2014 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)

# An ESL Timing & Power Estimation and Simulation Framework for Heterogeneous SoCs

Kim Grüttner*, Philipp A. Hartmann*, Tiemo Fandrey*, Kai Hylla*, Daniel Lorenz*,
Stefan Stattelmann†, Björn Sander‡, Oliver Bringmann¶,
Wolfgang Nebel§ and Wolfgang Rosenstiel¶
*OFFIS - Institute for Information Technology, Eschwerweg 2, 26121 Oldenburg, Germany,
E-Mail: {gruettner, hartmann, fandrey, hylla, lorenz}@offis.de
† ABB Corporate Research, Wallstadter Straße 59, 68526 Ladenburg, Germany, E-Mail: stefan.stattelmann@de.abb.com
‡Hitex Development Tools GmbH, Greschbachstr. 12, 76229 Karlsruhe, Germany, E-Mail: bjoern.sander@hitex.de
§C.v.O. Universität Oldenburg, Ammerländer Heerstr. 114-118, 26111 Oldenburg, Germany,
E-Mail: nebel@informatik.uni-oldenburg.de
¶Universität Tübingen, Sand 13, 72076 Tübingen, Germany, E-Mail: oliver.bringmann@uni-tuebingen.de

***Abstract*—Consideration of an embedded system's timing behaviour and power consumption at system-level is an ambitious task. Sophisticated tools and techniques exist for power and timing estimations of individual components such as custom hard- and software as well as IP components. But prediction of the composed system behaviour can hardly be made without considering all system components. In this paper we present an ESL framework for timing and power aware rapid virtual system prototyping of heterogeneous SoCs consisting of software, custom hardware and 3rd party IP components. Our proposed flow combines system-level timing and power estimation techniques with platform-based rapid prototyping. Virtual executable prototypes are generated from a functional C/C++ description, which then allows to study different platforms, mapping alternatives, and power management strategies. We propose an efficient code annotation technique for timing and power, that enables fast host execution and collection of power traces, based on domain-specific workload scenarios.**

## I. Introduction

The increasing use and growing complexity of MPSoCs (Multi-Processor System-on-Chip) and the resulting potential interaction of system components makes it very hard to analyse the timing and power consumption of complex embedded systems. For an early analysis of extra-functional system properties, the estimation of execution times and power consumption of MPSoC components becomes more and more important. In the last years, a lot of effort has been spent in estimating execution times and power on RT-level. However, today's application and target platform complexity inhibits full system simulations at such a low level of abstraction. Furthermore, simulating the different components of the target platform separately is not feasible since predictions and analyses of the overall system can hardly be made if components are only considered in isolation.

Thus, for complex applications on large MPSoCs the interaction of all components must be taken into account to capture the behaviour of the entire system. This is essential for accurate power and timing estimations. For specific platforms, proprietary simulation environments are available for both timing and power models. But a common and open methodology, suitable for a large range of platforms and designs, is still missing. Such a framework would allow comparing different platform characteristics and thus rapid prototyping and design space exploration. Performance bottlenecks and power peaks within the entire system could be identified in early design phases, where modifications of the system are easier and less costly than in later phases. For these reasons, a methodology and modelling infrastructure is required, which allows integration of timing and power information from RT-level estimations into a fast executable virtual platform at system-level.

In this paper, a methodology for estimating execution times and power consumption of hardware and software components in multiprocessor systems is presented. To simulate the timing and power behaviour of hardware and software for a given target architecture, low-level timing and power properties are annotated to the source code of the functional model. Then, the annotated source code is compiled and natively executed on the simulation host. Instead of directly annotating power and time values to the source code, three different approaches have been combined:

- For software, the binary-level control flow for the target processor's architecture is simulated in addition to the functionality. This allows a dynamic estimation of timing and power properties without interpreting target code on the simulation host.
- To consider custom hardware, the resulting controller and data path timing and power properties after high-level synthesis are simulated in addition to the functionality. This allows a dynamic estimation of timing and power properties without co-simulating RTL code on the simulation host.
- For third-party black-box *intellectual property (IP)* components or pre-existing RTL modules like memories, interconnects and communication peripherals, timing and power information is modelled as *Power State Machine (PSM)*. A PSM observes the interaction of the black-box IP component with its system environment and triggers the transition between different power modes, either based on data sheet information, designer knowledge, or trace-based power characterisation performed at RT-level.

By combining these approaches with a common timing

and power model, the interaction between software, custom hardware, and third-party black-box IP components can be analysed for complex MPSoCs running real application code using a source-level host-based simulation. In this paper a proof-of-concept integration based on SystemC is presented and evaluated using an MP3 player application on an ARM-based SoC.

## II. Related Work

To facilitate the design of complex systems, *virtual prototypes (VPs)* which are described in a system-level design language like SystemC [9] are in widespread use. A standard technique to improve simulation performance of these VPs is *transaction-level modelling (TLM)*, which separates the modelling of communication between system components and the computations performed inside these components [6]. In transaction-level models, low-level timing properties are often added to the source code of hardware and software components to perform a timed simulation. Neither SystemC nor TLM offer built-in features to model and trace extra-functional properties like power consumption.

In [4] an estimation of power consumption at behavioural level using SystemC is described. This approach extends signals with power macro models of RT components and overloaded arithmetic, logic, and assignment operators carrying power information. The main drawback of this approach is its limited application to system-level models, since the modelling style is very close to RTL design.

ActivaSC is a non-intrusive extension for activity-based analysis of SystemC models [20]. With this approach switching activity can be obtained from SystemC simulations without modifying the functional specification. The analysis and conversion of activity data to power consumption is not part of this approach.

In [10] annotations for power modelling in SystemC at Transaction Level with *dynamic voltage and frequency scaling (DVFS)* capabilities is presented. The basic idea is to separate the functional (IP) model from the power model and specific power information. Power monitors are used to trace and check power consumption. This separation has also been followed in our work. As an extension we explicitly distinguish between different extra-functional models for software, custom hardware, and IP components.

A top-down power and performance estimation methodology for heterogeneous multiprocessor systems-on-chip at *Electronic System Level (ESL)* is proposed in [17]. By separating the system functionality from its architecture, different design options can be assessed with low effort. The simulation-based approach permits to evaluate the effects of dynamic power management. In contrast to our approach the focus is mainly on software and a higher abstraction level without considering automatic extra-functional model generation.

A methodology for power estimation of SystemC transaction level models for bus power is presented in [5]. It describes a bus power characterization approach, a hierarchical representation of transaction level (TL) data, and a power model interface/mapping mechanism to augment TL simulation models with power information. These techniques were implemented for IBM CoreConnect based architectures. We have chosen a similar approach for the Power State Machine generation of our bus power model.

In [2] a framework for system-level power estimation using heterogeneous power models is proposed. The integration of heterogeneous component power models is implemented as a network of power monitors. The monitor-based framework provides interfaces, facilitating the integration of component simulation models on one hand, and a variety of heterogeneous power models on the other. Power monitors enable each component model to be associated with multiple (distinct) power models of differing accuracy and efficiency, or with configurable power models that can be tuned to different accuracy/efficiency levels and thus delivering a trade-off between power consumption, accuracy, and simulation speed. In our work we propose a native host-based simulation of functional and power models with a flexible tracing infrastructure that allows efficient SystemC TLM simulation with a scalable amount of context switches to enable the same accuracy vs. simulation time trade-off.

