ISS-based virtual platform.

The code generation step is also able to emit an annotated binary suitable for a host-based simulation by using the native code generator and leaving the function calls to the annotation backend. As can be seen in Fig. 2, such an annotated binary along with the database containing the characterization vectors can be used in a host-based simulation. An example of a simple application code and the instrumented LLVM IR, along with the resulting target and annotated host binaries is given in Fig. 3. The picture shows the application code in each step of the proposed approach. First, the input source code is compiled down to target instructions (Fig. 3d). The characterization process now iterates over the generated target code and creates a characterization vector (Fig. 3c) reflecting the temporal behaviour as well as the memory access patterns in the target MPSoC address space. As can be seen in the figure, a simple mapping which characterizes each instruction cost factor equally by the value 1 was used, along with the instruction's memory access properties. The vector is then annotated at the IRBB in Fig. 3b using the call to `__bb_mark` at the end of the block. This IR can be compiled to a host binary, as depicted in Fig. 3e, therefore enabling the simulation of the captured target behaviour in a host-based simulation.

Moreover, replaceable basic blocks can be omitted in the code emission step, which, depending on the size of the removed basic blocks, may speed up the simulation. In the example shown in Fig. 3, the host binary does not have to execute the contents of the replaceable basic block content to correctly model the platform workload of the target binary.

3.3 Testing the Characterization Vector

The generated characterization vectors are an abstract view on the platform-observable behaviour of the application. To test whether this reflects its behaviour sufficiently, it is possible to emit MBBs during the target code generation step according to the information given in the corresponding characterization vector. However, only *replaceable* basic blocks can be substituted, since we can assume that they don't influence the original application's control flow. This way, a modified target binary is constructed and its behaviour regarding memory and local CPU usage can be compared against the unmodified target binary. A closely related behaviour of both simulation runs in terms of simulated time and generated transactions is an indicator of an accurate characterization process. We use this technique to validate our characterization vector generation step.

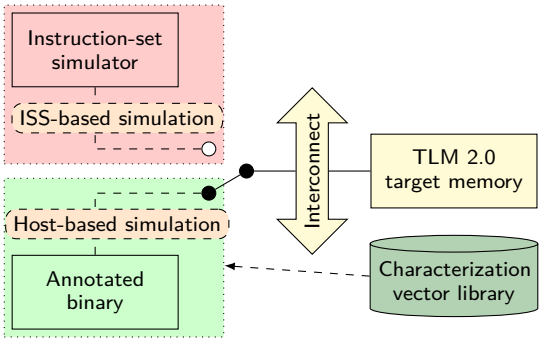

Figure 4: Virtual platform configuration used in the evaluation. The switch indicates the possibility of replacing the instruction-set simulator with a wrapper for the host-based annotated binary.

4. EVALUATION

To evaluate the approach, we have created a simple virtual platform consisting of an `ARM926EJ-S` instruction-set simulator (ISS) without a cache model and a SystemC/TLM 2.0 target memory connected to a simple untimed TLM 2.0 bus. The ISS can be replaced by a wrapper for the host-based, annotated binary, as depicted in Fig. 4. This way, it is possible to test both the target and the host-based binary versions within the same virtual platform. The results of a simulation run contain the total execution time for the model, the simulated time and the memory transaction count that occurred on the processor's data port or were generated by the host-based binary, respectively. Since we are only interested in the resource usage patterns of the original application code, we are not tracing the instruction port of the ISS.

The workload extraction framework was used to generate two output versions from the same application. The first version contained the original application code and can be run on an ISS, if cross-compiled to a target binary, or as an annotated binary in a host-based simulation. The resulting binaries are called *ISS* and *host-based*, respectively. For the second version, we have generated an abstracted workload

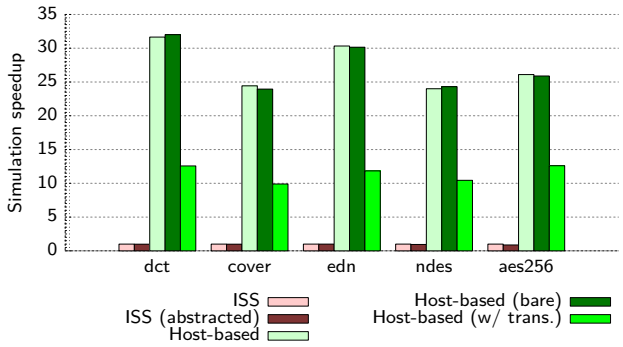

Figure 5: Simulation speedup.

