Estimation of Non-Functional Properties for Embedded Hardware with Application to Image Processing

Christian Herglotz, Jürgen Seiler, André Kaup
Multimedia Communications and Signal Processing
Friedrich-Alexander University Erlangen-Nürnberg (FAU),
Cauerstr. 7, 91058 Erlangen, Germany
{christian.herglotz, juergen.seiler, andre.kaup}@fau.de

Arne Hendricks, Marc Reichenbach, Dietmar Fey
Chair of Computer Architecture
Friedrich-Alexander University Erlangen-Nürnberg (FAU),
Martensstr. 3, 91058 Erlangen, Germany
{arne.hendricks, marc.reichenbach, dietmar.fey}@cs.fau.de

Abstract—In recent years, due to a higher demand for portable devices, which provide restricted amounts of processing capacity and battery power, the need for energy and time efficient hard- and software solutions has increased. Preliminary estimations of time and energy consumption can thus be valuable to improve implementations and design decisions. To this end, this paper presents a method to estimate the time and energy consumption of a given software solution, without having to rely on the use of a traditional Cycle Accurate Simulator (CAS). Instead, we propose to utilize a combination of high-level functional simulation with a mechanistic extension to include non-functional properties: Instruction counts from virtual execution are multiplied with corresponding specific energies and times. By evaluating two common image processing algorithms on an FPGA-based CPU, where a mean relative estimation error of 3\% is achieved for cacheless systems, we show that this estimation tool can be a valuable aid in the development of embedded processor architectures. The tool allows the developer to reach well-suited design decisions regarding the optimal processor hardware configuration for a given algorithm at an early stage in the design process.

I. INTRODUCTION

Recently, the demand for portable devices being capable of performing highly complex image processing tasks has increased rapidly. Examples are cameras, smart-phones, and tablet PCs that can capture images and videos in real time. Furthermore, consumers like to process their data directly to enhance image quality, compress videos and pictures, or perform other picture manipulating tasks. Due to their highly complex nature, these tasks are time and energy consuming which reduces the operating time of the battery significantly.

Hence, it is desirable to develop energy efficient and fast running image processing software. For the developer, a major problem in the software design is that in order to obtain these non-functional properties of an application, complex and complicated test setups have to be available. E.g., to obtain the energy consumption of a solution, the code has to be compiled, executed on the target platform and measured using a power meter. Not till then it is possible to decide if a solution is energetically efficient.

To overcome the complex task of measurement we propose to perform a simulation on a virtual platform that allows to estimate the required energy and time for the target processing platform. This facilitates predictions on virtual platforms and allows a very precise estimation. As in this paper we do not model the cache system, we chose two image processing algorithms as an application. These algorithms (video decoding and signal extrapolation) show a highly linear processing order featuring a very high locality such that cache misses play a minor role during execution. The latter algorithm shows a highly homogeneous processing flow using repeatedly the same methods, where the former incorporates highly heterogeneous functions that are called in an unpredictable manner. Hence, we believe that they represent typical algorithms used on portable, battery constrained devices.

By using the open-access platform OVP (Open Virtual Platform) by Imperas it is possible to simulate the execution of such a process on any CPU of interest. During the simulation it is counted how often a certain instruction is executed. Multiplying these instruction counts with instruction specific energies and times we can estimate the complete processing time and energy of the written code on the desired target platform. In our approach, these specific energies and times are measured beforehand using a predefined set of specialized executable kernels. By evaluating the influence of an FPU on the chip area, processing time, and energy of the two image processing algorithms we show that our model can help the developer in choosing a suitable processing platform for his application. In this contribution, we show that this approach is valid for a cacheless, re-configurable, soft intellectual property (IP) CPU on an FPGA, where further work aims at generalizing this concept to allow the estimation of any CPU.

In this paper, we build upon the work presented in [4]. We augment this concept by introducing the FPU into the model, showing the viability of this approach for an extended set of test cases, and giving an example for a concrete application.

