# Towards Generation of a Programmable Power Management Unit at the Electronic System Level

David Lemma[1] Mehran Goli[2] Daniel Große[1,2] Rolf Drechsler[1,2]
[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{lemma, grosse}@informatik.uni-bremen.de {Mehran.Goli, Rolf.Drechsler}@dfki.de

***Abstract*—Power-awareness is now crucial in the design flow for *System-on-Chip* (SoC) development. The main objective of power-awareness is to implement a *Power Management Strategy* (PMS) for the SoC by generating a flexible, yet efficient *Power Management Unit* (PMU). As the cost of structural changes to a design increases in advanced stages of development, the PMU should be incorporated into the design as early as possible. At early stages, *Virtual Prototype* (VP) based design at the *Electronic System Level* (ESL) has become an industry accepted solution. However, existing methods focusing on generating a PMU at the ESL have several drawbacks, such as relying on designers' domain expertise, a low degree of automation and a lack of programmability.**

**This paper introduces a novel approach that automatically generates a programmable PMU for a given VP at the ESL without the need for prior knowledge about the VP's structure and behavior. Our approach consists of three main phases: activity pattern extraction, power-aware analysis, and PMU generation. The programmability feature of the generated PMU enables designers to support various target applications. The efficiency and flexibility of the proposed approach are evaluated by the power consumption reduction enabled by the PMU within a real-world VP-based SoC platform.**

## I. Introduction

With the ever-increasing complexity of the modern *System-on-Chips* (SoCs), non-functional design aspects such as power consumption has become a major bottleneck in the design process. Power-awareness has been shown as a promising solution to reduce the power consumption of a given system [1], [2]. However, the traditional way of applying power-aware strategy at the *Register Transfer Level* (RTL) is not feasible anymore for modern SoCs as once the RTL designs are implemented, power saving opportunities have already been considerably reduced [3]. Moreover, since both software and hardware significantly affect the overall power consumption, a system-level strategy is necessarily required which is not available at RTL.

*Virtual Prototype* (VP) design flow at the *Electronic System Level* (ESL) has become an established industry practice for early Hardware/Software co-design [4]. A VP is an abstract and executable model of the underlying SoC hardware components which is typically written in SystemC [5], [6], [7] using its *Transaction Level Modeling* (TLM) framework [8]. In comparison to the RTL designs, VPs are much earlier available and have significantly faster simulation speed. Considering the aforementioned benefits of VP-based SoCs, extending VPs to be power-aware to enable early power analysis is a very promising direction. However, SystemC-based VPs do not explicitly convey any information suitable for power-awareness.

This work was supported in part by the University of Bremen's graduate school SyDe funded by the German Excellence Initiative and by the German Academic Exchange Service (DAAD).

In order to address the concerns raised by the need for power-awareness, VPs need to be analyzed to derive the *Power Management Strategy* (PMS). The PMS refers to an intelligent way of saving the overall power by putting unused modules into low-power states (or cutting off the power by e.g. power gating techniques) and waking them up properly whenever they need to be activated. The PMS will then have to be implemented by the *Power Management Unit* (PMU), whose design should also be set at the ESL for the design flow to take advantage of the early stage benefits of system-level modeling. Moreover, the PMU must be able to be reprogrammed in case that designers decide to apply different workload scenarios or update the target software.

The workloads may possibly have rare corner cases that are not considered at the ESL, or need to be changed (e.g. to increase the performance of the overall system) at lower levels of abstraction. It is then necessary for the PMU to accommodate to those changes in the design, which means that the PMU needs to be programmable.

Although the recent advances in ESL power modeling and estimation provide designers with ways to evaluate alternative VPs, there are still several shortcomings in the existing solutions. Approaches [9], [10], [11], [12] that focus on *Design Space Exploration* (DSE) can deal with the analysis of alternative workloads and designs at the ESL, but they do not fully support the automatic generation of a programmable PMU from the implicit PMS of a design. Conversely, the approaches [13], [14] that focus on the generation of a PMU at the ESL rely on the domain expertise of the designers to obtain the PMS from the VP, thus not being automated.