The back-annotation of power properties obtained from cycle-accurate custom hardware descriptions at RTL to a pure functional representation in C has been presented in [21]. Based on power macro models for each RTL component and explicit knowledge about the structural decomposition from functional C to RTL allows the instantiation of virtual power-aware components for each operator in the RTL. These virtual components are fed with the same values of the particular RTL-operation and are transformed to executable C models linked to the functional input model. The main drawback of this approach is that the execution of the virtual components slows down the functional system simulation drastically. In our approach the concept of *hardware basic blocks (HBBs)* is used to obtain an efficient executable functional model with power information.

Determining extra-functional properties of embedded software through source-level simulation has been proposed as an alternative to using an *instruction set simulator (ISS)*. In a source-level simulation, the source code of software components is enriched with annotations describing the extra-functional properties to be analysed. The annotated source is then compiled for the simulation host. Using the resulting host-compiled binary, extra-functional properties of the software running on the target architecture can be obtained while natively executing the software on the simulation host. In this paper we have extended the software timing back-annotation presented in [16] and [15] with a software power model as presented in [14].

## III. Proposed Methodology

### A. Overview

Our proposed concept for a rapid prototyping framework (based on [7]) is illustrated in Figure 1, which follows the Platform-Based Design approach with a separation of application model ⓐ, platform model ⓒ, and mapping description ⓑ. The platform model is a graph consisting of processing element, interconnect, and memory nodes. The parallel application is described as a task graph and a predefined communication and synchronisation scheme along the

edges between different tasks and service nodes. In a separate mapping step tasks and service nodes are mapped onto the processing element and memory nodes of the platform model.

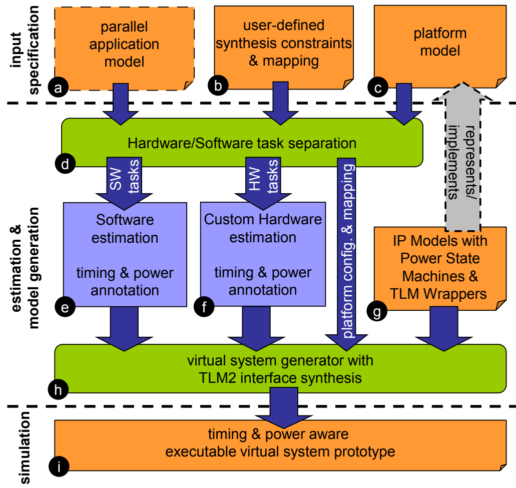


Fig. 1. Proposed Rapid Prototyping Framework

The most important property of the proposed framework is that *extra-functional timing and power modelling is separated from functional application specification and modelling*. For pre-defined black-box IP models ⓖ Power State Machines observe the communication behaviour of the component and extrapolate the timing properties and power consumption based on the externally observable behaviour. For software ⓔ and custom hardware ⓕ components timing and power estimation based on cross-compilation for the target processor and high-level synthesis for the target technology is performed. For these components, timing and power back-annotation to the executable input model is performed. During virtual system generation ⓗ timing and power annotated software, custom hardware, and IP components are functionally connected through TLM-2.0 wrappers. Furthermore, the power and timing annotation are connected to an extra-functional model to enable tracing of these properties during virtual system prototype execution ⓘ.

### B. Input Specification

**Parallel application model:** In our *Parallel Application Model*, the system is represented as a set of parallel, communicating processes, representing hardware or software independent tasks and services. A Service Node is the modelling primitive for inter-task communication. It provides a set of interfaces to its client tasks. Interfaces are used to group services. The services themselves are side-effect free C functions. Different synchronisation protocols can be chosen.

A *Service Node* is a tuple $\mathcal{SN} = (\mathcal{S}, \mathcal{I}, \mathsf{protocol})$, where

1) $\mathcal{S}$ is a set of symbols, representing the provided *services*.
2) $\mathcal{I} = \{\mathsf{IF}_0, \ldots, \mathsf{IF}_k\}$ is a set of *interfaces*, where each $\mathsf{IF}_i \subseteq \mathcal{S}$ denotes a subset of services.
3) $\mathsf{protocol} = \{\mathsf{FIFO}, \mathsf{handshake}, \mathsf{none}\}$ specifies the synchronisation protocol among clients of this Service Node.

Tasks communicate with other tasks via Service Nodes, statically bound to *ports*. The internal behaviour of a task is described as executable sequential C code with explicit *service calls* to Service Nodes. All calls to Service Nodes are blocking, i.e. the caller's behaviour can be continued only after the service call has been completed. When multiple tasks are requesting non-mutual exclusive services from the same Service Node scheduling is required. More details about this can be found in [3].

A *Task Node* is a tuple $\mathcal{TN} = (\mathcal{P}_s, \mathcal{P}_a, \mathcal{P}_e, \mathsf{F})$, where

1) $\mathcal{P}_s = \{\mathsf{p}_{s_0}, \ldots, \mathsf{p}_{s_n}\}$ is a set of *ports*, each representing a set of associated services $\mathsf{p}_{s_i} \subseteq \mathcal{S}_i$.
2) $\mathcal{P}_a$ is the activation port. There are two kinds of activation ports: *AND* ports activate the task when all bound activation edges observe an activation event. *OR* ports activate the task when at least at one of the bound activation edges an activation event occurs.
3) $\mathcal{P}_e = \{\mathsf{p}_{e_0}, \ldots, \mathsf{p}_{e_m}\}$ is a set of exit ports. An event on one of these port is emitted, when the execution of the task's functionality has been completed.
4) $\mathsf{F}$ specifies the functional behaviour of the task.

A *Task Node* describes a Runnable i.e. a process. A task starts running immediately after its activation through $\mathcal{P}_a$ and can only be blocked by communication on its ports $\mathcal{P}_s$. A task can be (self-)triggered again after a certain amount of time (time-triggered or periodic task).

Based on these notations for the modelling primitives, the overall *Application Model* $\mathcal{AM}$ consists of

- a set of Task Nodes $\mathfrak{T} = \{\mathcal{TN}_0, \ldots, \mathcal{TN}_n\}$,
- a set of Service Nodes $\mathfrak{S} = \{\mathcal{SN}_0, \ldots, \mathcal{SN}_m\}$, and
- a service port binding function $\mathfrak{B}_s : \bigcup_{\mathcal{TN} \in \mathfrak{T}} \mathcal{P}_{s_{\mathcal{TN}}} \rightarrow \mathfrak{S}$, that uniquely associates each port $\mathsf{p}_s$ to a compatible Service Node: $\forall \mathsf{p}_s : \mathfrak{B}(\mathsf{p}_s) = \mathcal{SN} \Leftrightarrow \exists\, \mathsf{IF} \in \mathcal{I}_{\mathcal{SN}} : \mathsf{p}_s = \mathsf{IF} \Leftrightarrow \mathsf{p}_s \models \mathcal{I}_{\mathcal{SN}}$,
- an exit to activation port binding function, that associates each exit node with a non-empty set of activation nodes,
- a set of initial activation edges, each bound to a unique activation port.