The paper is organized as follows: Section II presents an overview of existing approaches as well as a classification. Section III introduces the virtual platform we used to simulate the behavior of the processor. Subsequently, based on the simulation, the general model is presented for estimating processing time and energy in Section IV. Then, Section V explains the measurement setup and how the energies and times for a single instruction as well as for the complete image processing algorithms is determined. Finally, Section VI introduces two showcase image processing algorithms, evaluates the estimation accuracy for both these cases and shows how design decisions can be made.
Simulations are crucial tasks when developing hard- and software systems. Depending on the abstraction level, different simulation methods exist. A general goal is to simulate as abstract as possible (to enable a fast simulation) but to be as accurate as needed (to yield the desired properties). Therefore, we want to discuss several approaches of simulators for micro architectures and how to combine them to get very accurate results for non-functional properties like energy and time with a short simulation time.

The most exact results can be achieved by cycle-accurate simulators (CAS), such as simulations on hardware description level. These could be RTL (Register-Transfer-Level) simulations or gate level simulations, where a CAS is simulating each clock-cycle within a micro architecture. By counting glitches and transitions, very accurate power estimations can be achieved. Moreover, multiplying the simulated clock cycles with the clock frequency, the exact execution time can be determined. Unfortunately, CAS leads to very slow simulation speeds, due to the fact that they simulate the whole architecture. Typical examples include Mentor Modelsim, Synopsys VCS and Cadence NCsim.

One possibility to speed up the process is to use SystemC and TLM (Transaction-Layer-Modeling). By applying the method of \textit{approximately timed} models, as described in the TLM 2.0 standard, simulations run faster because clock cycles are combined to so-called phases. However, in contrast to the CAS simulations, the counted times and transitions are less exact. A simulator which follows this paradigm is the SoCLib library [8]. Moreover, frameworks such as Power-Sim have been developed to speed-up the simulation time by providing quasi cycle-accurate simulation.

On a higher abstraction level, simulation tools like Gem5 combined with external models like Orion or McPAT [5] can measure both time and energy to a certain extent, while at the same time sacrificing simulation performance: Complex applications like image processing can take up to several days until simulation is finished. Thus, if only functional properties such as correctness of algorithm, results or completion are of interest, an instruction accurate simulator is needed in order to speed up the simulation process.

Instruction set simulators (ISS) do not simulate the exact architecture, but rather “interpret” the binary executable and manage internal registers of the architecture. Typically, they require the least simulation time, but on the other hand do not include non-functional results such as processing time and energy consumption [14]. They are usually used to emulate systems which are not physically present, as debugger or as development platform for embedded software to get functional properties. OVPSim can be mentioned as an example for this simulator class.

The discussed approaches show that a very abstract description-level usually goes hand-in-hand with less information to be retrieved from simulation. One way to overcome this problem is executing on a very abstract level but compensating inaccurate results by applying a mathematical or statistical model in order to obtain relatively accurate estimations.

A typical example for this approach is presented by Carlson et al.: The Sniper simulator. According to [6], it is a simulator for x86 architectures such as Intel Nehalem and Intel Core2. For an interval model [7], where an interval is a series of instructions, possible cache and branch misses (due to interaction) are estimated. With the help of this information, homogeneous and heterogeneous desktop multi-core architectures are simulated. This is helpful when modeling rather complicated desktop architectures that include out-of-order execution, latency hiding and varying levels of parallelism [9]. While they are able to simulate multi program workloads and multi threaded applications, one drawback is the relative error which rises up to 25\%, which is acceptable for energy uncritical desktop applications and multi program simulation, but unacceptable when trying to find optimal design choices for energy aware embedded hardware. Furthermore, SniperSim is solely focused on Intel desktop architectures, which makes its use for general embedded hardware simulation unfeasible.