In this paper, we propose an automated power-aware analysis approach that provides designers with a programmable PMU at the ESL. The proposed approach consists of three main phases which are: 1) analyzing a given VP to extract structural and behavioral patterns, 2) using the extracted patterns to produce PMU parameters, and 3) generating the programmable PMU based on the PMU parameters. The proposed approach is the *first of its kind* which generates automatically a programmable PMU at the ESL. Since the LEON3-based VP SoCRocket [15] is the only freely available VP with support power modeling and estimation, it is used to evaluate the effectiveness of the generated PMU based on various workload scenarios. The proposed approach is, however, not limited to a particular VP or power modeling technique and can be applied to a given VP without needing designers to have prior knowledge about the VP structure or behavior.

## II. Related Work

The challenge of generating a PMU at the ESL has been tackled in different ways during the past decade. The different approaches can be summarized in two main categories: *High Level Synthesis* (HLS) and *Design Space Exploration* (DSE) oriented approaches.

### A. HLS Oriented Approaches

The HLS oriented approaches [13], [14], [16] are based on synthesizing a PMU from an abstract ESL implementation of said PMU. Typically this is achieved by extending SystemC constructs to incorporate power concepts described in *Unified Power Format* (UPF) descriptions. The main goal of these approaches is to be able to generate a PMU and synthesize it to hardware. However, these approaches have low degree of automation and usually requires a large programming effort by designers to analyze and understand a given VP and to generate the PMU.

### B. DSE Oriented Approaches

The DSE oriented approaches [17], [9], [10] are based on methods to evaluate different versions of VPs and other abstract designs (even UML based designs) to be able to generate PMUs, whose efficiency can be analyzed. The main goal of these approaches is to be able to evaluate alternative PMU designs that are manually crafted. Unfortunately, these approaches do not provide a way to generate a programmable PMU at the ESL.

In addition to the aforementioned category, there is smaller third category of approaches intend to address the programmability of the PMU and automated analysis leading to generating PMU, but are not full solutions. One approach in this category is that of Wang et al. [18], who evaluate alternative PD partitioning schemes for PMU designs that were obtained through an Evolutionary Algorithm being applied to the analysis of a SoC. Other approaches in this small category, are that of: 1) Macko [19], which introduced automated analysis of VPs as the bases for programmability, but does not fully support the generation of a programmable PMU and 2) Lemma et al. [20], which also does automated analysis of VPs (based on [21]), but does not support the generation of a PMU.

## III. Preliminaries

This section aims at keeping this paper self-contained by providing brief introductions on the basic concepts required for the rest of paper.

### A. Power Estimation Model

The static and dynamic power consumption can be estimated at the ESL by assimilating physical measurements (such as voltage, current and capacitance that come from lower levels of abstraction) to elements of the TLM methodology. Regardless of the type of TLM module (i.e. an initiator, an interconnect or a target), the static and dynamic power can be estimated using the equations as follows:

$$p_{static} = p_{staticnorm} * (T_{end} - T_{start}) \quad (1)$$

$$p_{dynamic} = \frac{(e_{read} * n_{reads}) + (e_{writes} * n_{writes})}{T_{end} - T_{start}} \quad (2)$$

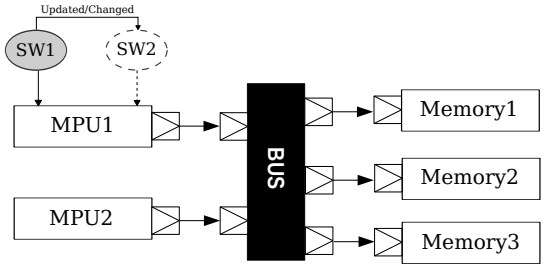


Fig. 1. The architecture of the motivating example

where:

- $p_{staticnorm}$ is a fixed value for normalized static power consumption that is technology dependent.
- $T_{end}$ and $T_{start}$ are obtained from simulation log files and measure the activity period of a given component.
- $e_{read}$, $e_{writes}$, $n_{reads}$ and $n_{writes}$ are the fixed value for normalized energy consumption per read and write instruction and the number of read and write operations respectively. These values are also technology dependent.

In this paper, the above parameters and other such normalized elements for power estimation are taken from the well-known SoCRocket platform, by considering *AHBIN*, *AHBCTRL* and *AHBMEM* as the SoCRocket components that implement the initiator, interconnector and target constructs, respectively. Estimation of power within the SocRocket platform is based on a set of back annotations from a 90 nm technology node [15].