**Platform Model:** The *Platform Model* $\mathcal{PM}$ is composed independently from the application model. It is a pure structural and non-executable parameterizable representation of the execution platform. It consists of a set of Processing Elements $\mathcal{PE} \in \{\mathcal{SW}, \mathcal{HW}, \mathcal{IP}\}$, where

- Software Processor $\mathcal{SW} = (\mathsf{ISA}, \mathsf{DMs}, \mathsf{IMs}, \mathcal{IS})$ with a specific Instruction Set Architecture (ISA), local data memory DMs and instruction memory size IMs, and a set of Initiator Sockets ($\mathcal{IS}$),
- Custom Hardware Component $\mathcal{HW} = (\mathsf{area}, \mathcal{IS}, \mathcal{TS})$ with a specific area constraint (area), a set of Initiator Sockets ($\mathcal{IS}$), and a set of Target Sockets ($\mathcal{TS}$),
- IP Component $\mathcal{IP} = (\mathcal{IS}, \mathcal{TS})$,

Memory Elements $\mathcal{ME} = (\mathsf{width}, \mathsf{size}, \mathcal{TS})$, and Router Elements $\mathcal{RE} = (\mathsf{dwidth}, \mathsf{scheduling})$ with a data width and selectable scheduling policy. A binding function uniquely associates each Initiator and Target Socket to a Router Element. $\mathcal{PE}$s, $\mathcal{ME}$s, and $\mathcal{RE}$s can be uniquely associated to a Power Island $\mathcal{PI}$. Each power island has its own set of pairs $(f, V_{\mathrm{dd}})$ that defines valid steps for dynamic voltage and frequency scaling (DVFS).

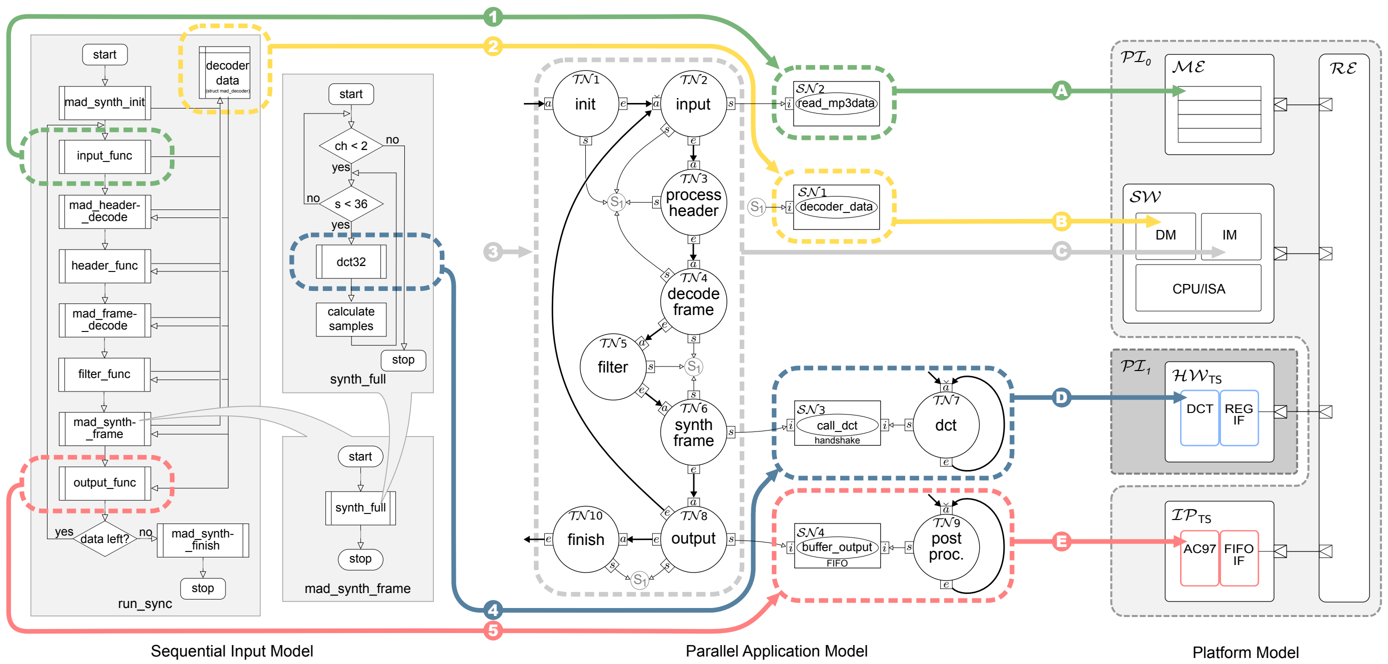


Fig. 2. Mapping of input models

**Mapping:** Figure 2 depicts the process of mapping an initial functional specification to a platform model, based on an MP3-decoder design that we will use as a demonstrator throughout this paper. The left part of Figure 2 shows the mapping of the initial functional specification to our parallel application model. This step involves the partitioning into appropriate tasks and services, as well as the identification of potential parallelism. We start from C code taken from [19]. The code outlined in the simplified flowcharts covers the main loops and sub-routines of the PCM synthesis, including a *discrete cosine transform (DCT)*. The sub-routines exchange data over a central data structure. Accessing the central decoder state and reading input data are both considered as a service and mapped to corresponding service nodes $\mathcal{SN}1$ and $\mathcal{SN}2$ (Figure 2, ① and ②). For most parts of the functional specification we keep the sequential nature of the decoding algorithm and map them to task nodes that reproduce the original structure of the code (Figure 2, ③). These tasks are activated one by one and require no service nodes for inter-task communication. However, they all rely on $\mathcal{SN}1$ in order to access the common decoder state data. Without loss of generality, we focused on the considerably complex DCT in the PCM synthesis part of the MP3 decoder and separated it to a concurrent task node (Figure 2, ④). In order to be able to introduce some temporal decoupling, we also parallelised the processing of output samples (Figure 2, ⑤). The corresponding task nodes $\mathcal{TN}7$ and $\mathcal{TN}9$ require additional service nodes $\mathcal{SN}3$ and $\mathcal{SN}4$ to define their communication.

The right hand side of Figure 2 shows the result of mapping the application model's task graph to a target platform model. The platform model consists of five processing elements: a software processor $\mathcal{SW}$, a custom hardware component $\mathcal{HW}$, and an IP component $\mathcal{IP}$. Two non-processing elements complete the platform: a memory element $\mathcal{ME}$ and a router element $\mathcal{RE}$. The platform model is divided into two power islands. One holds the custom hardware and the other one governs all remaining platform elements. Task nodes can be mapped to any kind of $\mathcal{PE}$. We map the bulk of task nodes to the single $\mathcal{SW}$ (Figure 2, Ⓒ). This mapping is valid, because the sequential execution imposes a static schedule on the mapped nodes. A static schedule is evident as each exit port is bound to exactly one single activation port within the set of task nodes. While the separated task node $\mathcal{TN}7$, which provides the DCT functionality, is mapped to $\mathcal{HW}$ (Figure 2, Ⓓ), the task node $\mathcal{TN}9$ with the output functionality is mapped to $\mathcal{IP}$ (Figure 2, Ⓔ). Service nodes can be mapped to $\mathcal{SW}$, $\mathcal{ME}$ or register interfaces (Reg IF) of $\mathcal{HW}$ or $\mathcal{IP}$ with a target socket ($\mathcal{TS}$), depending on the mapping target of its client task nodes. The service node $\mathcal{SN}1$ that is related to the shared decoder state data is mapped to $\mathcal{SW}$ as well, respectively to its local data memory, in order to avoid bus contention (Figure 2, Ⓑ). However, the input data related service node $\mathcal{SN}2$ is mapped to an external memory element in order to accommodate the size of the input data (Figure 2, Ⓐ). The inter-task communication related service nodes $\mathcal{SN}3$ and $\mathcal{SN}4$ are mapped to the interfaces of the corresponding $\mathcal{HW}$ and $\mathcal{IP}$. This mapping is legitimate as both $\mathcal{HW}$ and $\mathcal{IP}$ provide a target socket.