model of the input application for the ISS-based simulation by substituting all *replaceable* basic blocks with corresponding abstracted workload representations, as described in Sec. 3.3. These results are named *ISS (abstracted)*. In case of a host-based simulation approach, we keep the replaceable block's annotations but remove their generated host code, resulting in a control flow-aware, host-based simulation as illustrated in Fig. 3e, denoted as *host-based (bare)*.

In order to test and calibrate the consumption of the annotated instruction cost factor given in the characterization vectors, we have compared the platform-observable behaviour of an abstracted workload model generated from a discrete cosine transformation (DCT) binary to the original one running on the ISS. We have chosen the `dct` example due to the algorithm's small amount of control flow dependencies, allowing us to replace more than 90% of the target binary instructions.

Furthermore, we have tested the approach on examples of a WCET benchmark suite [11] using the `edn` example consisting of vector multiplications, the `ndes` benchmark executing a DES encryption, and the `cover` example containing jump tables. Additionally, a `dct` as well as an `aes256` implementation were used to test the approach.

A comparison of the simulation runtime is given in Fig. 5. With the host-based approach, a speedup of 12x can be achieved compared to the ISS simulation, which includes the bus model simulation using the annotated memory transactions, including a full transaction trace. When omitting the bus model in the virtual platform, e.g. in scenarios where the interference of multiple initiators is not of interest, it is possible to achieve a speedup of 30x. By exploiting the knowledge of the system's memory map, a combined model can improve the simulation speed by only emitting transactions that are routed through shared resources, while replacing local memory transactions by a pure delay annotation.

The simulation runtime for bare host-based workload models (replaceable basic blocks removed) as well as for abstracted workload models (replaceable basic blocks substituted) were similar to the corresponding versions containing the full functional behaviour. This shows that the ISS was not able to gain a simulation speedup when executing the abstracted workload representation basic blocks inserted in the target binary. In case of a host-based simulation, only a limited speedup of 3% was observable, likely due to the small size of the examples and the instruction cache behaviour of the simulation host, which mitigated the effect of a reduced

Table 1: Comparison of the workload model behaviour compared to the original binary running on the ISS.

	Simulated time		Generated transactions	
Example	Abs. (ns)	Dev. (%)	Abs.	Dev. (%)
`dct`	231 930.050	<0.01	11 587	−2.74
`edn`	1 450 110.050	−3.21	84 949	−1.93
`cover`	51 000.050	−0.02	3096	−0.23
`ndes`	1 110 198.100	−0.02	68 623	−5.76
`aes256`	581 730.050	<0.01	24 165	−0.92

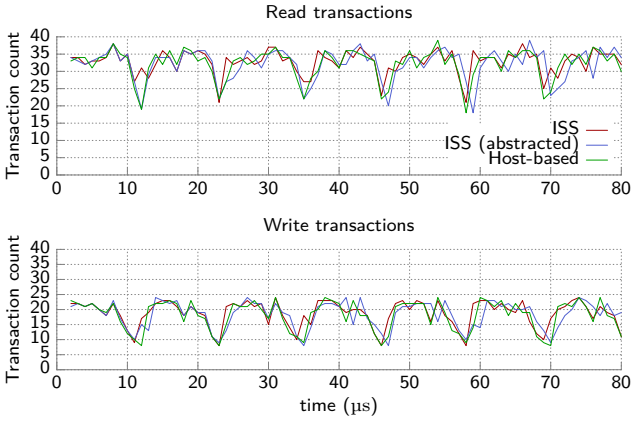

Figure 6: Excerpt of the DCT example's transaction trace.

instruction execution count.

