SysCon 2008 - IEEE International Systems Conference
Montreal, Canada, April 7–10, 2008

A Methodology for Performance Predictions of Future ARM Systems Modelled in UML

Lukas Pustina[1], Simon Schwarzer[1], Prof. Dr. Peter Martini[1], Jari Muurinen[2], Ari Salomaki[2]

[1]Institute of Computer Science IV
University of Bonn
D-53117 Bonn Germany
{pustina, schwarzer, martini}@cs.uni-bonn.de
Telephone: +49 228 73-4118

[2]Nokia-TP
Salo/Tampere, Finland

Abstract – *The increasing complexity and short product cycles drive developers of mobile systems to analyse the performance of systems before hardware prototypes are available. Therefore, it is necessary to predict application runtimes with the help of simulations of system models. Miscellaneous components and factors of mobile devices affect the performance, e.g. caches, buses etc. In order to predict the performance of new system designs already during early stages of development, models of the timing behaviour are necessary. We have developed a modular timing simulator for models of typical mobile systems which can be used to predict the runtime of applications on future systems. Since UML is the de-facto standard for software modelling and widely used, we use UML to specify the hardware of the system. In this way, the gap between hardware and software modelling may be closed and performance analysis of application and system design are tight closer. The UML system model consists of an architecture model and an instruction behaviour description. The architecture model describes the components of the system and the connections between them and the behavioural model specifies the timing of the processor instructions. These models are used to simulate different configurations of an ARM9 system. Traces from one configuration are used to predict the performance of another configuration. Predictions for an ARM11 system with parallel pipeline units are made.*

Keywords – *Performance, UML, ARM systems, Simulation, Modelling*

I. INTRODUCTION

Especially, in the area of mobile systems there exists the need for predicting the performance of systems without building hardware prototypes. This is due to the short product cycles and the rapid growth in this area with additional applications of the devices, e.g. multimedia cellphones. Since ARM processors are widely deployed in mobile systems, this paper and the proposed tools focus on the ARM processor family.

In this paper, we present a UML based modelling methodology to specify system, especially processor, details. The developer may connect several caches, writebuffers, memories, and buses to model the desired configuration. Moreover, the architecture of the processor pipeline consisting of pipeline stages and the timing behaviour of processor instruction in the pipeline are specified. UML diagrams are used to model the system architecture and timing behaviour. UML is a graphical modelling language and is the de-facto standard in the software development area. The presented hardware modelling with UML requires specialised concepts which are supported by the UML profile SysML [1]. The MARTE profile [2] is used to enhance the elements of the model with additional semantics.

From this UML descriptions simulator configurations are derived. Therefor we have developed a modular timing simulator for assembler instruction traces. For each component of a mobile system there is a representing simulation module. This modular approach allows for a flexible configuration of a large range of systems. Instruction traces of existing applications captured on existing hardware are used as input for the simulator to predict the runtime of the corresponding applications on the modelled systems. Due to this trace-based approach, there is no functional simulation necessary and the simulator only focuses on the mere timing of instructions and components. The simulator is designed to be as independent as possible from the instruction set.

The methodology of using captured traces as input for future system models is evaluated by using captured traces to fed simulations of modelled systems and compare the predictions with measured figures. Since caches play a significant role for the performance, ARM9 systems with and without caches are modelled in UML, simulated, and the results are compared with real world figures. ARM11 systems that include parallel pipeline stages, are also modelled and predictions based on ARM9 traces are presented.

The rest of this paper is structured as follows. Section II gives an overview of related work in the context of processor simulation and UML modelling. Section III presents details of the ARM processor family which is used as case study throughout this paper. The modelling methodology is introduced in section IV by means of example diagrams for the ARM9 and the ARM11 processors. Section V presents an

evaluation of the methodology and the timing simulator. Simulation predictions of system models are compared with real-world measurements. Section VI concludes the paper.

II. RELATED WORK

Due to the complexity of processors and microarchitectures, simulations are used to predict their performance [3]. In the context of processor simulation two approaches exist, i.e. trace-driven simulation and execution-driven simulation. Trace-driven simulation uses captured or synthetically generated trace files as input and simulates their timing behaviour on a modelled system. This approach is an old technique and widely used [3], [4]. Execution-driven simulation uses software programs as input and simulates their functional execution. SimpleScalar [5], [6] is an example for this approach. The execution-driven approach suffers from the drawback of a fix instruction set and the necessity to port operating systems and drivers to the simulation framework. Programs like Qemu [7] emulate the functionality of a processor, but lack a timing model for the processor and the architecture. The advantage of the trace-driven approach is, that every traceable program can be used as input without the implementation of special system calls or the need to adapt the operating system. The drawback is that, branch prediction in the pipeline can not always be modelled, because often the input traces contain only the executed instructions. However, this has typically no significant influence on the simulation accuracy [3]. The ChARM tool [8] for ARM-based systems follows the trace-driven approach, but the simulated processors are not up to date anymore and the simulated instruction set is not configurable.

We also follow the trace-driven approach and developed a modular timing simulator for ARM-based systems. The simulator supports user-defined architectures and instruction sets. Up to date processors of the ARM family are modelled and used to simulate traces gathered at the hardware level. Thus, effects of the operating system and drivers are automatically included which is important for accurate system simulations [9], [10]. Execution-driven approaches and emulators may be used as alternative to real hardware to generate input traces for the timing simulator.

