Towards Early Validation of Firmware-Based Power Management using Virtual Prototypes: A Constrained Random Approach⋆

Vladimir Herdt[1] Hoang M. Le[1] 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
{vherdt,hle,grosse,drechsle}@informatik.uni-bremen.de

***Abstract*—Efficient power management is very important for modern System-on-Chip to satisfy the conflicting demands on high performance and low power consumption. Nowadays, global power management is mostly implemented in firmware (FW) due to the relative ease of development and its flexibility. Recent advances in system-level power modeling and estimation open up opportunities for early validation of these FW-based power management strategies. In this paper, we propose a novel approach for this purpose using SystemC-based Virtual Prototypes (VPs) and constrained random (CR) techniques. The CR-generated representative system workloads are executed in a power-aware FW/VP co-simulation to validate that available performance and power budgets are satisfied. As a proof-of-concept, we demonstrate our power validation approach on the LEON3-based SoCRocket VP.**

I. Introduction

Modern *System-on-Chips* (SoCs) must satisfy stringent requirements on power consumption and performance. With a continuously fast increase in number of implemented functionalities as well as in their complexity, meeting these requirements has become one of the major challenges in embedded system design. This new challenge demands a major shift in the design flow where power optimization/management is no longer an afterthought. There is an industry-wide consensus that waiting for the availability of RTL is not feasible anymore, because once the RTL is written, power saving opportunities have already been greatly cut off [1]. As both software (SW) and hardware (HW) have a significant impact on the overall power consumption, early design steps at the system level, in particular HW/SW co-design, should take power into consideration.

On the other hand, the emergence of *Virtual Prototypes* (VPs) at the abstraction of *Electronic System Level* (ESL) has played a major role in modernizing the SoC design and verification flow. In industrial practice, the C++-based modeling language SystemC and *Transaction Level Modeling* (TLM) techniques [2], [3] are being heavily used together to create VPs. The much earlier availability as well as the significantly faster simulation speed in comparison to RTL are among the main benefits of SystemC-based VPs. These enable functional validation and verification [4], [5] as well as SW development very early in the design flow. Building on this success story, extending VPs to be power-aware to enable early power analysis is a very promising direction. Admittedly, RTL is the first stage where enough details are present to provide reasonably accurate power numbers, however, ESL power modeling and estimation techniques are rapidly getting better (see e.g. [6], [7], [8]).

At the system level, the focus is not on low-level techniques such as power gating or dynamic voltage and frequency scaling but rather on fundamental design decisions that have a large impact on the power consumption, e.g. low-power architectures or *power management strategies*. The latter can contribute a great deal to the overall power saving by putting unused components into low-power states and waking them up properly in an intelligent manner. In most modern SoCs, the global power management strategy is implemented in firmware (FW) with the main advantages being the relative ease to develop and the flexibility in reconfiguring the strategy for different target applications. The recent advances in ESL power modeling and estimation enable to execute a particular SW application in FW/VP co-simulation and check whether its power budget and performance requirement are met. However, there is still a number of shortcomings with this basic approach. First, production-level SW is not yet available in early design stages. Second, simulating a full SW stack can still be very time-consuming, even at the speed of VPs. Third, a SW application is executed under some predetermined workloads (i.e. application and environment inputs). These workloads might very possibly miss rare corner cases where the power budget is exceeded or the performance constraint is violated.

To address these shortcomings, we propose a novel VP-based approach to assess the power-versus-performance trade-off of FW-based power management. Instead of executing real SW applications, our approach makes use of system-level workload scenarios. The main novelty of the approach is the modeling of workload scenarios based on *constrained random* (CR) techniques [9] that are very successful in the area of SoC/HW functional validation and verification. Each workload scenario corresponds to a system-level use-case with a specific power comsumption profile and is described by

⋆ This work was supported in part by the German Federal Ministry of Education and Research (BMBF) within the project CONFIRM under contract no. 16ES0565 and by the German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1 and by the University of Bremen's graduate school SyDe, funded by the German Excellence Initiative.

a set of constraints. The constraints define the set of legal concrete workloads that are conform to the intended use-case. The constraint-based description enables automated generation of a large number of different workloads within the scenario, hence reducing the risk of missing a corner case.

In this paper, we present the first attempt of realizing the proposed approach together with a proof-of-concept case study. As there is no freely available VP with power modeling and estimation, we build our case study around the open-source LEON3-based VP SoCRocket [10]. We extend the base VP with power management features and implement a FW-based dynamic power management strategy. Our approach is however not limited to a particular VP or power modeling technique. The obtained results demonstrate the potential of the approach and point out areas for further improvements.