### B. Power-Awareness Technique

Power-awareness is predicated on the fact that a SoC's power consumption has two main parts: static and dynamic. The static power consumption originates from the structure of a SoC's modules (how many of its constituent elements, such as gates, are connected to a voltage source at any given time). The dynamic power consumption originates from the way the SoC's components operate at any given time (how many of them are switching their constituent elements's operational states). Typically the most straightforward way to curb static power consumption is to power off unused parts of the design (power gating). Analogously, to curb dynamic power consumption, the most straightforward way is to disable clock signals for unused components while they are not under active use (clock gating). Power gating encompasses clock gating, so it is traditionally the most effective power management technique [22], [23]. Hence, the focus of this work is on the use of the generic concept power gating at the ESL.

## IV. Motivating Example

The motivation behind the present work can be explained in light of a VP-based SoC. Consider the *LT_BUS* VP in Fig. 1 implemented in SystemC TLM-2.0. It consists of two microprocessor units (*MPU1* and *MPU2*), an interconnect *LT_BUS* and three memory modules (*Memory1* to *Memory3*). The *MPU1* and *MPU2* modules serve as initiator, the interconnect module *BUS* works as a router while *Memory1* to *Memory3* are the target modules. The *MPU1* has a running software *SW1* and access to *Memory1* and *Memory3* by generating a set of transactions while the *MPU2* module accesses only to *Memory2* (for brevity its software and functionality are

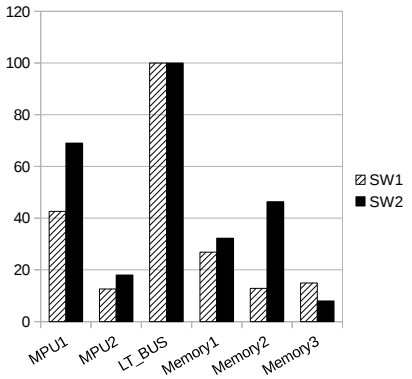


Fig. 2. Activity percentage of the *LT_BUS* VP modules for different workloads.

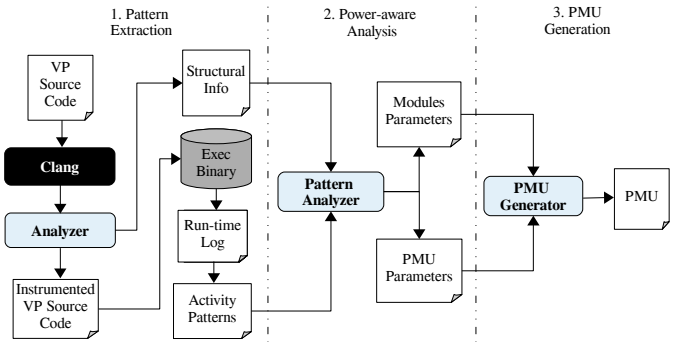


Fig. 3. Overview of the proposed approach.

abstracted away). The above case has an architecture that can benefit from a PMU to manage its power consumption. This PMU needs to take into account the activity patterns of the VP for its initiators, interconnect and target modules. The gray bar (*SW1*) in Fig. 2 shows the activity percentage of the VP modules when the *SW1* is running on the *MPU1*.

However, as an ESL prototype, a VP is typically refined at the ESL and the PMU once designed for it, needs to be revamped as well. A typical scenario at the ESL is that the running software on some modules of the SoC is modified (a behavioral change) or the architecture of the SoC is changed (a structural change). While changing on the hardware part (a structural change) of a given VP-based SoC is performed once at the ESL as a design decision (and typically stays constant), the software part can continuously have been changed or updated during the design process or even after the synthesize process.

As an example of changing/modifying the software part, consider the scenario that the *SW1* running on the *MPU1* is modified to *SW2* to increase the overall performance of the VP. This software change affects the behavior of the *MPU1* as the number of generated transactions, time of activation and access to the memories are changed. Thus, the changed behavioral patterns of the VP have a direct impact on the power management strategy and subsequently the PMU of the VP. The black bar (*SW2*) in Fig. 2 shows the activity percentage of the VP modules based on the new behavioral pattern of the VP after modifying the running software of the *MPU1* module. It shows that the activity percentages of modules in case of *SW2* are different than *SW1*, meaning a new power management strategy is required. Consequently, the parameters related to the PMU to realize the new power management strategy must be updated as well.