An illustration of the presented different simulation layers (including our extension of an ISS), ranged by their respective simulation speed, as well as the accuracy of the resulting non-functional parameter estimations, can be seen in Figure 1. Summarizing, we propose to estimate the processing time and energy using an extended ISS with only slightly increased simulation times. Therefore, we chose an existing tool that features a wide variety of embedded architectures, the OVP framework. Efforts have been undertaken to find modeling tools for CPUs modeled in OVP in the past [13], where the authors extended parts of a framework with pseudo-cycle accurate timing models. Using a watchdog component, they integrated an assembly parser and hash table with pre-characterized groups of instruction and timing information. The component then analyzes every instruction based upon a disassembly by OVP and assigns a suitable timing information. Unfortunately, simulation run-time is poor due to the external analysis of the instructions via disassembly. In regards of power modeling, Shafique et al. [17] proposed an adaptive management system for dynamically re-configurable processors that considers leakage and dynamic energy. Our approach is different as our emphasis is not on a low-level estimation including register-transfer level (RTL) techniques such as power-gating or instruction set building (often leading to a long simulation run-time), but rather on a fast high-level mechanistic simulation. Further work in regards of power and time consumption has also been done by Gruttner et al. in [11]. Their focus, however, lies on rapid virtual system prototyping.
of SoCs using C/C++ generated virtual executable prototypes utilizing code annotation.

In order to motivate our approach we first explain the fundamentals of a mechanistic simulation [2], which can be employed when using an ISS. Mechanistic simulation means simulating bottom up, starting from a basic understanding of an architecture. In this context, the source of this understanding is unimportant, thus it can also be achieved by an empirical training phase, which makes it suitable to proprietary IPs with limited knowledge of underlying details. This can be done by running measurements or experiments on actual hardware, in our case measuring processing energy and time of instructions. The resulting data is then prepared to be used in the simulation model - preparation can include regression and fitting of parameters. A mechanistic simulation is then run on a typical set of instructions using constant costs per instruction. In a very early work, this concept has been analyzed by Tiwari et al. [18] by measuring the current a processor consumes when a certain instruction is executed. Our approach is presented in the next section.

### III. Instruction Set Simulator: Virtual Platforms

As described in the section before, in this paper we want to combine an ISS with an additional model to get non-functional properties such as energy consumption and processing time. To achieve this, we discuss the ISS in detail and explain how it is modified for our purposes.

Open Virtual Platform (OVP) is a functional simulation environment which provides very fast simulations even for complex applications. Moreover, this flexible simulation environment is easy to use because once compiled applications can be run on both the real hardware and the OVP simulator without additional annotations to the program. The fast simulation run-times even for complex application is possible because OVP simulates instruction-accurate, not cycle-accurate processing. The user can debug the simulation and will know at any point, e.g., which content the registers and program counters have, but not the current state of the processor pipeline. Analyzing non-functional properties like energy consumption or execution run-time are therefore natively not possible [2].

To run a simulation using OVP two things are necessary, first a so-called platform model which denotes the current hardware platform, e.g., a CPU and a memory. Second, the application has to be provided as a binary executable file (the kernel). For a fast start in OVP, several processor and peripheral models are included, where some of them are open source (e.g., the OR1K processor model). These models are dynamically linked during run-time to the simulator called OVPsim. Today, there is a number of different manufacturers of embedded processors on the market, such as ARM or INTEL. Because they are Hard-IP (intellectual property), these processors have the disadvantage that they cannot be individualized to the needs of the hardware designer, e.g., grade of parallelism, cache size or cache replacement strategy. Fortunately, there are also some open source Soft-IP processors available, like the LEON3 processor [1]. This processor can be edited individually to the needs of the hardware designer. The LEON3 implements the SPARC V8 processor architecture, which is available as VHDL source code. The different configurations of the processor can be synthesized and tested on an FPGA.

For our work, we have chosen the LEON3 processor because of its configurability. It is easier to analyze energy and time if unnecessary components can be disabled to allow well defined measurement environment. Moreover, the availability as an open source soft IP core allows easy debugging. As previously there was no SPARC V8 processor model available, a new complete processor model was developed by us [3]. By that we reach the same level of flexibility and configuration possibilities on simulation as in the processor on real hardware. Therefore, a C API for implementing and extending own processor models provided by OVP was used.

The general simulation flow of a single instruction in our SPARC V8 processor model is visualized in Fig. 2. First, the decoder analyzes the 32-bit instruction for patterns and decides what kind of instruction it is. Then, the instruction receives an internal tag which is used for representation in the disassembler and morpher. The disassembler includes functions for simulation output if the user wishes to debug the simulated instructions. The morpher part of the processor model generates native code for the simulator to execute. These functions represent what the simulator should do. E.g., an arithmetic operation extracts the source registers, reads the values of these registers, executes the operation, and saves the result to the target register. Moreover, depending on the kind of arithmetic operation, the internal ALU state can be changed to implement further instructions like branches.