The system and the processor details are modelled with UML. Software performance engineering methods (SPE) [11] use annotated UML diagrams to model the system and software under study [12], [13], [14]. Since UML does not allow for the modelling of non-functional aspects many authors apply the *UML Profile for Schedulability, Performance, and Time Specification* (SPT) [15] to enhance the diagrams with the necessary semantics [13], [14]. The *UML Profile for Modeling and Analysis of Real-Time and Embedded systems* (MARTE) [2] is the successor of the SPT profile, allows for a detailed modelling of performance aspects, and supports UML 2. Composite structure diagrams of UML 2 are used in SPE methods to model system architectures without processor details [13], [16]. Our modelling methodology uses new diagram types of the SysML profile [1]. SysML is a subset of UML 2 with some extensions to allow for a detailed system modelling.

The authors of [17] analysed the performance of a video codec on ARM systems and determined components which affect the performance. The proposed modelling methodolgy and the timing simulator consider these components.

Using a subset of benchmark suites is often applied to analyse system architectures [18], [19]. Therefore, applications of the MiBench suite [20] with inputs from MiDataSets [21] are used in the evaluation.

III. ARM PROCESSOR FUNDAMENTALS

Figure 1 depicts a schematic overview of components typically used in mobile systems like cellphones. A processor consisting of a pipeline and registers is connected to caches. These caches are connected via a system bus to the main memory and other peripherals. A writebuffer is placed in between the data cache and the bus, so that the processor is not delayed by write accesses to the memory.

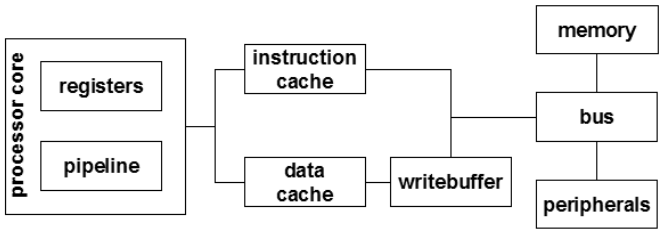


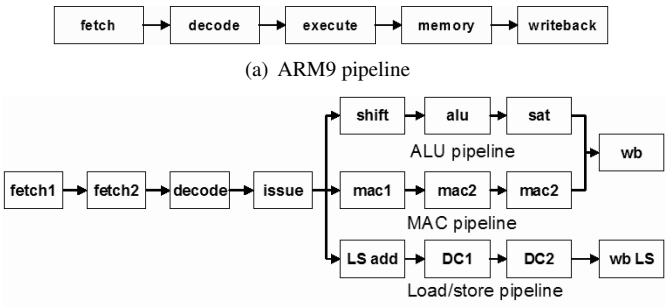
Fig. 1. Schematic overview of typical components in a mobile phone.

The main memory is often built from one or more RAM modules. These modules have different latencies for read and write access and support a special *burst* mode when successive data is addressed. Since the RAM modules have to internally address the requested data first, the read and write access latencies depend on the distance between the requested data addresses and the internal data cell structure.

ARM processors are widely used in mobile devices, thus this paper focuses on this processor family. The ARM processors are *Reduced Instruction Set Computer* (RISC) based and employ modern concepts like pipelines and Harvard separated instruction and data caches. The advantage of the pipeline concept is that all pipeline stages work in parallel and thus multiple instructions can be processed during one pipeline cycle. There are two types of pipeline cycles. Arithmetical and logical calculation steps, reading registers etc. last one *internal* cycle (I-cycle) which duration depends on the clock speed of the processor. Requesting instructions or data from the memory depends on the latencies of the bus and the memory modules. This duration is referred to as *memory* cycle (M-cycle) in the following. The theoretical maximum parallelism of the pipeline stages cannot always be achieved due to *interlocks* and *stalls*. An instruction in the pipeline may need the result of a predecessor instruction for its own calculation. Such a situation is called an *interlock* and the pipeline is automatically

stalled until the required values become available. Another cause of pipeline *stalls* are instructions needing more than one I-cycle in a stage, e.g. complex multiplications.

The ARM9 [22] and the ARM11 [23] processors are used as examples throughout this paper. Figure 2(a) depicts the fiveary pipeline of the ARM9. The *fetch* stage reads instructions from the memory. The *decode* stage decodes instructions and the *execute* stage performs arithmetic, logical, and multiply operations.The *memory* stage performas data access to the memory and in the *writeback* stage results are written back to registers. The pipeline of the ARM11 introduces the concept of parallel pipeline stages. Figure 2(b) depicts a schematic overview of the ARM11 pipeline which splits up after the *issue* stage into the three pipelines *ALU pipeline*, *MAC pipeline*, and *load/store pipeline*. The *issue* stage reads registers and issues the instructions to the succeeding pipelines which may process instructions in parallel e.g. a logical operation in the ALU and a multiplication in the MAC can be processed in parallel.



(a) ARM9 pipeline

(b) ARM11 pipeline

Fig. 2. Schematic overview of ARM processor pipelines.

IV. MODELLING METHODOLOGY

This section presents the UML modelling methodology for mobile systems that applies the SysML profile for UML and the MARTE profile. The SysML profile enhances the semantics of UML to allow for the specification of systems consisting of hardware and software. The MARTE profile is used to enhance the model elements with the additional needed semantics of the components.