## C. Extra-Functional Model

The timing and power properties are represented in a common extra-functional model for all estimated platform components HW, SW, and IP. The proposed extra-functional model allows scalability for different operation conditions $(f, V_{\text{dd}})$ per component, as defined by its associated Power Island $\mathcal{PI}$. Each Task Node $T \in \mathfrak{T}$ mapped on platform component $X \in \{\mathcal{PE}, \mathcal{ME}, \mathcal{RE}\}$, written $T \rightarrow X$, can be modelled as an abstract state transition system with a set of states $S_{T\rightarrow X}$ (basic blocks), a set of Boolean Guards $G_{T\rightarrow X}$ (branches), and a state transition function $S_{T\rightarrow X} \times G_{T\rightarrow X} \rightarrow$

$S_{T\to X} \times \Gamma_{T\to X}$. Each state is annotated with the tuple $\Gamma_{T\to X} = (\mathsf{Cycles}, \mathsf{Capacitance})$. $\mathsf{Cycles} \in \mathbb{N}_0$ is the number of clock cycles, and $\mathsf{Capacitance} \in \mathbb{N}_0$ is the average switched capacitance in $nF$.

The execution of the functional model can be represented as a use-case and data-dependent path through the possible states. For the execution sequence from state/basic block $s_i$ to $s_j$: $s_i \to^* s_j$, with $s_i, s_j \in S_{T\to X}$ a trace of extra-functional information $\gamma_0, \ldots, \gamma_n$, with $\gamma_i \in \Gamma_X$ can be obtained. Given the clock frequency $f$ at time $t$ the duration $d(\gamma_i)$ of each state can be calculated as $d(\gamma_i) = \mathsf{Cycles}(\gamma_i)/f(t)$. With this information an index function $p(T \to X, t) \in \{\mathsf{Capacitance}(\gamma_0), \ldots, \mathsf{Capacitance}(\gamma_n)\}$ can be generated as a lookup table to access the average switched capacitance of component $T$ mapped on component $X$ at time $t$. With this information the dynamic power consumption over time of task $T$ mapped on component $X$ is calculated:

$$P_{\text{dyn}}(T \to X, t) = \frac{1}{2} \cdot V_{\text{dd}}(X,t)^2 \cdot f(X,t) \cdot p(T \to X, t)$$

For static power a leakage resistance model with $R_{\text{leakage}}$ is used to calculate the static power over time:

$$P_{\text{stat}}(T \to X, t) = V_{\text{dd}}(X,t)^2 / R_{\text{leakage}}(T \to X)$$

The total system's power consumption over time is:

$$P_{\text{tot}}(t) = \sum_{\substack{\forall T \to X,\\ X \in \mathcal{PM}}} \left(P_{\text{dyn}}(T \to X, t) + P_{\text{stat}}(T \to X, t)\right)$$

### D. Estimation & Extra-Functional Model Generation

The estimation and extra-functional model generation step augments the functional input specification model with timing and power properties. Depending on the user-defined mapping, each task of the parallel application model is characterized regarding its timing and power properties. During hardware/-software task separation (Figure 1, ⓓ) the functional C code is extracted from each task for analysis of its timing and power consumption. The following three sections describe the estimation and extra-functional model generation for Software ⓔ, Hardware ⓕ, and IP components ⓖ.

*1) Software:* Timing and power properties of a software task are analysed using the optimized target machine code of the software. The results of the respective low-level analyses are back-annotated to the source code of the program to estimate its extra-functional properties during a system-level simulation based on SystemC. As compiler optimizations often obfuscate the relation between source code and target binary code, matching both program representations to perform the back-annotation requires an intricate analysis of the program structure on both levels.

To overcome the issue of matching the structure of the source code and the machine code, the design methodology proposed in this paper uses the annotation approach described in [15], [16]. This technique is based on the reconstruction of compiler-generated debugging information and the dynamic simulation of binary-level control flow. Modelling binary-level control flow in parallel to the functionality of the software allows the dynamic selection of annotations during simulation. Thus, additional information can be used to select annotations

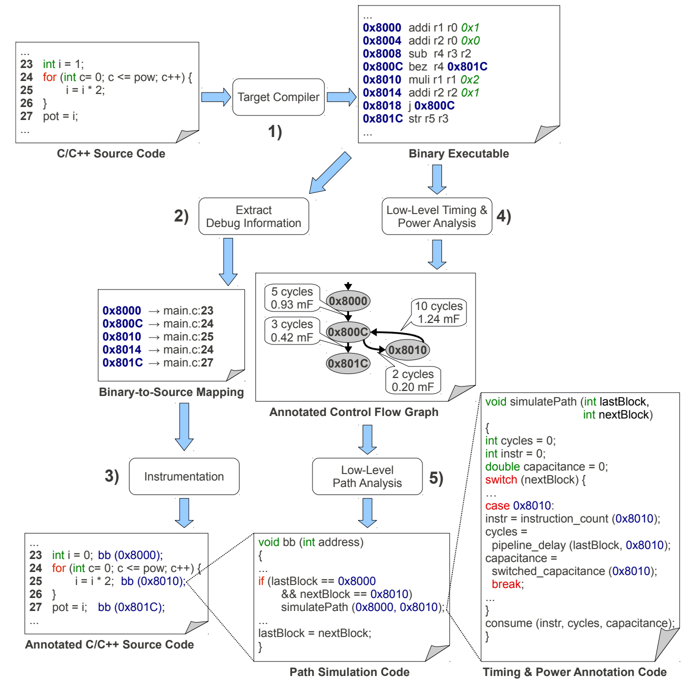


Fig. 3. Software Annotation Flow

more accurately while the program is executed. This enables a precise consideration of compiler optimizations modifying the program structure like loop unwinding, e.g. by correcting the number of simulated iterations to compensate for the unrolling performed by the compiler.

The complete annotation work flow for software tasks is depicted in Figure 3. After a program has been cross-compiled for the target architecture using a standard compiler (Figure 3, step 1), the compiler-generated debug information is used to relate the source code and the binary code (Figure 3, step 2). Instead of using this information to relate source-level and binary-level basic blocks for a direct annotation of low-level properties, the proposed method only uses this information for a tentative estimation of which source code portions correspond to the binary-level basic blocks. For every binary-level basic block, an equivalent source code position is determined from this data. During instrumentation (Figure 3, step 3), address references to the binary-level basic blocks are added to the source code at the determined position.

The extra-functional properties of the basic blocks in the binary code are obtained using an analysis of machine instructions which considers the low-level effects on the target processor (Figure 3, step 4). The execution time of each basic block in terms of clock cycles is obtained using the commercial timing analysis tool AbsInt aiT [1]. The power consumption per basic block is determined using an instruction-dependent power model generated through exhaustive simulation of the processor RTL model [14]. The result of these analyses is an annotated control flow graph of the binary executable. The edges in this graph, which describe the transition between basic blocks during an actual execution of the program, are labelled with the execution time and switched capacitance required by the respective sequence of machine instructions. Based on the binary-level CFG, the program control flow on the target

architecture is analysed to create *path simulation code* which models the target-specific behaviour of the program (Figure 3, step 5).