As writing a morpher function for every possible instruction is highly complex, instructions were grouped and morphing functions summarized. E.g., arithmetic instructions like add or sub and their variants (analyzing flags, setting flags) are one group. Because of different data manipulation, register-register and register-immediate instructions had their own group, e.g., arithmetic-register-register instructions and arithmetic-register-immediate instructions. Figure 3 shows a visualization of this grouping.

The methods described above were used to get a fully functional simulation environment. To enable the estimation of non-functional properties, the functional simulation has to be extended. On real hardware, not all instructions have the same data or control path in the processor, e.g., a floating point operation is much more complex, needs more cycles and therefore more run-time than a simple integer instruction. Thus, all instruction groups are further divided into categories like integer, floating point, jumps, etc. The internal counters for
these instruction categories are realized without using callback functions to ensure a high simulation speed. Instead, in every morpher function a counter is implemented that increases an internal temporal register after the corresponding instruction was executed by the simulator. For every category one internal register exists. After the full execution of the application, the simulator reads out these registers and presents the results.

In this context we want to mention that the non-functional behavior of an instruction is not necessarily constant. Especially for complex instruction set architectures (CISC) the context has a major impact on energy and time (instructions can take longer due to mispredicted branches, depending on where they are found in the context of the program). On the other hand, for reduced instruction set architectures (RISC), most instructions can be executed using fewer cycles compared to a CISC based system. Time and energy waste for, e.g., a flushed pipeline due to a mispredicted branch are consequently not as severe when compared to a CISC based processor. As our work is mainly focused on embedded hardware, which often implements RISC architectures such as the Sparc V8 based LEON3 (which does not even feature a pipeline), we argue that due to architectural properties it is valid to assume that an instruction shows a roughly constant non-functional behavior regardless of the context it is found in.

IV. ENERGY AND TIME MODELING

The general equations used to estimate the processing energy \( E \) and time \( T \) are given as

\[
\dot{E} = \sum_c e_c \cdot n_c \quad \text{and} \quad \dot{T} = \sum_c t_c \cdot n_c,
\]

where the index \( c \) represents the instruction category as introduced before, \( e \) and \( t \) the instruction specific energy and time, and \( n \) the instruction count.

For our model, nine instruction categories have been identified as summarized in Table I. The first six categories describe the energy consumption of the basic integer unit. The remaining three categories correspond to the FPU operations. The category “FPU Arithmetic” comprises the floating point add, subtract, and multiply operation.

The middle and the right column of Table I include the instruction specific energies and times that are assigned to the respective categories. The specific time \( t_c \) can be interpreted as the mean time required to execute one instruction of this category in our test setup. Likewise, the specific energy \( e_c \) describes the mean energy needed during the execution of one instruction. The values shown in the table have been derived by the measurement method explained in Section V.

Now if we know how often an instruction of a given category is executed during a process, we can multiply this number with the specific energy and time, add up the accumulated values for each category and obtain an estimation for the complete execution time and energy. These numbers \( n_c \) are called instruction counts and they are derived by the simulation in the ISS as presented above.

V. MEASUREMENT SETUP

In order to prove the viability of the presented model, we built a dedicated test setup for measuring the execution time and energy consumption of a SPARC LEON3 softcore processor \[\text{[1]}\] on an FPGA board. The FPGA board was a Terasic DE2-115 featuring an Altera Cyclone IV FPGA. The board was controlled using GRMON debugging tools \[\text{[10]}\] and the LEON3 was synthesized using Quartus, where the cache system and the MMU were disabled. Hence, in this publication, we consider a baseline CPU including an FPU. We utilized an FPGA because it offers great flexibility: The CPU can be customized according to our needs for highly versatile testing. We exploited this property to generate a useful platform for the step-by-step construction of an accurate and general RISC model.

For the measurement of the execution time of a process we used the \texttt{clock()}-function from the C++ standard library \texttt{time.h}. The measurement method for the energy consumption of the process is the same as already presented in \[\text{[4]}\].