The *architecture model* describes the system components which affect the performance, e.g. caches, buses, the instruction set etc. It defines the architecture of the system consisting of these components and the communication paths between them. Basic block diagrams (BBD) of SysML are used to define the components of the system model. Basic block diagrams are the SysML counterpart of UML class diagrams. Internal block diagrams (IBD) of SysML are used to model the internals of the components and their interconnections. They are similar to composite structure diagrams in UML.

The pipeline behaviour and the timing behaviour of the processor instructions are modelled in the *instructions behaviour model*. Sequence diagrams specify behavioural aspects of the processor instructions.

A. Architecture Model

System components are defined as SysML blocks in a block definition diagram. SysML blocks are modular units of the system which can be logical or physical. The block definition diagram defines the static relationships between the blocks i.e. compositions and generalisations. A Block is a stereotype of the UML class element. The architecture and the communication between the components are specified in internal block diagrams. This diagram allows for refinements of components and the modelling of runtime communication between the modelled elements. Instances of the defined blocks are used in internal block diagrams as so-called *parts*.

Blocks and parts do not carry any semantical meaning in respect to mobile system components. Thus, stereotypes are used to enhance the model with semantics. The MARTE profile is applied, which introduces multiple stereotypes to annotate instances in a sophisticated manner, i.e. ≪`HwProcessor`≫, ≪`HwCache`≫, ≪`HwMemory`≫, and ≪`HwBus`≫. Nevertheless, the granularity of this profile is not detailed enough for our system modelling approach. Thus, we extend stereotypes by additional tags and introduce new stereotypes. The stereotype ≪`HwProcessor`≫ provides tags for the specification of the number of pipeline stages and the number of registers. Since only the amount of stages does not suffice to simulate an assembler trace with our timing simulator in a realistic way, we introduced the new stereotypes ≪`HwPipelineStage`≫ and ≪`HwRegisterbank`≫.

Figure 3 shows a block definition diagram with the involed components modelled as blocks. The *Registerbank* and *Pipelinestage* blocks are parts of the *Processor* block modelled by composition. The *Cache* block is refined by an instruction cache *ICache* block and a data cache *DCache* block. Instances of these blocks are used as parts in internal block diagrams. Figure 4 depicts an internal block diagram describing the architecture of an ARM9 system with the aforementioned components (cf. fig. 1 and **??**). The instruction cache is directly connected to the bus, whereas the data cache is connected via a writebuffer to the bus. The bus is connected to the main memory. All parts in this internal block diagram are annotated with stereotypes. The memory part is annotated with ≪`HwMemory`≫ and latencies for different data accesses depending on the distance between the requested memory addresses (cf. section III) are specified in the annotation. The stereotype ≪`HwBus`≫ describes the properties of the bus part. The tags `bandwidth`, `clock`, and `schedPolicy` are used as defined in the profile and specify the bandwidth, the clock speed, and the arbitration scheme of the bus. In order to specify the burst capabilities of a bus, we extended this stereotype by the tag `burst` which gives the supported burst lengths. The ≪`HwWritebuffer`≫ stereotype provides tags to specify the properties of the buffer. The tag `addressBuffer` specifies how many non-successive write requests can be buffered.

The tag `buffersize` specifies the maximum number of data which can be buffered for the requests. The ≪HwCache≫ stereotype provides tags for the specification of cache properties like size, latency, replacement policy etc. The ≪HwProcessor≫ stereotype specifies the clock speed of the processor and gives the annotated component a processor role.

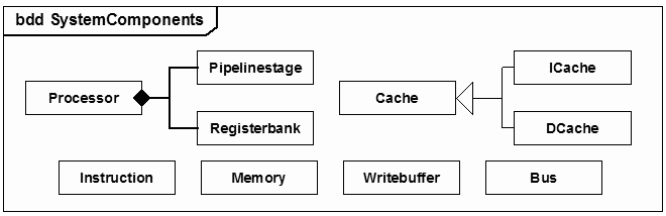


Fig. 3. Block definition diagram of the components used in the ARM9 model.

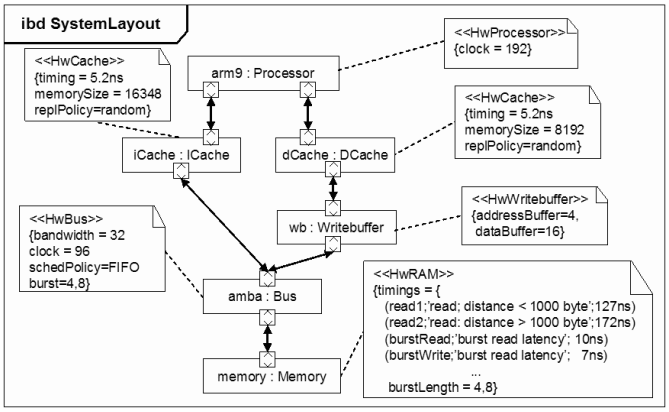


Fig. 4. Internal block specifying an ARM9 system.