The generated PMU at the ESL must be able to accommodate a degree of programmability to support these changes that may happen not necessarily at the ESL but at lower levels of abstraction. This need for programmability is the main motivation for this work.

## V. Power-aware Approach

As illustrated in Fig. 3, the proposed approach consists of three phases:

1) extracting the both structural information and activity patterns from a given VP by analyzing its source code and logging its run-time behavior,
2) performing a power-aware analysis on the extracted information from the previous phase to obtain the required parameters to generate PMU, and
3) generating PMU based on the extracted parameters.

In the following, each phase of the proposed approach is explained in detail and illustrated using the motivating example LT_BUS VP from Fig. 1.

### A. Activity Pattern Extraction

The first phase of the proposed approach is to understand the structure and behavior of a given VP. This requires to access the static information describing the VP structure (i.e., the name of TLM modules, their sockets, signals and member functions) and the run-time behavior which is defined in terms of transaction. In order to extract the structural information and trace all transactions of the VP generated by different initiator modules to access the corresponding targets through the interconnect, we take advantage of the Clang [24] compiler (inspired by [25]). The information related to the VP's structure is extracted by visiting the reverent node in the *Abstract Syntax Tree* (AST) of the VP which is generated by Clang from its source code. To trace the VP's behavior, an instrumented version of the VP source code is generated by the *Analyzer* module (Fig. 3, phase 1). The instrumented version of the VP's source code is generated by inserting a set of *retrieving* statements to the VP's source code. The *retrieving* statements include the instructions that properly trace transactions. This information is the transaction's reference address, the value of transaction's attributes and its related parameters such as its timing annotation. The *retrieving* statements are inserted into two possible locations to trace all activity of modules

```
1  struct MPU1: sc_module {
2   tlm_utils::simple_target_socket<MPU1,32> socket;
3   void thread_process(){
4   ...
5   socket->b_transport(*trans, delay);
6   Fout<<" MPU1:thread_process:ID = ''<<trans<<" Cmd = "
        << trans->get_command()<<" address = '' << hex
        << trans->get_address() <<" at time '' <<
        sc_time_stamp()<<" delay = '' << delay << endl;
7   ...}
```

Fig. 4. Part of the Instrumented Source Code of Module *Target_A* of the *LT-AT_BUS* VP.

```
SQ1: ([MPU1, thread_process ,20 ns ,0 x6631b0 ],[0 x006 ,W,5 ns ])
SQ2: ([LT_BUS, b_transport ,25 ns ,0 x6631b0 ],[0 x003 ,W,5 ns ])
SQ3: ([Memory1 , b_transport ,30 ns ,0 x6631b0 ],[0 x003 ,W,5 ns ])
SQ4: ([LT_BUS, b_transport ,35 ns ,0 x6631b0 ],[0 x003 ,W,5 ns ])
SQ5: ([MPU1, thread_process ,40 ns ,0 x6631b0 ],[0 x006 ,W,5 ns ])
```

Fig. 5. A part of the *Run-time Log* of the *LT_BUS* VP.

which is related to any possible changes of their transactions. These locations are the line of code where the transaction is defined (e.g., as a function arguments), and the function call (e.g., transport interface b_transport) where the transaction object is used as an input argument. By executing the executable model of the generated source code, the behavior of the VP is extracted and stored in the *Run-time Log* file (Fig. 3, phase 1). A further analysis is performed on the *Run-time Log* file to present for each module of the design, a list that includes all simulation time when the module was activated during the execution time. The generated file is called *Activity Pattern* and shows the detailed activity of the entire VP's modules from the beginning until the end of the execution time.

For example, consider the *MPU1* module of the *LT_BUS* VP (Fig. 1). To trace all transactions related to this module, functions of the module (e.g., *thread_process*) in which a transaction object is referenced need to be traced. This is done by analyzing the VP's AST using the *Analyzer* module (Fig. 3, phase 1) to generate the *retrieving* statement and inserted into the source code. As an example, consider line 5, in Fig. 4 where the transaction object *trans* is used as a function argument of the *b_transport* interface. To trace the transaction and consequently the activity of *MPU1*, the *retrieving* statement *Fout* (line 6, Fig. 4) is automatically generated and inserted after the function call in the new source code. Fig. 5 shows a part of the generated *Run-time Log* of the *LT_BUS* VP which include an accurate trace of a transaction generated by *MPU1* to write a data in *Memory1* through the *LT_BUS* module. It also includes the delay and simulation time for each phase of transaction within its lifetime as well as this overall required time to complete the transaction which is *20 ns*. The corresponding activity pattern of the VP is illustrated in Fig. 6. This information is used in the next step to perform a power-aware analysis.