To obtain the energy required to execute a single instruction we measured two kernels: A reference and a test kernel as indicated by Table I. The processing in both kernels features the same amount of baseline instructions, e.g., jumps for a loop. In contrast, the test kernel additionally contains a high amount of specific instructions that are not included in the reference kernel. Subtracting the processing energy and time of the reference kernel \( (E_{\text{ref}}, T_{\text{ref}}) \) from that of the test kernel \( (E_{\text{test}}, T_{\text{test}}) \), we obtain the time and energy required by the additional instructions. This value is then divided by the number of instruction executions \( n_{\text{test}} \), which is the product of the number of loop cycles with the number of instructions inside the loop. Thus, we obtain an instruction specific time and energy as

\[
e_c = \frac{E_{\text{test}} - E_{\text{ref}}}{n_{\text{test}}} \quad \text{and} \quad t_c = \frac{T_{\text{test}} - T_{\text{ref}}}{n_{\text{test}}},
\]

as the mean time required to execute one instruction of this category in our test setup. Likewise, the specific energy \( e_c \) describes the mean energy needed during the execution of one instruction. The values shown in the table have been derived by the measurement method explained in Section V.

Now if we know how often an instruction of a given category is executed during a process, we can multiply this number with the specific energy and time, add up the accumulated values for each category and obtain an estimation for the complete execution time and energy. These numbers \( n_c \) are called instruction counts and they are derived by the simulation in the ISS as presented above.

V. MEASUREMENT SETUP

In order to prove the viability of the presented model, we built a dedicated test setup for measuring the execution time and energy consumption of a SPARC LEON3 softcore processor \[\text{[1]}\] on an FPGA board. The FPGA board was a Terasic DE2-115 featuring an Altera Cyclone IV FPGA. The board was controlled using GRMON debugging tools \[\text{[10]}\] and the LEON3 was synthesized using Quartus, where the cache system and the MMU were disabled. Hence, in this publication, we consider a baseline CPU including an FPU. We utilized an FPGA because it offers great flexibility: The CPU can be customized according to our needs for highly versatile testing. We exploited this property to generate a useful platform for the step-by-step construction of an accurate and general RISC model.

For the measurement of the execution time of a process we used the \texttt{clock()}-function from the C++ standard library \texttt{time.h}. The measurement method for the energy consumption of the process is the same as already presented in \[\text{[4]}\].

To obtain the energy required to execute a single instruction we measured two kernels: A reference and a test kernel as indicated by Table I. The processing in both kernels features the same amount of baseline instructions, e.g., jumps for a loop. In contrast, the test kernel additionally contains a high amount of specific instructions that are not included in the reference kernel. Subtracting the processing energy and time of the reference kernel \( (E_{\text{ref}}, T_{\text{ref}}) \) from that of the test kernel \( (E_{\text{test}}, T_{\text{test}}) \), we obtain the time and energy required by the additional instructions. This value is then divided by the number of instruction executions \( n_{\text{test}} \), which is the product of the number of loop cycles with the number of instructions inside the loop. Thus, we obtain an instruction specific time and energy as

\[
e_c = \frac{E_{\text{test}} - E_{\text{ref}}}{n_{\text{test}}} \quad \text{and} \quad t_c = \frac{T_{\text{test}} - T_{\text{ref}}}{n_{\text{test}}},
\]

as the mean time required to execute one instruction of this category in our test setup. Likewise, the specific energy \( e_c \) describes the mean energy needed during the execution of one instruction. The values shown in the table have been derived by the measurement method explained in Section V.

Now if we know how often an instruction of a given category is executed during a process, we can multiply this number with the specific energy and time, add up the accumulated values for each category and obtain an estimation for the complete execution time and energy. These numbers \( n_c \) are called instruction counts and they are derived by the simulation in the ISS as presented above.