Compiling the instrumented source code and the path simulation code for the simulation host yields a model of the program which determines its execution time and power consumption on the target processor. Using the markers that were added to the original source code during instrumentation, the path simulation code can approximate the path taken through the binary executable. This reconstruction of binary-level control flow is executed in addition to the functionality of the original source code during simulation on the simulation host and allows the dynamic selection of annotations. As the path reconstruction is based on the binary-level CFG, only feasible paths through the binary program are simulated. By simulating the transition between basic blocks, the path simulation can also consider structural differences between source code and binary code. So not every marker in the source code always results in the simulation of the respective binary-level basic blocks. Instead, the path simulation code can accumulate markers, for instance to model loop unrolling, or completely skip them if they do not match an actual path through the binary-level control flow graph.

*2) Custom Hardware:* To consider the challenges of custom hardware power-modelling we combine synthesis with cycle-accurate simulation at RT-level and a subsequent phase of basic block identification and power/timing annotation [8]. Based on the results of the synthesis, a characterisation of the RT data path and the corresponding controller is performed. Scheduling (determine order of operations), allocation (determine required number of RT components) and binding (assign operation to RT operator) phases performed during synthesis inhibit a back-annotation to the original source. Transformations and optimisations applied by the synthesis phases do not allow to directly relate data path and controller elements to statements of the original source code. In order to deal with the significant differences between the original source code and the resulting RTL description, no back-annotation is performed, but an augmented C/C++ version of the generated RTL model is generated automatically. This model is functionally equivalent to the initial input source, but in its internal structure it follows the generated RTL model, allowing an accurate estimation of the behaviour in terms of power and timing.

The characterisation and model generation flow is shown in Figure 4. Using a high-level synthesis tool (PowerOpt), the initial source code is transformed into a *control and data flow graph (CDFG)* (Figure 4, step 1). The CDFG serves as input for the scheduling, allocation, and binding phases (Figure 4, step 2). The result of these phases is two-fold. First, an RTL data path is generated that contains all operations and performs the computation. Second, a controller is generated that controls the data path. The controller itself is represented as an FSM, whose inputs are given by the data path. The outputs of the controller are used to enable registers and select the correct inputs of multiplexers belonging to the data path.

Based on the RTL data path, *hardware basic block (HBB)* identification and characterisation is performed (Figure 4, step 3). An HBB is a set of RT components from the data path that are jointly active. An HBB is defined by the combination of the actual state of the controller's FSM as well as the evaluation of the controller's conditions. The conditions themselves depend on the actual values obtained from the data path. In simplified terms, an HBB can be considered as the set of RT components that is required for providing the input to the registers that are enabled by the controller in the particular clock cycle. As a result of the functional simulation during synthesis all values for each RT component are known. Since an RT component might be used by several HBBs, the data patterns that belong to the particular HBB are identified. These patterns are then used for estimating the power consumption of the particular component [13]. Since the real patterns obtained from the simulation are used, data dependencies between individual RT components are considered implicitly. In order to support different supply voltages and frequencies as defined by the power-islands, switched capacitance instead of power dissipation is used to represent an RT component. This allows re-positioning of the HW module in another power-island without performing the characterisation again. The power value is then averaged for all activations of the component. Switched capacitance of an HBB is the sum of the average capacities switched by all RT components belonging to the HBB. Along with the average switched capacitance the number of clock cycles for each HBB are annotated.

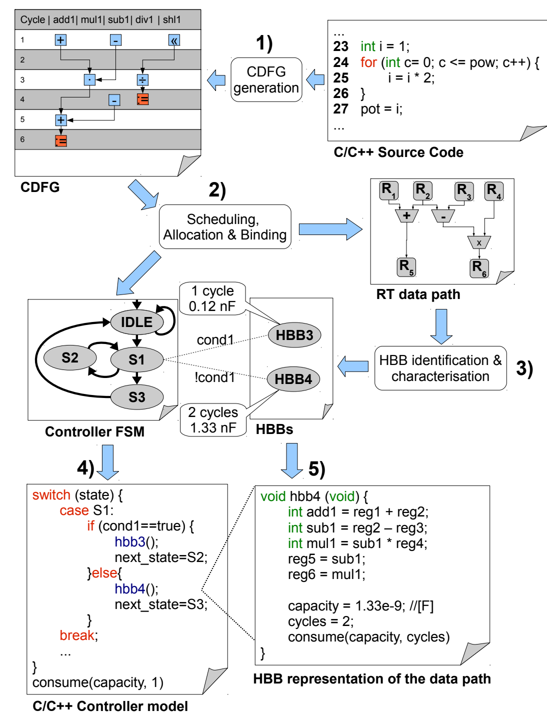


Fig. 4. Custom Hardware Annotation Flow

For virtual platform integration the controller model is transformed into executable C/C++ code by means of a switch-statement (Figure 4, step 4). This control structure activates, i. e. calls the particular HBBs and computes the next state of the FSM. The functionality of each HBB can be easily obtained from the data path (Figure 4, step 5), since enabled registers and values of all MUX-select signals are known for the particular HBB.

*3) Hardware IP:* Generic IP components delivered by third-party vendors cannot be estimated like custom HW and SW components. System-level simulation models of these IP's are typically provided as black-box executable models (e. g. API to a compiled object-file). These black-box modules usually contain timing but no power information. In order to obtain at least approximate power values a simple monitor is used which observes the component's input/output behaviour. Based on this observable activity a *Power State Machine (PSM)* [12] inside the monitor is triggered.

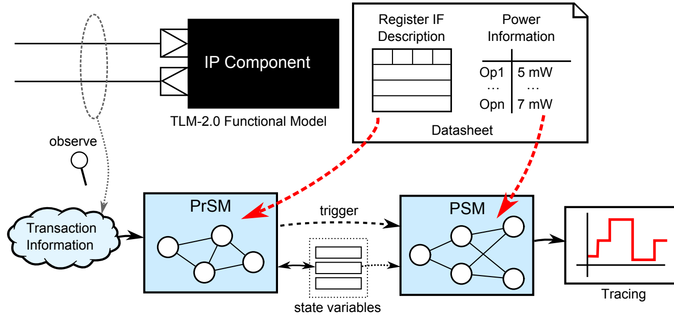


Fig. 5. Overview of Power-State Machine Model

A Power State Machine models the internal power states and possible transitions between them through observing an IP component's interaction with its environment. For our PSM approach (see Figure 5) we assumed that IP components use a SystemC TLM-2.0 compatible interface using the Generic Payload for communication with other system components. TLM-2.0 models bus-based communication with memory-mapped I/O, where transactions deliver data from initiator (master) to target (slave) socket and from the target back to the initiator socket. A transaction describes aspects like protocol phase, base address, data length and the transmitted data. To filter power state relevant information from the TLM-2.0 transactions and to trigger state transitions within the PSM, a protocol pre-processing automaton, called *Protocol State Machine (PrSM)* is used. The states of the PrSM can be derived from the bus protocol and the register interface description of the IP component.