In order to describe the internal properties of the processor part, this part is refined by another internal block diagram. The internal components are pipeline stages and the register bank. The ≪HwRegisterbank≫ stereotype contains all necessary information to specify the register bank properties, i.e. the number of registers and the register size. The refined processor block consists of pipeline stages which are connected with each other to specify possible instruction paths. Pipeline stage parts are annotated with the ≪HwPipelinestage≫ stereotype that may contain the tags `defaultCycle` and `branchExecuteStage`. The `defaultCycle` tag is used to ease the specification of the timing behaviour of instructions that will be described in section B. The tag `branchExecuteStage` is set for stages which determine whether a branch is taken or not. If a branch was miss-predicted, the pipeline is flushed to remove already fetched instructions from the predecessor stages. Pipeline stages which access the caches or the memory need appropriate ports which connect them to the cache or memory ports of the processor block (cf. fig. 4). Stages which processing require register values, need a connection to the register bank.

Figure 5 depicts the five-ary pipeline of an ARM9 processor (cf. fig. 2(a)). For simplicity, only the annotations of the *fetch* and *execute* stages are shown here. The *fetch* and *memory* stages have memory ports, because the fetch stage loads instructions from the memory and the *memory* stage reads or writes data.

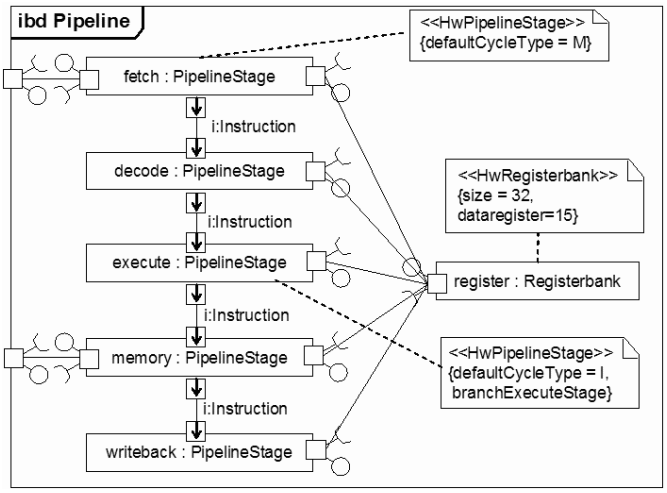


Fig. 5. Internal block diagram of an ARM9 pipeline.

In case of a linear pipeline, all instructions take the same path through the pipeline. Consecutive pipeline stages are connected with each other via *atomic flow ports* (cf. fig. 5), a special variation of UML ports which accept only one type and have only one flow direction. The atomic flow port is typified with the *Instruction* block (cf. fig. 3). In this way the path of instruction objects through the pipeline is defined.

In case of a pipeline with parallel pipeline stages, e.g. ARM11 pipeline, instructions may take different paths (cf. fig. 2(b)). For example, an *add* (addition) is processed by the ALU stage whereas a *mul* (multiplication) is processed by the MAC stage, but both stages have the same predecessing stage. Therefore, it is necessary to specify the paths depending on the instruction types. Specialised instruction blocks are introduced which are employed to typify the *atomic flow ports* of the pipeline stages. In this way, the paths for the different instruction types is defined.

Figure 6 presents specialised instruction blocks for the instruction set of the ARM11 which correspond to the parallel stages, i.e. *ALUInstruction*, *MACInstruction*, and *LSInstruction*. The actual instructions are specialisations of these blocks and are reused in the behavioural model (cf. sec. B). Figure 7 depicts an extract of the internal block diagram defining the pipeline of an ARM11 system. The flow of instruction objects through the pipeline is restricted by the typed flow ports and the *item flow* of the connectors between the ports.

The block definition diagram of all involved components, the internal block diagram of the system layout and its refinements, i.e. the pipeline description, and the instruction inter-

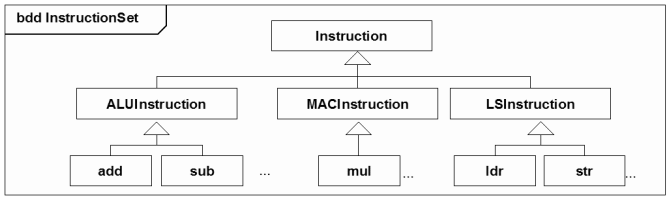


Fig. 6. Block definition diagram defining the instruction types of the ARM11.

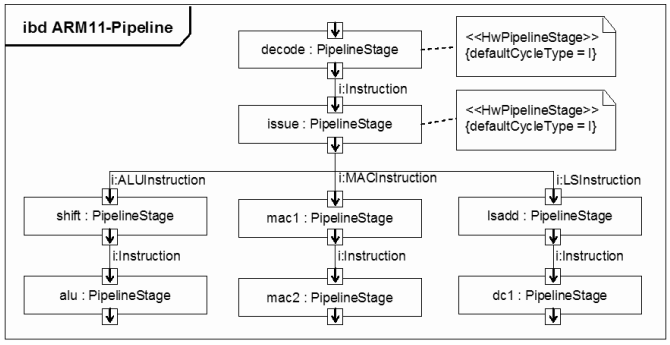


Fig. 7. Extract of an internal block diagram for the pipeline of an ARM11.

face hierarchy, form the architectural model. In the next section the pipeline behaviour and the timing behaviour of instructions are modelled with the help of sequence diagrams.

B. Instruction Behaviour Modelling

Processor instruction sets consist of instruction types with different timing behaviours. For example, a multiply instruction needs more processor cycles than an addition and registers may be needed in different pipeline stages. In the following, the behaviour specification is exemplary described by means of the instruction `add`. It adds two register values and stores the result in the first register.

$$\texttt{add} \quad r_6, \quad r_4 \qquad // \; r_6 = r_6 + r_4$$

Optionally, the instruction `add` supports *shift* flags to left/right shift (multiply/divide by a power of two) the value of the second register before the addition. The power is specified as a third parameter.