V. MEASUREMENT SETUP

In order to prove the viability of the presented model, we built a dedicated test setup for measuring the execution time and energy consumption of a SPARC LEON3 softcore processor \[\text{[1]}\] on an FPGA board. The FPGA board was a Terasic DE2-115 featuring an Altera Cyclone IV FPGA. The board was controlled using GRMON debugging tools \[\text{[10]}\] and the LEON3 was synthesized using Quartus, where the cache system and the MMU were disabled. Hence, in this publication, we consider a baseline CPU including an FPU. We utilized an FPGA because it offers great flexibility: The CPU can be customized according to our needs for highly versatile testing. We exploited this property to generate a useful platform for the step-by-step construction of an accurate and general RISC model.

For the measurement of the execution time of a process we used the \texttt{clock()}-function from the C++ standard library \texttt{time.h}. The measurement method for the energy consumption of the process is the same as already presented in \[\text{[4]}\].

To obtain the energy required to execute a single instruction we measured two kernels: A reference and a test kernel as indicated by Table I. The processing in both kernels features the same amount of baseline instructions, e.g., jumps for a loop. In contrast, the test kernel additionally contains a high amount of specific instructions that are not included in the reference kernel. Subtracting the processing energy and time of the reference kernel \( (E_{\text{ref}}, T_{\text{ref}}) \) from that of the test kernel \( (E_{\text{test}}, T_{\text{test}}) \), we obtain the time and energy required by the additional instructions. This value is then divided by the number of instruction executions \( n_{\text{test}} \), which is the product of the number of loop cycles with the number of instructions inside the loop. Thus, we obtain an instruction specific time and energy as

\[
e_c = \frac{E_{\text{test}} - E_{\text{ref}}}{n_{\text{test}}} \quad \text{and} \quad t_c = \frac{T_{\text{test}} - T_{\text{ref}}}{n_{\text{test}}},
\]
Due to the unrealistic programming flow of the reference and test kernel, these values may differ from the values observed in a real application. Hence, the values are checked for consistency and manually adapted, if necessary.

It should be noted that the energy for the execution of a certain instruction can be variable depending on the preceding and succeeding instruction, or the input data. To overcome this problem, we take the assumption that in real application, this variation is averaged to an approximately constant value when the corresponding instruction is executed multiple times in different contexts, which is supported by the results of our evaluation.

**VI. Evaluation**

In this section, we show that our model returns valid energy and time estimations for the given CPU by testing two conventional image processing algorithms: High-Efficiency Video Coding (HEVC) decoding that performs mainly integer arithmetics and Frequency Selective Extrapolation (FSE) which makes extensive use of floating point operations.

**A. HEVC Decoding**

To test the energy consumption of the HEVC decoder we used the HM-reference software [12] that was slightly modified to run bare-metal. Therefore, we included input and output streams directly into the kernel and cross-compiled it for the LEON3. While the HEVC can be fully implemented using pure integer arithmetics, the used software performs few floating point operations, e.g., for timing purposes. Furthermore, it uses a high variety of different algorithmic tools and methods like filtering operations and transformations. Due to predictive tools, a high amount of memory space is required.

To have a representative test set, we measured the decoding process of 36 different video bit streams. These bit streams were encoded with four different encoding configurations (intra, lowdelay, lowdelay_P, and randomaccess), three different visual qualities (quantization parameters 10, 32, and 45), and three different input raw sequences.

**B. Frequency Selective Extrapolation**

The second test of the model was carried out by computing the Frequency Selective Extrapolation (FSE) [15] algorithm on the device. FSE is an algorithm for reconstructing image signals which are not completely available, but rather contain regions where the original content is unknown. This may, e.g., happen in the case of transmission errors that have to be concealed or if an image contains distortions or undesired objects. For the extrapolation purpose, FSE iteratively generates a parametric model of the desired signal as a weighted superposition of complex-valued Fourier basis functions. As the model is defined for the available as well as for the unknown samples, one directly obtains an extension of the signal into these unknown regions.

For generating the model, the available samples are iteratively approximated at which in every iteration one basis function is selected by performing a weighted projection of the approximation residual on all basis functions. This process can also be carried out in the frequency-domain. In doing so, a Fast Fourier Transform is necessary. Due to the high amplitude range and the required accuracy, all operations need to be carried out with double precision accuracy. For a detailed discussion of FSE, especially the relationship between the spatial-domain implementation and the frequency-domain implementation and its influence on the required operations and run-time, please refer to [15, 16].