### B. Power-aware Analysis

In this phase, both the structural and activity patterns are analyzed to extract the required information to generate the PMU. The outputs of this phase are module and PMU parameters that are used as the inputs of the *PMU Generator* module in the PMU generation phase.

The structural patterns are analyzed to understand the modules parameters for the PMU generation. The structure of the modules is revealed by the module parameters, such as

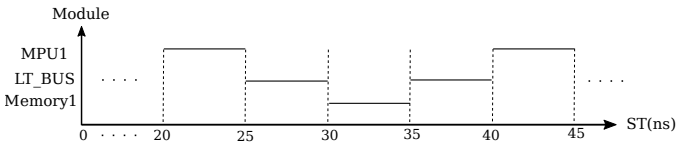


Fig. 6. A part of the activity pattern of the *LT_BUS* VP modules.

```
1  #define patternDataSize 6
2  #define moduleNum 6
3  #define memSize 1024
4  /* ... */
5  struct PMU: sc_module{
6   sc_out<bool> CS[moduleNum];
7   sc_bv<patternDataSize> memory[memSize];
8
9   void process ();
10  sc_bv<moduleNum> decode_CS (sc_bv<patternDataSize>);
11  int decode_time (sc_bv<patternDataSize>);
12  /* ... */
13  SC_CTOR(PMU){
14   SC_THREAD (process);}
15 };
16 /* ... */
17 void PMU::process () {
18  int memAddress = 0; //starting point
19  int delay; //last 10 bits
20  sc_bv<moduleNum> CStemp;
21  /* ... */
22  while(1) {
23   if (memAddress < memSize){
24    delay = decode_time (memory[memAddress]);
25    CStemp = decode_CS (memory[memAddress]);
26    for (int i = 0; i < moduleNum; ++i)
27     CS[i].write(CStemp[i]);
28    wait(delay);
29    memAddress++;}
30  /* ... */
31 }
```

Fig. 7. A part of the generated PMU of the *LT_BUS* VP.

their name, instance, number of sockets and class (initiator, target, interconnect). In order to control the power state of each module, a *Control Signal* (CS) needs to be inserted to each module of the VP. Thus, the structure information is used to know the number of CS (which is equal to the number of modules) and to subsequently insert it into each module of the design.

The activity patterns are analyzed to understand which modules of the design are active at each time unit (a period between two simulation time steps). The PMU parameters represent the way the PMU issues CS to drive different power modes (ON/OFF state) into modules of a given VP.

In order for the PMU to be programmable, the PMU takes advantage of an internal read-only memory. From the reading of this memory the PMU must identify the modules for whom it has to issue an activation/deactivation CS and the time unit for that CS (the duration of the ON/OFF state of the issued activation/deactivation). The memory size is related to the number of time-units (extracted by analyzing the activity pattern) and the the number of bits to code the activity pattern in each time unite which is called "pattern data". The pattern data consists of two main parts which are 1) the amount of bits that will encode the maximum difference between two time-units and 2) the amount of bits required to code each module of the design with an unique ID. The aforementioned information is encoded and stored in the memory of the PMU.

### C. PMU Generation

After extracting the required parameters from both the structural and activity patterns, the final phase is to generate the power management strategy in terms of a PMU module. This phase involves modeling the PMU following a simplified, but efficient and flexible structure. As illustrated in Fig. 3 (phase3), the *PMU Generator* module received the PMU parameters and automatically generates the PMU. The PMU is generated and