The generated memory transactions of the host-based approach compared to the original binary running on an ISS is depicted in Tab. 1. Except for the `edn` example, whose simulated time is 3.21% shorter than the original binary, the difference is negligible. Regarding the generated transactions, we can see that the `ndes` example has the highest deviation, generating 5.76% fewer transactions than the original binary. The second-highest deviation of −2.74% was generated by the `dct` example. The other examples were able to produce a very similar transaction count with a deviation of less than 2%. The similarity between the original and the generated memory traces for read and write operations can be seen in the excerpt of the recorded memory transactions of the `dct` example in Fig. 6. The bare host-based model was omitted in the graph, since it contains the same annotations as its unmodified counterpart and therefore produces identical transaction traces.

Fig. 6 also contains a trace of the abstracted binary running on the ISS. As can be seen in the excerpt, by substituting replaceable basic blocks from the binary with their workload representations as explained in Sec. 3.3, it is possible to generate a trace similar to the original binary.

5. CONCLUSION & OUTLOOK

In this work, we have presented our approach towards an automatic workload extraction framework for embedded software performance estimation. By using an LLVM-based characterisation and compilation toolchain, the source code is characterized according to its target performance characteristics at basic block level, both in terms of local

execution time and shared resource access patterns. The workload model is calibrated by using a control-flow analysis and incremental substitution of *replaceable* basic blocks with abstracted workload representations, both running on an instruction-accurate ISS as reference model.

As a result, a control-flow aware workload model is obtained, which can be compiled as a final target binary, as well as a natively execution simulation model for fast simulation. In the evaluation, we have demonstrated the feasibility of our approach based on common WCET benchmarks, both in terms of accuracy (in terms of execution time and memory accesses) and simulation speed, especially for the host-based execution.

In the future, we intend to refine our control flow analysis to enable further omission of functional code without losing timing/memory accuracy, as well as the introduction of an inter-basic block cost, to model pipeline stalls, etc. Another important improvement plan is to address compiler optimizations, based on a similar approach as presented in [21].

Last, but not least, we intend to integrate energy cost to the characterization vector, to enable the integration into our ESL simulation framework for heterogenous MPSoCs [10].

Acknowledgements

This work has been partially supported by the *ARAMiS* project (grant agreement 01IS11035M) and the *EMC*^{2} collaborative project (ARTEMIS, grant agreement 01IS14002R), both funded by the German BMBF.

References

[1] A. Awad and Y. Solihin. STM: Cloning the spatial and temporal memory access behavior. In *2014 IEEE 20th Intl. Symposium on High Performance Computer Architecture (HPCA)*, pages 237–247, 2014. doi: 10.1109/HPCA.2014.6835935.

[2] R. H. Bell, Jr. and L. K. John. Improved automatic testcase synthesis for performance model validation. In *Proc. of the 19th Annual Intl. Conf. on Supercomputing*, ICS '05, pages 111–120. ACM, 2005. ISBN 1-59593-167-8. doi: 10.1145/1088149.1088164.

[3] A. Bouchhima, P. Gerin, and F. Pétrot. Automatic instrumentation of embedded software for high level hardware/software co-simulation. In *2009 Asia and South Pacific Design Automation Conference*, pages 546–551. IEEE, Jan. 2009. ISBN 978-1-4244-2748-2.

[4] C. Brandolese, S. Corbetta, and W. Fornaciari. Software energy estimation based on statistical characterization of intermediate compilation code. *Proc. of the Intl. Symposium on Low Power Electronics and Design*, pages 333–338, 2011. ISSN 15334678.

[5] A. Díaz, H. Posadas, and E. Villar. Obtaining Memory Address Traces from Native Co-Simulation for Data Cache Modeling in SystemC. *XXV Conf. on Design of Circuits and Integrated Systems (DCIS'10)*, 2010.

[6] L. Eeckhout, J. Bell, R.H., B. Stougie, K. De Bosschere, and L. John. Control flow modeling in statistical simulation for accurate and efficient processor design studies. In *31st Annual Intl. Symposium on Computer Architecture, 2004. Proceedings*, pages 350–361, June 2004. doi: 10.1109/ISCA.2004.1310787.