As a test set for the input data we chose 24 different pictures from the Kodak test image database where for each picture a different mask was defined. Hence, we obtained 24 kernels differing by the input image and mask.

**C. Experimental Results**

To see the benefit and the influence of an additional FPU in a CPU, for both algorithms we tested two cases: Processing with and without floating point operations (float and fixed, respectively). The latter case is achieved by compiling the kernel using the compiler flag -msoft-float which emulates floating point operations using integer arithmetics. The use of this compiler flag does not influence the precision of the process such that the output matches exactly the output of the kernel that was compiled without this flag.

We show the validity of the model by measuring the execution time and the energy consumption of the processes presented above and comparing them to the estimations returned by the proposed model. The bar diagram in Figure 4 shows the results for four different representative cases.

The dark blue bars depict the measured energies, the light blue bars the estimated energies (left axis). The yellow bars represent the measured times and the red bars the estimated times (right axis). We can see that all estimations are located close to their corresponding measured values.

To evaluate the performance of our algorithm, we calculated...


| TABLE III. MEAN ABSOLUTE ESTIMATION ERROR AND MAXIMUM ABSOLUTE ERROR OF OUR MODEL. |
|---------------------------------------------------------------|
| Mean absolute error $\bar{\varepsilon}$ | Energy | Time |
|--------------------------------------------|--------|------|
|                                             | 2.68%  | 2.72%|
| Maximum absolute error $\varepsilon_{\text{max}}$ | 6.32%  | 6.95%|

| TABLE IV. THE CHANGE OF NON-FUNCTIONAL PROPERTIES OF AN ALGORITHM WHEN INTRODUCING AN FPU INTO THE HARDWARE. |
|---------------------------------------------------------------|
| FSE       | HEVC Decoding |
| Energy consumption | $-92.6\%$ | $-42.88\%$ |
| Processing Time   | $-92.8\%$ | $-43.49\%$ |
| # logical elements | $+109\%$ | $+109\%$ |

The estimation errors for all $M$ tested kernels are given by

$$\varepsilon_m = \frac{\hat{E}_m - E_{\text{meas},m}}{E_{\text{meas},m}},$$

where $\hat{E}_m$ is the estimated energy from (1), $E_{\text{meas},m}$ is the measured energy, and $m$ is the kernel index. In the same way, the estimation error for time was derived. Table III shows two summarizing indicators for the evaluated kernels: first, the mean absolute estimation error $\bar{\varepsilon} = \frac{1}{M} \sum_{m=1}^{M} |\varepsilon_m|$ and second, the maximum absolute error $\varepsilon_{\text{max}} = \max_{m} |\varepsilon_m|$, $m = 1, ..., M$ for both energy and time. The maximum error is the highest error we observed for our evaluated kernel set. The small mean errors which are lower than 3% show that the estimations of our model can be used to approximate the real energy consumption and processing time.

D. Application of the Model

As a first basic application of this model the above presented information can be used to help the developer decide for a suitable architecture. If he, e.g., would like to know if it makes sense to include an FPU on his hardware, he can simulate the execution of his code with and without an FPU and obtain the information about processing time and consumed energy. The result of such a benchmark for our framework is shown in Table IV.

The values in the table are mean values over all tested kernels. The third row shows the increase of chip area needed for an FPU which can be obtained by the synthesisization of the processor. We can see that if we spend more chip area (about twice the size as indicated by the number of logical elements), we save more than twice the size as indicated by the number of logical elements), in the same way, the estimation error for time was derived. Table III shows two summarizing indicators for the evaluated kernels: first, the mean absolute estimation error $\bar{\varepsilon} = \frac{1}{M} \sum_{m=1}^{M} |\varepsilon_m|$ and second, the maximum absolute error $\varepsilon_{\text{max}} = \max_{m} |\varepsilon_m|$, $m = 1, ..., M$ for both energy and time. The maximum error is the highest error we observed for our evaluated kernel set. The small mean errors which are lower than 3% show that the estimations of our model can be used to approximate the real energy consumption and processing time.