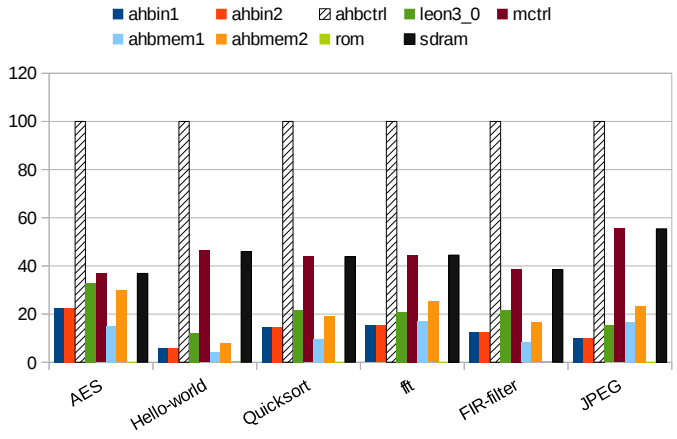


Fig. 8. Activity percentage of SoCRocket modules for different workloads.

implemented as a SystemC module and integrated with the VP. The structure of the PMU consists of a set of output signals (which is defined based on the number of modules in the VP), a process (which defines the behavior of the PMU) and a set of functions (which are used to decode the pattern data stored in a memory). The pattern data is loaded once in the memory (which is a read-only memory) at the beginning of the simulation.

Fig. 7 represents a part of the generated PMU related to the *LT_BUS* VP. The parameters *patternDataSize*, *moduleNum* and *memSize* (lines 1 to 3) shows the size of each memory line that keeps a pattern data, the number of modules in the VP and the size of the memory that includes all pattern data, respectively. As the maximum difference between two time-units is *5 ns* (obtained by analyzing the activity pattern, Fig. 6), three bits are defined by the *PMU Generator* module to code the difference between two time units. Moreover, three bits are used to code the modules of the VP (generating a unique ID for each module). Thus, the parameter *patternDataSize* is set to six. The *process* thread (lines 17 to 29) reads each line of memory, decodes the pattern data to obtain the duration of modules' activation *delay* (line 24) and the modules that must be activated and deactivated in this time unit *CStemp* (line 25). After decoding the pattern data, the controlling values stored in *CStemp* are assigned to the corresponding output controlling signals and subsequently are received by the corresponding modules. These controlling signals keep the value until the next address of memory is read and the same process is continued to end of the execution.

## VI. Experimental Results

This section presents results of applying our proposed approach to the real-world VP-based SoCRocket [15]. All analyses have been performed on a PC equipped with 8 GB RAM and an Intel core i7-2760QM CPU running at 2.4 GHz. The complete VP includes more than 50,000 lines of code. The VP combines several IP cores working together in *master* or *slave* mode. The VP is designed to be bus-centric, meaning the IP cores are connected through an on-chip bus. The AMBA-2.0 AHB (Advanced High-performance Bus) bus is used as the common on-chip bus. In the VP, the LEON3 processor is directly connected with AHB as AHBMaster device. A memory controller (mctrl) is connected as AHBSlave with AHB bus, which serves several memories. In order to support various workload scenarios we consider the case of integrating four TLM IPs with SoCRocket: two initiator processing elements *ahbin1* and *ahbin2* working as master mode and two memories *ahbmem1* and *ahbmem2* working in slave mode.

Table I illustrates results of applying the proposed approach to SoCRocket VP based on four workload scenarios and six different running softwares on the LEON3 processor. In order to cover different characteristics of power, corner-cases and validate the flexibility of our proposed approach, a combination of the following four scenarios (column *Scenario*) were used:

- **S1:** Light-weight access to one memory and intensive access to other.
- **S2:** Intensive access to all memories.
- **S3:** Light-weight workload on processor (i.e. LEON3 processor).
- **S4:** Intensive workload on processor (i.e. LEON3 processor).

The *ahbin1* and *ahbin2* were used to generate different number of transactions to create the S1 and S2 scenarios. We used various types of software (all implemented in C programming language) to create S3 and S4 scenarios. Columns *Software* and *LoC* show the name of each running software on the LEON3 processor and its complexity in term of lines of code, respectively. The column *#Trans* presents the number of generated transactions with respect to each workload scenario and were extracted in the first phase of the proposed approach.

Column *Power Consumption* demonstrates the power consumption of the SocRocket VP for each workload. In this regard, column *without-PMU* shows the VP power consumption when no PMU is integrated with the VP (e.g. the VP works in full power mode), while column *with-PMU* presents the VP power consumption when the generated PMU using the proposed approach is integrated with VP. Column *Difference* shows the percentage of power budget that is saved in each workload. The power consumption of the PMU is also included in the column *with-PMU*. The maximum power consumption of the PMU is about 5% which is for the case of *AES*. This shows that the generated PMU has very low power consumption overhead. The power consumption of the PMU is estimated based on the SocRocket parameters (Section III) and the PMU structure (Fig.7), including an internal read-only memory and the process which issues the controlling signals.