Figure 8 shows the specification of the pipeline behaviour of the instruction `add` in an ARM9 processor. Instances of the pipeline stage block corresponding to the stages modelled in the internal block diagram *pipeline* (cf. fig. 5) are shown on top. The *registerbank* is part of the diagram, because the instruction has to access registers during processing. Since the instruction is fetched by the *fetch* stage from the memory, a message to the `memory` element is modelled. The ≪`HwCycle`≫ annotation and its tag `cycleType` specify that the instruction consumes a memory cycle in the this stage. If the tag `cycleType` is omitted, the default cycle type defined in the internal block diagram *pipeline* is taken (cf. fig. 5). In the *decode* stage an I-cycle is consumed which is modelled by an annotated message to the *decode* stage. In the *execute* stage exclusive access to the first register of the instruction (r_6) is needed, because the result of the addition is stored into the first register, the second register (r_4) is only read. This is modelled by a message to the *registerbank* annotated with the ≪`HwRegisterComm`≫ stereotype. The value of the tag `registerLock` is set to 1, because the first register of the parameters is used to store the result. The tag `registerRead` is set to 2, because the value of the second register is read.

As mentioned above, the instruction `add` may be invoked with a *shift* flag, to shift the second register value before the addition. A UML alt-frame is used to distinguish the different behaviours depending on the instruction flags. In case of a set *shift* flag, an additional I-cycle is consumed in the *execute* stage to perform the shift operation and a third register is read. The additionally read register value is again modelled by a message to the *registerbank*. At the end of the calculations in the *execute* stage, the register containing the final value, is unlocked and the *registerbank* is informed that the registers are not needed anymore. The message to the *registerbank* annotated with ≪`HwRegisterComm`≫ and the tag `registerReady` set to 1,2, indicates that the first and second register (r_6 and r_4) are not needed anymore. In the following stages only I-cycles are consumed due to the `defaultCycle` tag value in the internal block diagram *pipeline*. In this way, the pipeline behaviour and the timing behaviour of the instruction set is modelled.

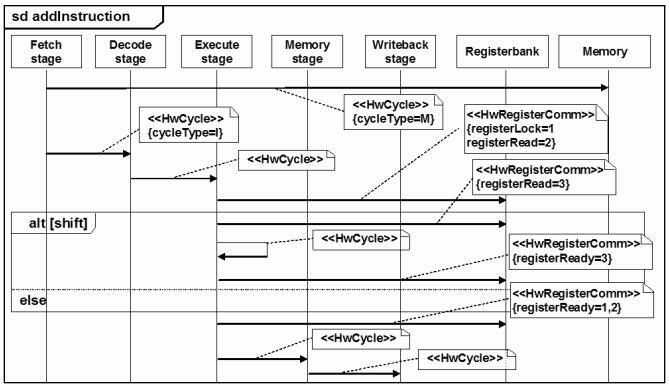


Fig. 8. Sequence diagram specifying the behaviour of the instruction `add`.

UML sequence diagrams are used to model the pipeline behaviour and the timing behaviour of instructions. The MARTE profile is enhanced by new stereotypes to specify the model in the desired level of detail. The instructions' behaviour is modelled cycle-accurate and register access is modelled, too.

The *architecture model* and the *instructions behaviour model* specify the architecture and the behavioural aspects of mobile systems. UML profiles, i.e. SysML and MARTE, are employed in the modelling. SysML helps in specifying the architecture and the MARTE profile enhances the model elements with semantical meanings.

V. EVALUATION

This section presents the evaluation of the simulation framework and the applicability of the proposed modelling methodology. Traces from an ARM9 system are gathered and used as

input for two system models. First, a system with direct connection between the processor and the bus is analysed. Second, instruction and data caches are placed in between the processor and the bus. These system architectures are modelled in UML according to the presented methodology (cf. sec. IV). Simulation results for the standard C-lib *memcopy* function, jpeg encoding, and jpeg decoding for both systems are presented in detail and compared to measured real world figures. Samples of simulation results for the AES algorithm (encoding and decoding) and the Dijkstra algorithm are presented, too. Finally, predictions for a system with an ARM11 processor are shown.

The applications have been traced on an OMAP5912 board [24] with the help of a Lauterbach tool set [25]. The OMAP board contains an ARM926EJ-S MPU [22] with a clock frequency of 192 MHz. The MPU provides a 16 kB instruction cache as well as an 8 kB data cache. Both caches use a block size of 32 bytes and are four-way associative. The AMBA system bus [26] connects the processor, caches, and peripherals like the main memory. A 32 MByte SD-RAM is used as main memory. The operating system used is a Linux branch from Montavista [27]. The Lauterbach tool is able to trace the executed instructions, the instruction addresses, and the data addresses in case of load and store instructions. The measured timings are cycle-accurate. Since neither the specification of the board, nor the specification of the used RAM give exact information about the memory access latencies, we measured the latency of load and store accesses. Table I shows the measured mean values as well as their standard deviation. The distance between the requested addresses of successive instructions is varied. It can be seen that the memory latency for successive instructions with an address distance of 1000 bytes is around 50 ns larger than for smaller distances. Therefore, we modelled the RAM accordingly with different latencies for a distance less and larger than 1000 bytes. Measurements of burst accesses are stable and the following values are used in the simulations; burst read: 10 ns and burst write 7 ns.