PrSMs and PSMs are modelled as a combination of *Extended Finite State Machines (EFSM)* and *Timed Automata (TA)* to express time dependent power state transitions. The PrSM gets triggered by TLM-2.0 transactions and emits events to trigger the PSM. The PrSM's state space can be extended through state variables (EFSM). This extended state is shared by the PSM with read-only access. The PSM states are annotated with a timing invariant and static & dynamic power. The timing invariant specifies after which time the state has to be left, the dynamic power represents the average switched capacitance, and the static power the leakage resistance for this state of the IP component. The transitions between PSM states are triggered by PrSM events and can be guarded by a state variable (to model payload data dependencies on the internal power state) and a clock/timer (to model time dependent power transitions).

Power states of an IP component can be obtained from IP datasheet power information (if available), an estimation of the IP component's size and functional complexity (top-down approach), or from RTL or gate level power estimation (bottom-up approach) [11].

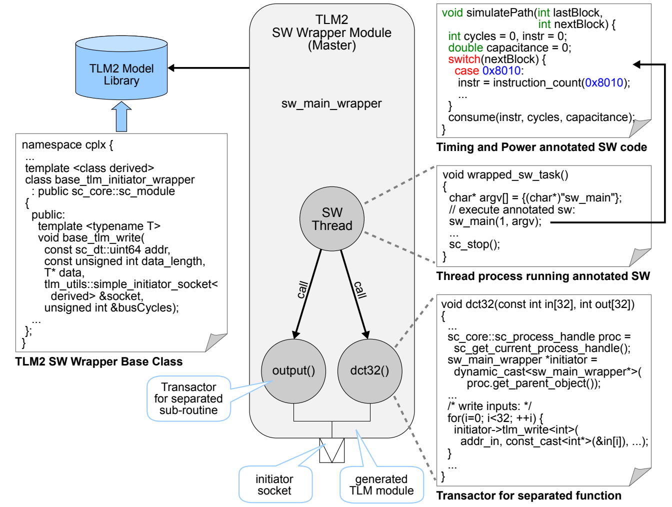


Fig. 6. TLM-2.0 Wrapper Module for SW

## E. Virtual System Generation

During generation of the virtual system, annotated sources from ⓔ and ⓕ as well as the selected models from ⓖ are combined to a virtual prototype (Figure 1). The annotated sources for the HW and SW components have to be wrapped in appropriate TLM-2.0 models. These TLM wrappers are generated based on an analysis of the source code from the input specification. The generated wrapper modules must account for different characteristics of the hardware and software.

**TLM wrapper for annotated software:** We assume that a single function represents the entry point for every piece of software, e.g. function *main*, and that software always represents a master within the system. The functionality of certain sub-routines might be provided by corresponding HW accelerators. Furthermore, (implicitly) accessed data, like stack variables, is assumed to be located in external memory. The generated TLM-2.0 SW wrapper module therefore provides an initiator socket that allows establishing the communication with subordinated service modules. Calls to sub-routines that have been separated out from the software need to be transformed into corresponding TLM-2.0 transactions. This requires replacing the sub-routines' software implementations by transactor functions. Likewise, all accesses to data located in external memory must be redirected to similar transactors, conducting TLM-2.0 transactions for every access. Figure 6 gives an overview on a TLM-2.0 SW wrapper module. As the software is supposed to run on some CPU core right from the start after system initialisation, the wrapper module just contains one thread process that simply executes the annotated software code and stops a simulation run as soon as the top-level software function returns. The transactor functions (*output* and *dct32* in Figure 6) are free functions that are called from within the annotated software which itself runs as part of the wrapper module's thread process. For convenience, a base class from a specialized library is utilized when generating a new wrapper module. This base class also provides methods

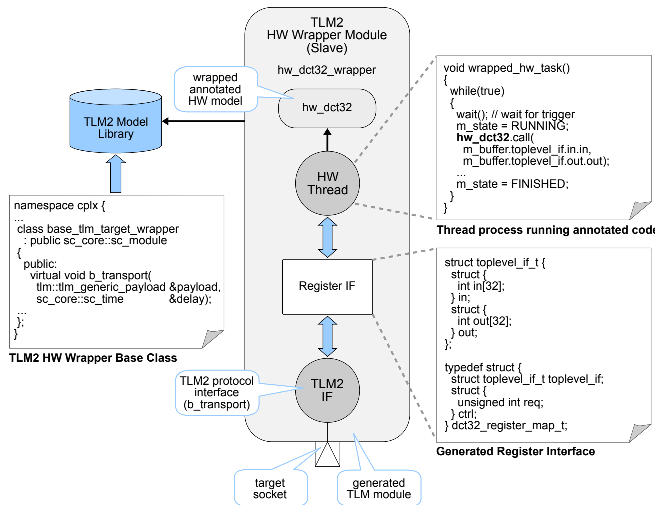


Fig. 7. TLM-2.0 Wrapper Module for HW

(*tlm_read* and *tlm_write*) that can be used to issue TLM-2.0 transactions from within a transactor function. Derivation from this base class reduces the size of the code that needs to be generated. For simplification, the current flow does not consider re-allocated sub-routines when generating the TLM wrapper for software. Instead, the transactor functions are generated together with the wrappers for separated HW and need to be integrated with the annotated software manually. In general this only requires to ignore those sub-routines during software annotation and to replace the original implementation with the generated transactor code. The automated task separation facilitates this kind of replacement by allocating every software routine to a separate source file.

**TLM wrapper for annotated hardware:** In contrast to software, models that represent hardware are assumed to represent slave components, providing functionality that is separated out from the original software implementation. Therefore, the generated TLM-2.0 HW wrapper module only provides a target socket as its functional interface. The TLM wrapper holds the annotated controller and data-path model described in III-D2 as its sub-component and provides a thread process that executes that model. As shown in Figure 7, the wrapper module also allocates a buffer for storing inputs and outputs of the wrapped hardware block. The input parameters that do arrive sequentially, each within a single TLM-2.0 transaction, are stored in this buffer until all inputs have been send by the initiator and the wrapped functionality can be executed. Analogously, the outputs are buffered as well and sent back to the initiator in single TLM-2.0 transactions. The layout of the buffer (size and address offsets) are derived from a struct type that is also generated during the task separation step. We use the TLM-2.0 base protocol with the blocking transport interface for communication. A generic *b_transport* method is defined in a base class which is available from a library. As in the SW case, the use of the base class reduces the amount of code that needs to be generated. However, the details of the wrapped hardware block, like input and output parameter numbers and types, can not be considered in the base class' generic *b_transport* implementation. Therefore, the generated wrapper needs to provide methods for reading and writing parameters. These methods are used as callbacks from the generic *b_transport* method.

**Integration:** The timing and power annotated execution-models for HW and SW are integrated with the remaining timing and power characterized platform elements. In the example platform model in Figure 2 these are: a TLM-2.0 router and a system memory model. For the MP3 decoder example design we take these models from [18], wrap them in TLM-2.0 modules manually, customize their timing behaviour, and extend them to monitor their power behaviour as described in Section III-D3. The router model is configured to provide the required number and type of sockets for connecting the platform elements: a pass-through target socket that is connected to the initiator socket of the SW wrapper module and three initiator sockets for passing TLM commands to the three TLM targets (HW, IP, and memory). As the design only contains a single master, no arbitration logic is needed inside the router.