The percentage of power budget saved ranges from approximately **33%** to **64%**, which is a very meaningful reduction at the ESL and shows the advantage of using a strong power management strategy and subsequently the corresponding PMU module to realize it at the early stage of the design process. As we take advantage of the power gating technique to realize the power management strategy, the percentage of power budget saving is very related to the static power consumption of each module. Hence, if the static power in comparison is higher than the dynamic power consumption, a high power budget saving can be achieved. As shown in TableI and Fig. 8, for the scenario combination (column *Scenario*) of a light-weight workload on the LEON3 processor and access to memories (S1, S3), the power budget saving is minimum. The reason is that the required time for the processor to execute its running

TABLE I
EXPERIMENTAL RESULTS FOR ALL VARIANTS OF SOCROCKET VP WITH DIFFERENT WORKLOAD SCENARIOS

| VP Model* | Scenario | Software | #LoC | #Trans | Power Consumption (uW) | | | Execution Time (s) | | | | CET (s) |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | | without-PMU | with-PMU | Difference | Phase1 | Phase2 | Phase3 | Total | |
| SoCRocket | S1,S3 | Hello-world | 20 | 5,143 | 1026942 | 659337 | -35.79% | 39.2 | 3.1 | 1.2 | 43.5 | 2.1 |
| | S1,S4 | FIR-filter | 123 | 7,422 | 1280771 | 492147 | -61,57% | 41.4 | 4.5 | 1.3 | 47.2 | 6.3 |
| | S2,S3 | JPEG | 943 | 43,063 | 1359092 | 913069 | -32.81% | 42.2 | 4.8 | 1.3 | 48.3 | 3.3 |
| | S2,S4 | fft | 652 | 124,701 | 1866998 | 1011110 | -45.84% | 49.7 | 5.3 | 1.4 | 56.4 | 10.1 |
| | S2,S4 | Quicksort (QS) | 43 | 133,310 | 1824925 | 965922 | -47.07% | 54.5 | 6.1 | 1.5 | 62.1 | 11.2 |
| | S2,S4 | AES | 490 | 2,080,740 | 20822862 | 7518165 | -63.89% | 108.1 | 11.9 | 1.5 | 121.5 | 37.4 |

**LoC**: Lines of Code **#Trans**: number of extracted Transaction **CET**: The Compilation and Execution Time without any modification *Please note that the VP model is modified to support different types of scenarios.

software (*Hello-world*), as well as memories' activation time, are very short, thus the leakage (the main factor of static power consumption) is not noticeable. By increasing the software complexity (*FIR-filter*), the required time for the LEON3 processor to perform the task increases, meaning the leakage increases, thus the static power is dominant.

In the case of *JPEG* software running on the LEON3 processor (S2, S3), as the number of generated transactions to perform the task is higher than the case of *Hello-world*, the dynamic power consumption is dominant. Therefore, the power budget saving is less than the (S1, S3) scenario combination. In case that both the workload on the LEON3 processor and memories' access increase (the last three scenarios in the table), the execution time of the VP increases which leads to higher leakage and static power consumption. The maximum impact of the proposed power management strategy on the power budget saving can be observed in case the of *AES* software running on the LEON3 processor, where about 35 seconds was required for the task completion and overall two million transactions are generated.

In all of the scenarios, the approach introduces acceptable execution time (on average about a minute), showing its efficiency. The low time overhead makes the approach usable at the ESL, in which the proposed power-aware analysis is meant to be done as fast as its functionality oriented counterparts.

*Limitation:* Although initial results look promising, the proposed approach currently only supports the analysis of static workloads. This means that it is not yet possible to analyze workloads that have dynamic user dependent inputs.

## VII. CONCLUSION

This paper presented an automated power-aware analysis approach to generate a programmable PMU for the SystemC-based VPs at the ESL. The proposed approach enables designers to have a black-box analysis of a given VP, thus no prior knowledge about the VP is required. The first experiments with the LEON3-based SoCRocket VP demonstrate the applicability and effectiveness of the proposed approach. As a future work, we plan to extend our approach to support workloads that have dynamic user dependent inputs.