The remainder of the paper is organized as follows. Section II reviews briefly related work. The proposed approach is outlined in Section III. The SoCRocket case study is then described in Section IV including the details of our power-aware extensions to the SocRocket VP. After the results are presented at the end of Section IV, Section V concludes the paper and discusses future work.

II. Related Work

ESL power estimation provides the basic technique for comparing different architectural/implementation options by providing power estimates using simulation. It has been intensely investigated in academia and industry.

In academia, approaches using cycle-accurate architectural simulators (e.g. [11]), power models per functional unit (e.g. [12], [13]) and ESL design extensions by power models (e.g. [7], [8]) have been proposed.

As a next step methods which allow for including power-management concepts (e.g. power states and power domains) have been developed. The PwARCH framework has been introduced in [14]. This framework follows the UPF principles and allows to add a power architecture to a SystemC TLM model such that different power design alternatives can be explored. A similar approach has been presented in [15] but it is based on metamodeling techniques.

A complete HW/SW co-design exploration methodology wrt. power has been introduced in [6]. The authors of [16] proposed an exploration approach targeting power domain partitioning at ESL. A design space exploration approach for power-efficient distributed embedded applications has been presented in [17].

Among the commercial tools for ESL power estimation are for instance Virtualizer from Synopsys or Vista from Mentor Graphics.

However, while these solutions (both academic and commercial) finally enable the comparison of power consumption for different design alternatives, they assume that appropriate workloads are already provided (mostly in form of some existing SW benchmarks). If the provided workloads are not representative enough, especially wrt. intended system-level use-cases, the comparison results might be misleading. The approach proposed in this paper targets this issue by providing means to specify abstract workload scenarios and enable automatic generation of concrete workloads.

III. Early Validation of FW-based Power Management Strategies

This section introduces the proposed approach for FW-based power management validation using constrained random techniques. At first, the overall workflow is described. Then, we present the specification principles for workload scenarios using constraints. Finally, the developed constrained random generator for these workload scenarios is introduced.

A. *Overall Workflow*

The overall workflow of the proposed approach is depicted in Fig. 1 and detailed in the following. The approach starts with a set of workload scenarios that have been formulated by the user (e.g. system architect or power validation engineer). The scenarios should have different characteristics of power consumption to ensure the thoroughness of the validation. Please note that it is possible to evaluate this thoroughness in an automated manner based on coverage metrics. This is, however, not in the scope of this paper, and is left for future work. Each scenario is described by workload constraints together with its power and performance budget. The workload constraints define the set of possible legal concrete workloads. Each scenario is furthermore associated with a number N – the minimum number of concrete workloads to be exercised in this scenario. The power and performance budget specification can be either absolute (i.e. absolute power consumption in μW or execution time in μs) or *relative*. Since the former is rather straight-forward, in the following, we focus only on the latter (relative) for a more compact representation. Also, in many cases, as the concrete workloads can be strongly varying, it might be not appropriate to specify an absolute budget. The relative budget is specified by percentages of the maximum possible for a concrete workload, e.g. performance within 70% of the maximum but power consumption not more than 50%. This maximum will be calculated by the approach as described below.

Our approach processes each scenario individually. Since the scenarios are independent, it is possible to distribute the computation over a cluster to speed-up the overall validation process. In the first step, a *Constrained Random Generator* (CRG) is instantiated. Then, for each scenario, the workload constraints are fed into the CRG, which is then instructed to solve the constraints and generate N different solutions. If the number of solutions is less than N, this is reported back to the user[1]. Each solution of the workload constraints is a concrete application workload for the considered scenario.

Then, for each concrete workload, our approach, in particular the Program Generator, generates two different programs. These programs are to be executed in a FW/VP co-simulation on the target VP. While they are equivalent from

[1]This step is omitted from Fig. 1 for the simplicity of representation.