VII. Conclusions

In this paper, we presented a method to accurately estimate non-functional properties of an algorithm using simulations on a virtual platform. The model is based on a mechanistic approach and reaches an average error of 2.68% for energy and 2.72% for time estimation. Furthermore, we have shown that the information can be used by developers for time and energy estimates. Further work aims at incorporating a model for the cache and multi-core processors and generalizing this concept to any CPU of interest. Additionally, we will evaluate the estimation accuracy of this model for further algorithms to show the general viability.

ACKNOWLEDGMENT

This work was financed by the Research Training Group 1773 “Heterogeneous Image Systems”, funded by the German Research Foundation (DFG).

REFERENCES

[1] Leon3 processor. [Online.] Available: http://gaisler.com/index.php/products/processors/leon3?task=view&id=13.

[2] B. Bailey. System level virtual prototyping becomes a reality with OVP donation from imperas. White Paper, June, 1, 2008.

[3] S. Berschneider. Modellbasierte hardwareentwicklung am beispiel eingebetteter prozessoren fur die optische messtechnik. Master’s thesis, Friedrich-Alexander-Universitat Erlangen-Nurnberg, Germany, Dec. 2012.

[4] S. Berschneider, C. Herglotz, M. Reichenbach, D. Fey, and A. Kaup. Estimating video decoding energies and processing times utilizing virtual hardware. In Proc. 3PMCES Workshop, Design, Automation & Test in Europe (DATE), 2014.

[5] Binkert and Beckmann. The gem5 simulator. ACM SIGARCH Computer Architecture News, 39(2):1–7, May 2011.

[6] T. E. Carlson, W. Heirman, and L. Eckhout. Sniper: Exploring the level of abstraction for scalable and accurate parallel multi-core simulations. In Proc. International Conference for High Performance Computing, Networking, Storage and Analysis. ACM, Nov 2011.

[7] L. Eckhout. Computer architecture performance evaluation methods. Synthesis Lectures on Computer Architecture, 5:1–145, 2010.

[8] K. Z. Elabidine and A. Greiner. An accurate power estimation method for MPSoC based on SystemC virtual prototyping. In Proc. 3PMCES Workshop, Design, Automation & Test in Europe (DATE), 2014.

[9] S. Eyerman, L. Eckhout, T. Karkhanis, and J. E. Smith. A mechanistic performance model for superscalar out-of-order processors. ACM Transactions on Computer Systems (TOCS), 27(2), May 2009.

[10] A. Gaisler. Gmon2 debug monitor. [Online.] Available: http://www.gaisler.com/index.php/products/debug-tools/gmon2.

[11] K. Gruttner, P. Hartmann, T. Fandrey, K. Hylla, D. Lorenz, S. Statkowla, B. Sander, O. Bringmann, W. Nebel, and W. Rosenstiel. An ESL timing and power estimation and simulation framework for heterogeneous SoCs. Proc. International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV), pages 181–190, 2014.

[12] ITU/ISO/IEC. HEVC Test Model HM-11.0. [Online.] Available: https://hevc.hhi.fraunhofer.de.

[13] F. Rosa, L. Ost, R. Reis, and G. Sasatelli. Instruction-driven timing CPU model for efficient embedded software development using OVP. In Proc. IEEE International Conference on Electronics, Circuits, and Systems (ICECS), 2011.

[14] D. Sanchez and C. Kozyrakis. Zsim: Fast and accurate microarchitectural simulation of thousand-core systems. Proc. 40th Annual International Symposium on Computer Architecture (ISCA), 2013.

[15] J. Seiler and A. Kaup. Complex-valued frequency selective extrapolation for fast image and video signal extrapolation. IEEE Signal Processing Letters, 17(11):949 – 952, November 2010.

[16] J. Seiler and A. Kaup. A fast algorithm for selective signal extrapolation with arbitrary basis functions. EURASIP Journal on Advances in Signal Processing, 2011:1–10, 2011.

[17] H. Shafigue, Bauer. Adaptive energy management for dynamically reconfigurable processors. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 33(1):50–63, January 2014.

[18] V. Tiwari, S. Malik, and A. Wolfe. Power analysis of embedded software: A first step towards software power minimization. IEEE Transactions on VLSI Systems, 2(4):437–445, Dec 1994.