TABLE I. Mean measured memory latencies and standard deviations.

memory addresses distance	read	std.	write	std.
4 bytes	125 ns	0.45	74 ns	11.75
100 bytes	129 ns	3.83	77 ns	3.88
1000 bytes	172 ns	15.70	120 ns	15.61
4000 bytes	172 ns	15.44	120 ns	15.47

A. C-lib memcopy Function Analysis

As first application multiple runs of the standard C-lib *memcopy* function with an increasing block size to copy are performed. This application has a very high percentage of memory operations, so that the caches, the bus, and especially the main memory affect the performance of the system.

The *memcopy* function is executed 32 times, each time the buffer to copy is increased by 1024 bytes, starting with a buffer size of 1024 bytes. Traces gathered on the OMAP board are used to simulate the runtime on the two aforementioned systems, i.e. a system without caches and a system with instruction and data caches.

Figure 9 presents a comparison of the measured real-world figures of the *memcopy* runs and simulation results with traces of the *memcopy* function for the system without caches. The mean values, the minimum and maximum of the measured figures, and the simulation results are shown. The variations of the measured figures and the simulation results can be explained by interrupts of the Linux kernel. The y-axis on the right presents the relative errors of the mean values of the simulations compared to the measured figures. It can be seen that the relative error decreases with an increasing block size of the *memcopy* function. An explanation for this is the aforementioned fact of varying RAM latencies which is not modelled in detail in our simulator and just simulated in an abstract way by using mean access latencies. The influence of this imprecise main memory modelling is larger for small traces, because only a few memory accesses occur.

Figure 10 presents a comparison of measurements and simulation results as described above for the system with instruction and data caches. The mean relative error of these simulations is larger than for the system without caches, but the relative errors of the predictions decreases with an increasing size of the *memcopy* traces. This can be explained by the fact that in this configuration (with caches) significantly less memory accesses influence the runtime, because the caches handle most of the memory requests. This deviation decreases with the amount of memory accesses, so that for larger trace files the influence of the abstract RAM modelling is reduced.

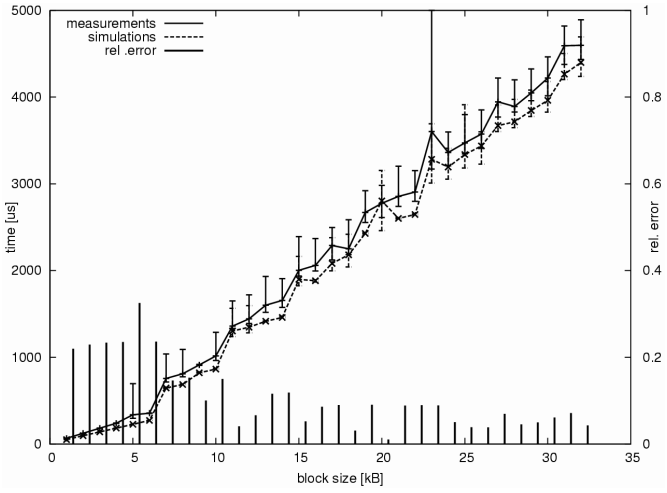


Fig. 9. Comparison of real-world measurements and simulation results of the standard C-lib *memcopy* function for a system without caches.

B. MiBench Algorithms Analysis

The number of instructions and the number of memory accesses for real applications like jpeg encoding and decoding is

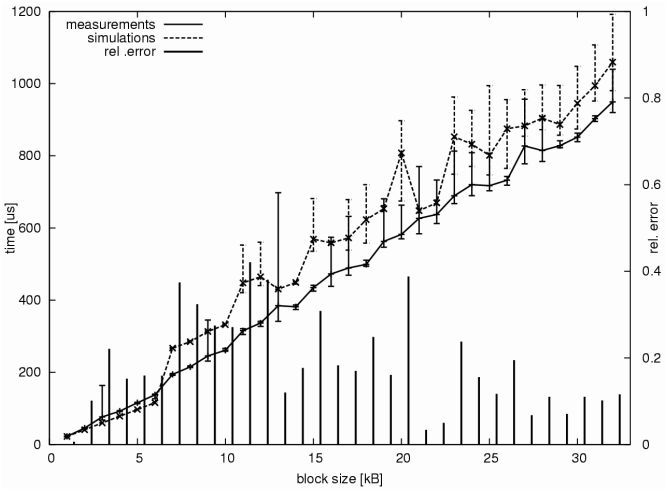


Fig. 10. Comparison of real-world measurements and simulation results of the standard C-lib *memcopy* function for a system with caches.