[7] K. Ganesan and L. K. John. Automatic generation of miniaturized synthetic proxies for target applications to efficiently design multicore processors. *IEEE Transactions on Computers*, 63(4):833–846, Apr. 2014. ISSN 0018-9340. doi: 10.1109/TC.2013.36.

[8] P. Gerin, M. M. Hamayun, and F. Pétrot. Native MPSoC co-simulation environment for software performance estimation. *Proc. of the 7th IEEE/ACM intl. conf. on Hardware/-software codesign and system synthesis (CODES+ISSS'09)*, pages 403–412, 2009.

[9] P. González, P. P. Sánchez, and A. Díaz. Embedded software execution time estimation at different abstraction levels. *XXV Conf. on Design of Circuits and Integrated Systems (DCIS'10)*, 2010.

[10] K. Grüttner, P. A. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Stattelmann, B. Sander, O. Bringmann, W. Nebel, and W. Rosenstiel. An ESL Timing & Power Estimation and Simulation Framework for Heterogeneous SoCs. In *Intl. Conf. on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, pages 181–190, 2014.

[11] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *10th Intl. Workshop on Worst-Case Execution Time Analysis (WCET'10)*, volume 15 of *OASIcs*, pages 137–147, 2010.

[12] C. Hughes and T. Li. Accelerating multi-core processor design space evaluation using automatic multi-threaded workload synthesis. In *IEEE Intl. Symposium on Workload Characterization (IISWC'08)*, pages 163–172. IEEE, 2008. ISBN 978-1-4244-2777-2. doi: 10.1109/IISWC.2008.4636101.

[13] Y. Hwang, S. Abdi, and D. Gajski. Cycle-approximate retargetable performance estimation at the transaction level. *Proc. of the Design, Automation & Test in Europe Conf. (DATE'08)*, pages 3–8, 2008.

[14] A. Joshi, L. Eeckhout, R. Bell, and L. John. Performance cloning: A technique for disseminating proprietary applications as benchmarks. In *IEEE Intl. Symposium on Workload Characterization*, pages 105–115. IEEE, Oct. 2006. ISBN 1-4244-0509-2, 1-4244-0508-4. doi: 10.1109/I-ISWC.2006.302734.

[15] T. Kempf, K. Karuri, S. Wallentowitz, G. Ascheid, R. Leupers, and H. Meyr. A SW performance estimation framework for early system-level-design using fine-grained instrumentation. *Proc. of the Design Automation & Test in Europe Conf. (DATE'06)*, 1:468–473, 2006.

[16] LLVM. The LLVM Compiler Infrastructure. URL `http://llvm.org`.

[17] S. Nussbaum and J. E. Smith. Modeling superscalar processors via statistical simulation. In *Proc. of the 2001 Intl. Conf. on Parallel Architectures and Compilation Techniques*, PACT '01, pages 15–24. IEEE Computer Society, 2001. ISBN 0-7695-1363-8.

[18] R. Plyaskin and A. Herkersdorf. A method for accurate high-level performance evaluation of MPSoC architectures using fine-grained generated traces. In *Architecture of Computing Systems - ARCS 2010*, volume LNCS 5974, chapter 18, pages 199–210. Springer, Berlin, Heidelberg, 2010. ISBN 978-3-642-11949-1.

[19] H. Posadas, A. Díaz, and E. Villar. SW Annotation Techniques and RTOS Modelling for Native Simulation of Heterogeneous Embedded Systems. In K. Tanaka, editor, *Embedded Systems - Theory and Design Methodology*, chapter 13. InTech, Mar. 2012. ISBN 978-953-51-0167-3.

[20] H. Shen, M.-M. Hamayun, and F. Pétrot. Native Simulation of MPSoC Using Hardware-Assisted Virtualization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 31(7):1074–1087, July 2012. ISSN 0278-0070.

[21] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *48th ACM/EDAC/IEEE Design Automation Conf. (DAC)*, pages 486–491. IEEE, 2011. ISBN 9781450306362.

[22] L. Van Ertvelde and L. Eeckhout. Benchmark synthesis for architecture and compiler exploration. In *2010 IEEE Intl. Symposium on Workload Characterization (IISWC)*, pages 1–11, Dec. 2010. doi: 10.1109/IISWC.2010.5650208.