### F. Simulation and Tracing

During virtual system execution the extra-functional model (Section III-C) aggregates and transforms the annotated cycles and average switched capacitances of tasks mapped to SW, HW, and IP components into traceable power information. Depending on the workload different execution paths of the functional model, leading to different extra-functional traces, are possible. After simulation, the collected information can be illustrated in a power-over-time diagram of the entire system or as a power-breakdown per task or platform component (see Figure 9).

Our annotations can be traced at different levels of granularity to allow a user-defined trade-off between simulation speed and accuracy. On the most abstract level tracing resolution is on *task closure granularity*. In this mode, extra-functional information is accumulated along the task execution sequence, also crossing component boundaries, including communication via service nodes. This tracing implicitly reschedules the system to a pure sequential execution, omitting possible partial order relationships expressed in the parallel application model. The next level of granularity is on *communication granularity*. Extra-functional properties are accumulated and traced when accessing ports of service nodes, i.e. when inter-task communication between platform components is performed. For a deeper analysis of the timing and power behaviour traces on *basic block granularity* of a CDFG for HW and SW is also possible.

Analysis of timing and power traces allows an evaluation of the chosen application mapping, the performance of the architecture and the effects of the synthesis constraints. Different design configurations or iterations with adjusted mapping, platform composition and constraints allow multi-objective design-space exploration.

## IV. EXPERIMENTAL RESULTS

Applying the proposed methodology to our example MP3-decoder design resulted in the virtual platform shown in Figure 8. The virtual platform model consists of five TLM-2.0 modules: a SW wrapper with annotated software implementing the main decoder functionality, a HW wrapper containing an

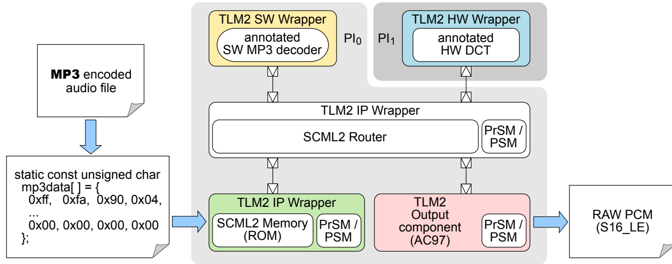


Fig. 8. MP3-decoder Virtual Platform

annotated model of a hardware DCT, a memory for storing the encoded input data, an AC97 related output component, and a router for managing bus transfers between the components. While the annotated SW MP3 decoder and HW DCT modules were generated from the input specification as described in Sections III-D1 and III-D2, the memory, router, and output modules represent IP models whose internals are partly taken from SCML2 [18] and whose timing and power behaviour is made observable by adding monitors as described in Section III-D3. The power annotations are based on average values obtained from characterisations of corresponding models using a 65nm technology. In accordance with the platform model depicted in Figure 2, two power islands do exist in the virtual platform.

The virtual prototype was used to compare different implementation variants. These variants include different operation sequences of SW decoder and HW DCT and different configurations of the power islands. The main goal of the exploration was to put our methodology to a test, surveying whether a power and timing estimation of the complete system is feasibly and whether it is capable to show the effects of different system configuration. In order to assess the benefit of moving the DCT to hardware, we also examined a platform where the DCT is implemented in software instead of custom hardware. Table I lists the different prototypes that have been analysed. The

TABLE I. PLATFORM VARIANTS

| ID | Description | Parameters | Simulation Factor[1] |
|---|---|---|---|
| $PM_1$ | SW only | $\mathcal{PI}_0$=(205MHz, 1.2V) | 137 |
| $PM_2$ | HW DCT, sequential | $\mathcal{PI}_0$=(200MHz, 1.2V) | 142 |
| | | $\mathcal{PI}_1$=(200MHz, 1.2V) | |
| $PM_3$ | HW DCT, parallel | $\mathcal{PI}_0$=(200MHz, 1.2V) | 163 |
| | | $\mathcal{PI}_1$=(115MHz, 1V) | |

power island configurations shown in Table I are sufficient to meet the real-time constraints of the system. We used about 1.1 seconds of audio data as input when simulating a platform variant. The simulation of every virtual platform model variant just took about three minutes, as shown by the simulation factor in Table I. Variant $PM_1$ that implements the DCT in software requires a frequency of 205MHz in order to fulfill the timing requirements imposed by the output component. The frequency can be slightly reduced in platform model $PM_2$ when the DCT is performed in hardware and executed sequentially. When parallelising the execution of software and hardware DCT ($PM_3$), the frequency of the power island containing the CPU can not be further reduced.

[1]defined as $\frac{\text{simulation time}}{\text{simulated time}}$

Table II shows the estimated timings and power consumptions for the different platforms, focusing on the PCM synthesis of a single frame. The PCM synthesis covers task $\mathcal{TN}6$, service node $\mathcal{SN}3$ and task $\mathcal{TN}7$. When synthesising a single frame, task $\mathcal{TN}6$ request the execution of task $\mathcal{TN}7$ via service node $\mathcal{SN}3$ for 72 times. The exploration shows

TABLE II. TIMING AND POWER FOR PCM SYNTHESIS OF A FRAME (2X36 SAMPLES, EACH WITH 32 SUB-BANDS)

| ID | Task | Time [ms] | Average Power [mW] | Error [%][2] |
|---|---|---|---|---|
| $PM_1$ | DCT (72x$\mathcal{TN}7$) | 1.16 | 9.51 | 3 |
| | rest ($\mathcal{TN}6$, $\mathcal{SN}3$) | 4.48 | 6.39 | 3 |
| $PM_2$ | DCT (72x$\mathcal{TN}7$) | 0.04 | 1.81 | 4 |
| | rest ($\mathcal{TN}6$, $\mathcal{SN}3$) | 5.35 | 6.49 | 3 |
| $PM_3$ | DCT (72x$\mathcal{TN}7$) | 0.07 | 0.72 | 4 |
| | rest ($\mathcal{TN}6$, $\mathcal{SN}3$) | 5.35 | 6.49 | 3 |

that the DCT executes much faster when mapped to a custom hardware block. The gained speed-up is minimal though, as the functionality of the PCM synthesis that remained in software consumes most of the execution time. Furthermore, bus transfers for communication with the HW DCT add up to the cost. The average power with respect to the execution time shows a noticeable reduction for the DCT task $\mathcal{TN}7$ when moving it from software to hardware. Parallelising the HW DCT allows to reduce frequency and voltage of its power island. Though this increases the time required to perform the DCT functionality, it further reduces its power consumption. Thus, platform $PM_3$ shows the best balance between performance and average power consumption.

Figure 9 shows power over time traces for the tasks and services involved in PCM synthesis. The traces have been retrieved from platform variant $PM_3$ and illustrate the extra-functional behaviour during the decoding of the first two frames. Due to the fact that it implements a rather small portion of the decoding algorithm, the custom hardware is idling most of the time. By contrast, the router also shows activity when data traffic occurs because the software reads from the memory or sends data to the output component. The CPU, which displays the highest power consumption, is even more active, as several parts of the decoding algorithm are solely performed by software and require no interaction with other components of the platform.

## V. CONCLUSION

In this paper we presented a framework allowing rapid virtual prototyping of heterogeneous embedded HW/SW systems under consideration of timing and power aspects. The presented flow considers custom hard- and software as well as third party IP components. For each of these different types we have used state-of-the-art research tools for estimation and source-level instrumentation of timing and power properties

[2]Absolute error of average power consumption compared to power-aware ISS simulation for the software ($\mathcal{TN}6$, $\mathcal{TN}7$ in $PM_1$) and gate-level simulation for hardware ($\mathcal{SN}3$ in $PM_1$, $PM_2$ and $PM_3$ & $\mathcal{TN}7$ in $PM_2$ and $PM_3$).