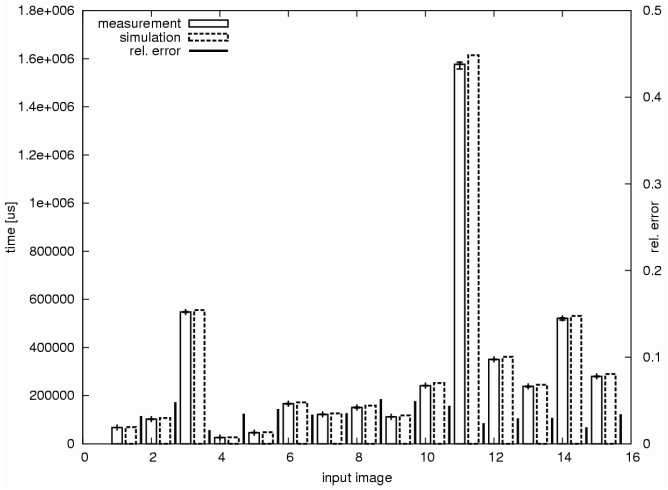


Fig. 11. Comparison of real-world measurements and simulation results of the jpeg encoding algorithm for a system with instruction and data caches.

magnitudes larger than for the *memcopy* function. The following section presents measurements and simulations for applications taken from the MiBench suite [20]. Simulation results of the jpeg encoding and decoding algorithms are presented in detail. Sample results for AES and the Dijkstra algorithm are presented. We used inputs from the MiBench suite or from the MiDataSet collection [21].

The Mibench traces are much larger than the traces of the *memcopy* function. The number of instructions of the *memcopy* traces ranges from around 180 to ca. 17,000 depending on the size of the buffer to copy. The number of instructions in the jpeg traces range from around 2.4 million to 160 million depending on the size of the input. The jpeg algorithms have been also traced on the OMAP board and measurements for the systems with and without caches have been performed.

Figure 11 shows measured real-world figures and the simulated runtimes for the jpeg encoding of inputs taken from the MiDataSet collection. The measurements and the simulations are performed with instruction and data caches. On the x-axis the input images are listed and the measured and simulated runtimes are plotted on the y-axis. The right y-axis is the scale for the relative error. It can be seen that the measured runtimes on the OMAP board do not vary like the measurements of the *memcopy* function. This is due to the much larger runtime so that interrupts of the Linux kernel do not significantly influence the measurements. The relative error (right y-axis) of the simulation results is plotted for each input. The mean value of the relative error is around 3.5%. The simulation results are closer to the measured real-world figures than in the *memcopy* analysis. This underlines the observation that the relative error decreases with the number of instructions in the trace due to the abstract modelling of the main memory.

Figure 12 shows the runtime predictions of the jpeg encoding algorithm for a system without caches. The relative error (right y-axis) of the simulation prediction is again plotted for each input. It can be seen that the measured real-world figures are predicted by the simulations for this configuration. The mean relative error for the predictions is around 2%.

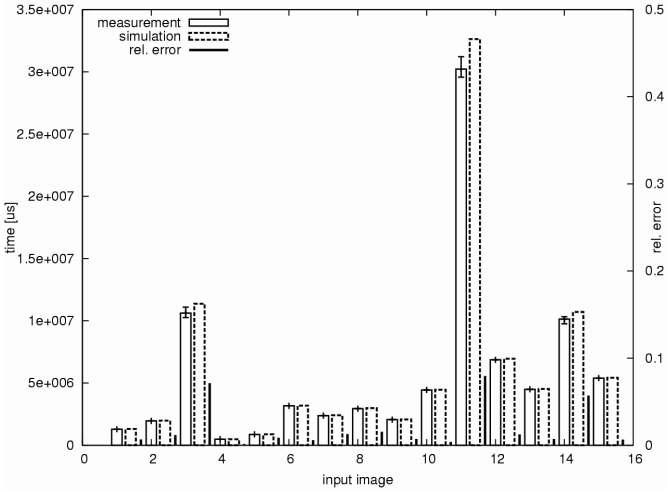


Fig. 12. Comparison of real-world measurements and simulation predictions of the jpeg encoding algorithm for a system without caches.

The results for the jpeg decoding algorithm are similar to the results of the jpeg encoding. Measurements and simulations of six input images result in a mean relative error of less than 1% in case of the system without caches and an error of around 4% for the system with caches. Sample measurements and simulations for the AES algorithm and the Dijkstra algorithm are also in this range. The relative error for AES encoding is less than 2%, for AES decoding around 4%, and the Dijkstra runtime is simulated with less than 1% deviation.

Due to the lack of cycle-accurate measurements for an

ARM11 system, the Linux *gettimeofday* function has been used to measure the runtime of ARM9 jpeg encoding and decoding binaries on an ARM11. The ARM11 system runs at a clock speed of 210 Mhz and incorporates an 32 kB instruction and a 32 kB data cache. The measurements show a speedup of 1.28 for encoding and a speedup of 1.14 for decoding, which implies a 12% higher speedup of encoding. The encoding process has a higher ratio of arithmetical operations which are processed in I-cycles and therefore, the encoding profits more from the higher clock rate of the ARM11 than the decoding process. Due to the unknown memory access latencies of the ARM11, the ARM9 OMAP board settings are used in the simulations. Thus, only the ratio of the predicted speedups can be compared with the measurements. The simulations of ARM9 traces on the ARM11 model results in a speedup prediction of 1.39 for encoding and 1.23 for decoding. This implies a 13% higher speed up of encoding. Thus, nearly the same speedup ratio as in the measurements (12%) is predicted.

The evaluation shows that the accuracy of the simulation results increases with the size of the input traces. An explanation for this is the abstract simulation of RAM modules in the timing simulator. The simulated systems are modelled according to the proposed modelling methodology which shows the applicability of the methodology.