## REFERENCES


[1] O. S. Unsal and I. Koren, "System-level power-aware design techniques in real-time systems," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1055–1069, 2003.

[2] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, and et al., "The complex reference framework for HW/SW co-design and power management supporting platform-based design-space exploration," *Microprocessors and Microsystems*, vol. 37, no. 8, p. 966–980, 2013.

[3] B. Bailey. Power limits of EDA. [Online]. Available: http://semiengineering.com/power-limits-of-eda

[4] T. De Schutter, *Better Software. Faster!: Best Practices in Virtual Prototyping*. Synopsys Press, March 2014.

[5] Accellera Systems Initiative. OSCI SystemC 2.3. [Online]. Available: http://www.accellera.org/

[6] D. Große and R. Drechsler, *Quality-Driven SystemC Design*. Springer, 2010.

[7] M. Goli, J. Stoppe, and R. Drechsler, "Automated nonintrusive analysis of electronic system level designs," *TCAD*, vol. 39, no. 2, pp. 492–505, 2020.

[8] J. Aynsley, Ed., *OSCI TLM-2.0 Language Reference Manual*. Open SystemC Initiative (OSCI), 2009.

[9] H. Affes, A. B. Ameur, M. Auguin, F. Verdier, and C. Barnes, "An esl framework for low power architecture design space exploration," in *ASAP*, 2016, pp. 227–228.

[10] A. B. Mrad, M. Auguin, F. Verdier, and A. B. Ameur, "A framework for system level low power design space exploration," in *ICECS*, 2017, pp. 437–441.

[11] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Towards early validation of firmware-based power management using virtual prototypes: A constrained random approach," in *FDL*, 2017, pp. 1–8.

[12] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Maximizing power state cross coverage in firmware-based power management," in *ASP-DAC*, 2019, pp. 335–340.

[13] A. Qamar, F. B. Muslim, J. Iqbal, and L. Lavagno, "LP-HLS: Automatic power-intent generation for high-level synthesis based hardware implementation flow," *Microprocessors and Microsystems*, vol. 50, pp. 26–38, 2017.

[14] K. Jelemenska and D. Macko, "Adopting high-level synthesis approach to accelerate power management design," in *ICRITO*, 2018, pp. 124–130.

[15] T. Schuster, R. Meyer, R. Buchty, L. Fossati, and M. Berekovic, "SoCRocket - a virtual platform for the european space agency's SoC development," in *ReCoSoC*, 2014, pp. 1–7, http://github.com/socrocket.

[16] M. Škuta and D. Macko, "Automated integration of dynamic power management into FPGA-based design," in *DDECS*, 2019, pp. 1–4.

[17] H. Affes and M. Auguin, "SOC power management strategy based on global hardware functional state analysis," in *DSD*, 2015, pp. 614–620.

[18] B. Wang, Y. Xu, R. Hasholzner, C. Drewes, R. Rosales, S. Graf, J. Falk, M. Glaß, and J. Teich, "Exploration of power domain partitioning for application-specific SoCs in system-level design," in *MBMV*, 2016.

[19] D. Macko, "Contribution to automated generating of system power-management specification," in *DDECS*, 2018, pp. 27–32.

[20] D. Lemma, M. Goli, D. Große, and R. Drechsler, "Power intent from initial ESL prototypes: Extracting power management parameters," in *NORCAS*, 2018, pp. 1–6.

[21] M. Goli, J. Stoppe, and R. Drechsler, "AIBA: An automated intra-cycle behavioral analysis for SystemC-based design exploration," in *ICCD*, 2016, pp. 360–363.

[22] Y. Shin, J. Seomun, K.-M. Choi, and T. Sakurai, "Power gating: Circuits, design methodologies, and best practice for standard-cell VLSI designs," *TODAES*, vol. 15, no. 4, Oct. 2010.

[23] F. Schirrmeister, "Design for low-power at the electronic system level." ChipVision Design Systems, 2009, White Paper.

[24] The Clang Team, "Clang: a C language family frontend for LLVM," https://clang.llvm.org/.

[25] M. Goli, M. Hassan, D. Große, and R. Drechsler, "Automated analysis of virtual prototypes at electronic system level," in *GLSVLSI*, 2019, pp. 307–310.