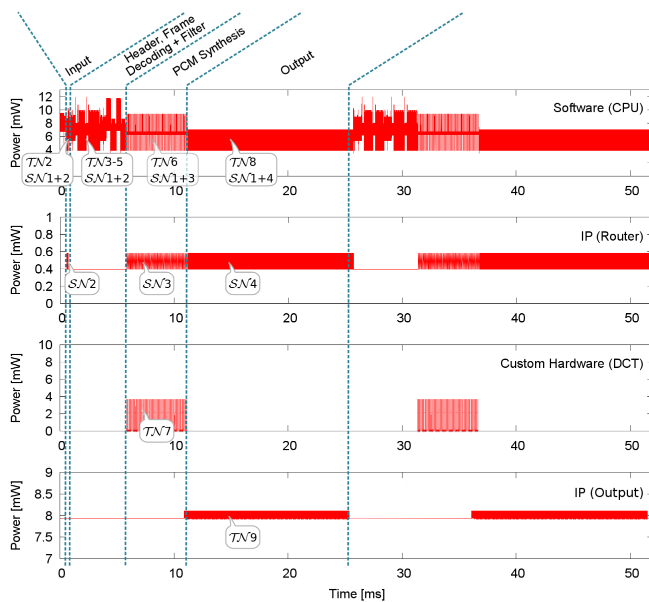


Fig. 9. Power over time traces obtained from Virtual Platform ($PM_3$)

for individual components. A generated SystemC TLM-2.0 prototype allows a fast and unitary simulation of the complete system, thereby helping the designer to identify and eliminate performance bottlenecks and power issues. Our framework for rapid virtual prototyping at ESL can be used for early trade-off between different design alternatives considering different platforms, mapping alternatives, and power configuration/management strategies.

In addition to improving the simulation performance of the presented techniques, our future work will address explicit support for software run-time systems and multiple concurrent applications. This allows the fast and accurate evaluation of extra-functional system properties at high levels of abstraction, while faithfully representing low-level effects induced by task scheduling, hardware interrupts and dynamic power management strategies, like clock and power gating of temporarily unused subsystems. With this, an early optimisation and functional validation of such dynamic effects under real workload scenarios will become feasible.

## Acknowledgments

This work has been partially supported by the EU integrated projects COMPLEX (FP7-247999) and CONTREX (FP7-611146), by the BMBF projects SANITAS (01M3088C) and RESCAR 2.0 (01M3195E).

## References


[1] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzer. http://www.absint.com/ait.

[2] N. Bansal, K. Lahiri, A. Raghunathan, and S. T. Chakradhar. Power monitors: A framework for system-level power estimation using heterogeneous power models. In *Proceedings of the 18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design*, VLSID '05, pages 579–585, Washington, DC, USA, 2005. IEEE Computer Society.

[3] M. Büker, K. Grüttner, P. A. Hartmann, and I. Stierand. Mapping of concurrent object-oriented models to extended real-time task networks. In *Proceedings of the 2010 Forum on specification & Design Languages (FDL'2010)*, pages 43–48, Sept. 2010.

[4] R. Damasevicius and V. Stuikys. Estimation of power consumption at behavioral modeling level using SystemC. *EURASIP Journal on Embedded Systems*, 2007, 2007.

[5] N. Dhanwada, I.-C. Lin, and V. Narayanan. A power estimation methodology for SystemC transaction level models. In *Proceedings of the 3rd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '05, pages 142–147. ACM, 2005.

[6] A. Donlin. Transaction Level Modeling: Flows and Use Models. In *Proceedings of the Intl. Conference on Hardware/Software Codesign and System Synthesis*, 2004.

[7] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, C. Ykman-Couvreur, D. Quaglia, F. Ferrero, and R. Valencia. The COMPLEX reference framework for HW/SW co-design and power management supporting platform-based design-space exploration. *Microprocessors and Microsystems*, 37(8,C):966–980, 2013. Special Issue on European Projects in Embedded System Design (EPESD'2012).

[8] K. Hylla. *Bridging the Gap Between Precise RT-Level Power/Timing Estimation and Fast High-Level Simulation*. PhD thesis, Carl von Ossietzky Universität Oldenburg, Dept. for Computer Science, 2014.

[9] IEEE Computer Society. IEEE Standard SystemC Language Reference Manual. IEEE Std 1666–2011, Sept. 2011. ISBN 978-0-7381-6802-9.

[10] H. Lebreton and P. Vivet. Power modeling in SystemC at transaction level, application to a DVFS architecture. In *IEEE Computer Society Annual Symposium on VLSI, ISVLSI'08*, pages 463–466, Apr. 2008.

[11] D. Lorenz, K. Grüttner, N. Bombieri, V. Guarnieri, and S. Bocchio. From RTL IP to functional system-level models with extra-functional properties. In *Proceedings of the 10th Intl. Conference on Hardware/-Software Codesign and System Synthesis (CODES+ISSS'2012)*, pages 547–556, Oct. 2012.

[12] D. Lorenz, P. A. Hartmann, K. Grüttner, and W. Nebel. Non-invasive power simulation at system-level with SystemC. In J. L. Ayala, D. Shang, and A. Yakovlev, editors, *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation*, volume 7606 of *Lecture Notes in Computer Science*, pages 21–31. Springer Berlin Heidelberg, 2013.

[13] W. Nebel and D. Helms. High-level power estimation and analysis. In C. Piguet, editor, *Low-Power Electronics Design*, chapter 38, pages 38–1 – 38–24. CRC Press, 2005.

[14] B. Sander, J. Schnerr, and O. Bringmann. ESL power analysis of embedded processors for temperature and reliability estimations. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, CODES+ISSS '09, pages 239–248. ACM, 2009.

[15] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Dominator homomorphism based code matching for source-level simulation of embedded software. In *Proceedings of the Seventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 305–314. ACM, 2011.

[16] S. Stattelmann, O. Bringmann, and W. Rosenstiel. Fast and accurate source-level simulation of software timing considering complex code optimizations. In *Proceedings of the 48th Design Automation Conference*, DAC '11, pages 486–491. ACM, 2011.

[17] M. Streubuhr, R. Rosales, R. Hasholzner, C. Haubelt, and J. Teich. ESL power and performance estimation for heterogeneous MPSoCS using SystemC. In *2011 Forum on Specification and Design Languages (FDL'11)*, pages 1–8, Sept. 2011.

[18] Synopsys, Inc. SCML Source Code Library. http://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi.

[19] Underbit Technologies, Inc. libmad - MPEG audio decoder library. http://www.underbit.com/products/mad.

[20] C. Walravens, Y. Vanderperren, and W. Dehaene. ActivaSC: a highly efficient and non-intrusive extension for activity-based analysis of SystemC models. In *Proceedings of the 46th Annual Design Automation Conference*, DAC'09, pages 172–177. ACM, 2009.

[21] L. Zhong, S. Ravi, A. Raghunathan, and N. Jha. RTL-Aware Cycle-Accurate Functional Power Estimation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(10):2103–2117, Oct. 2006.