VI. CONCLUSION

Simulations of instruction traces of applications gathered on existing hardware to predict the performance of non-existing hardware architectures is a common means in the area of performance engineering. This paper presented a methodology for modeling system architectures of mobile systems, and especially processor details. These models are used to configure a modular trace-based timing simulator. The modular design of the simulator allows for a flexible configuration so that a large number of systems may be configured. The modelling methodology is UML-based. The SysML profile and the MARTE profile are utilised by the methodology to enhance the model with the necessary semantics. The stereotypes ≪HwPipelinestage≫, ≪HwRegisterbank≫, ≪HwWritebuffer≫, ≪HwRegisterComm≫, and ≪HwCycle≫ are introduced to model the systems with the desired level of detail. The UML diagrams are not only used to describe the system but are used to configure the simulator. An ARM9 case study shows the applicability of the modelling methodology and of the modular simulator. Runtime predictions for system architectures with caches and without caches are presented and compared to measured real-world figures.

Future work will focus on an extensive analysis of further applications and algorithms from the MiBench suite. The main memory module has to be modelled and simulated in more detail. Thus, the modelling methodology will be enhanced to support this requirements, so that an increase of the predictions' accuracy can be expected. Furthermore, the ARM9 case study will be transfered to an ARM11-based system.

REFERENCES

[1] OMG, "Omg systems modeling language (omg sysml), v1.0," Sep 2007.
[2] ——, "A uml profile for modeling and analysis of real-time and embedded systems (marte), beta 1," Aug 2007.
[3] J. J. Yi, L. Eeckhout, D. J. Lilja, B. Calder, L. K. John, and J. E. Smith, "The future of simulation: A field of dreams," *Computer*, vol. 39, no. 11, pp. 22–29, 2006.
[4] L. Eeckhout, K. de Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," in *ISPASS'00: Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*. Washington, DC, USA: IEEE Computer Society, 2000, pp. 1–6.
[5] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.
[6] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Tech. Rep. CS-TR-1997-1342, 1997.
[7] F. Bellard, "Qemu homepage," http://fabrice.bellard.free.fr/qemu/, 2007.
[8] C. A. Prete, M. Graziano, and F. Lazzarini, "The charm tool for tuning embedded systems," *IEEE Micro*, vol. 17, no. 4, pp. 67–76, 1997.
[9] H. Cain, K. Lepak, B. Schwartz, and M. Lipasti, "Precise and accurate processor simulation," in *Proceedings of the Fifth Workshop on Computer*, Feb 2002, pp. 13–22.
[10] J. Flanagan, B. Nelson, J. Archibald, and G. Thompson, "The inaccuracy of trace-driven simulation using incomplete mulitprogramming trace data," in *MASCOTS*, 1996.
[11] C. U. Smith and L. G. Williams, *Performance Solutions, A Practical Guide to Creating Responsive, Scalable Softwarre*. Addison-Wesley Pearson Education, 2001.
[12] M. Woodside, D. C. Petriu, D. B. Petriu, H. Shen, T. Israr, and J. Merseguer, "Performance by unified model analysis (puma)," in *WOSP '05: Proceedings of the 5th international workshop on Software and performance*. New York, NY, USA: ACM Press, 2005, pp. 1–12.
[13] L. Pustina, S. Schwarzer, M. Gerharz, P. Martini, and V. Deichmann, "Performance evaluation of a dvb-h enabled mobile device system model," in *WOSP '07: Proceedings of the 6th International Workshop on Software and Performance*. New York, NY, USA: ACM, 2007, pp. 164–171.
[14] S. Balsamo and M. Marzallo, "Performance evaluation of uml system architectures with mutliclass queueing network models," in *WOSP*, 2005.
[15] OMG, *UML Profile for Schedulability, Performance, and Time Specification: Version 1.0*. Object Management Group, 2003.
[16] X. Wu and M. Woodside, "Performance modeling from software components," vol. 29, no. 1. New York, NY, USA: ACM, 2004, pp. 290–301.
[17] R. Klein, K. Travilla, and M. Lyons, "Performance estimation of mpeg4 algorithms on arm based designs using co-verification," 2002.
[18] A. Phansalkar, A. Joshi, L. Eeckhout, and L. John, "Measuring program similarity: Experiments with spec cpu benchmark suites," in *ISPASS'05: Proceedings of the 2005 IEEE International Symposium on Performance Analysis of Systems and Software*. Austin, TX: IEEE, 3 2005, pp. 10–20.
[19] J. Yi and D. Lilja, "Simulation of computer architectures: Simulators, benchmarks, methodologies, and recommendations," *IEEE Transactions on Computers*, vol. 55, no. 3, pp. 268–280, 2006.
[20] 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 *WWC '01: Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop on*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.
[21] G. Fursin, J. Cavazos, M. O'Boyle, and O. Temam, "Midatasets: Creating the conditions for a more realistic evaluation of iterative optimization," in *International Conference on High Performance Embedded Architectures & Compilers (HiPEAC 2007)*, January 2007.
[22] *ARM926EJ-S Technical Reference Manual*, ARM Limited, 2003.
[23] *ARM11 MPCore Technical Reference Manual*, ARM Limited, Feb 2008.
[24] *OMAP5912 Applications Processor Data Manual*, Texas Instruments, Dec 2003.
[25] Lauterbach, "Lauterbach homepage," http://www.lauterbach.com, 2008.
[26] *AMBA Specification (Rev 2.0)*, ARM, May 1999.
[27] "Montavista linux," http://www.mvista.com, 2008.