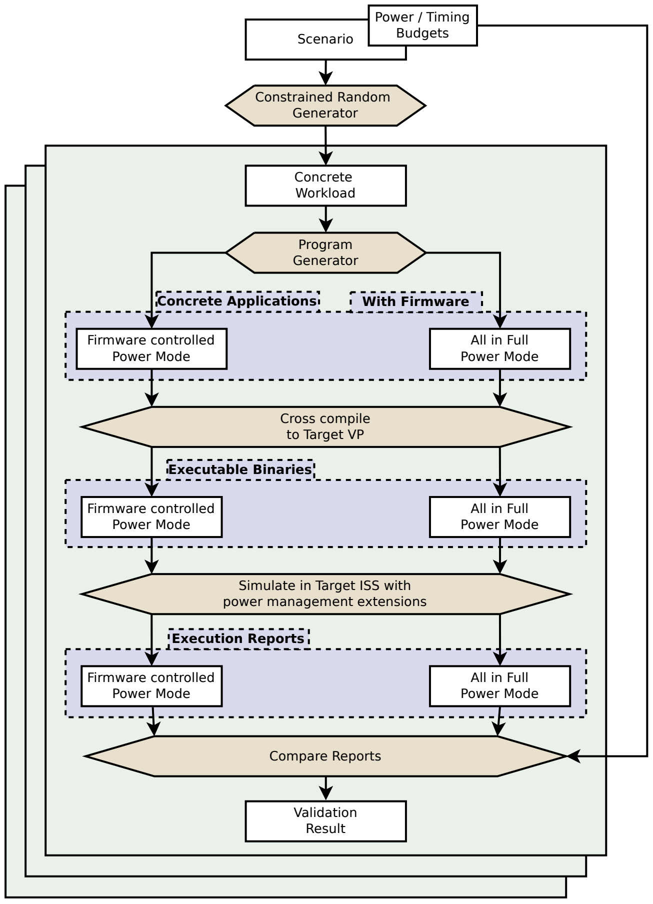


Fig. 1. Power management validation overview

the functional point of view, their power consumptions will be different: While the first program uses the FW-based power management strategy under validation, in the second program, all components of the VP are set to work in full-power mode (i.e. without FW-based power management). The attentive reader will already have deduced that the second program will be used to calculate the maximum power consumption and performance for the concrete workload. Subsequently both programs are cross-compiled to the instruction set of the target VP. The resulting binaries are loaded into the VP and executed. Under the assumption that the VP is *power-aware* and can generate detailed reports on power consumption and performance, these reports are inspected by our approach to validate whether the FW-based power management satisfies the specified (relative) power- and performance budget.

We now describe the two most important ingredients of the approach: how workload constraints for a scenario are specified (Section III-B) and how concrete workloads are generated (Section III-C). Then, we continue with the case study in Section IV.

B. Constraint-based Workload Scenarios

Before dealing with the workload constraints for a scenario, let us focus on how a concrete application workload is modeled. A workload is viewed as an abstraction of an execution of a SW application and contains a list of *instruction blocks* (IBs). Currently, our approach supports three types of IBs: arithmetic, memory and IO-device. Besides the basic common fields, e.g. block type *IB.type* and position in the list *IB.pos*, every instruction block has specific options, for example, with the block type is arithmetic, the options *num_instr* (i.e. number of instructions in block) and *op_type* (i.e. type of operation) are available. These options describe how many instructions are executed and what operation, e.g. integer addition or multiplication is used, respectively.

A scenario is then a symbolic description of a family of concrete workloads. The constraints describe the relationships between the instruction blocks and their specific options. Currently, we provide the following primitive functions to formulate these relationships: 1) Exists, 2) Ensure, 3) Assert, 4) Size and 5) Select. Formally a solution, i.e. list of NB instruction blocks, is valid if it satisfies the conjunction of all Exists, Ensure and Assert constraints. With the list of instruction blocks denoted as LIB, the primitive constraints are defined as follows:

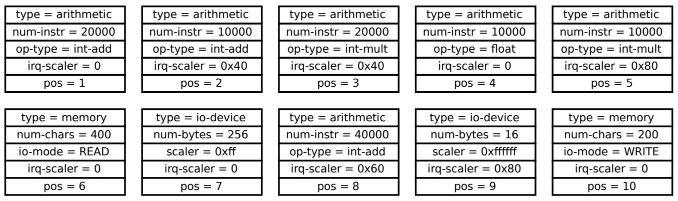
- *Exists(pred)* : $\exists b \in LIB : pred(b)$. The Exists constraint accepts an IB predicate *pred*. It is satisfied if such an IB satisfying *pred* exists in the list.
- *Ensure(sel, pred)* : $\forall b \in LIB : sel(b) \implies pred(b)$. The Ensure constraint accepts an IB selection predicate *sel* and a further IB predicate *pred*. It is satisfied if every IB that satisfies *sel* (i.e. selected IB) also satisfies *pred*.
- *Ensure(sel1, sel2, pred)* : $\forall b_1, b_2 \in LIB$ with $b_1 \neq b_2$: $sel1(b_1) \wedge sel2(b_2) \implies pred(b_1, b_2)$. This extended form of Ensure has the same semantics as the simple form, but works for a pair of IBs instead of one single IB.
- *Assert(expr)*. The Assert constraint expects a Boolean expression as argument and is satisfied if the expression is valid.
- *Size(pred)*: $x = |\{b \in LIB | pred(b)\}|$. The *Size* function returns a new symbolic variable x that represents the number of IBs that satisfies the predicate *pred*. The result of the *Size* function can be used to build larger predicates, which can then be passed to either *Exists*, *Ensure* or *Assert* constraints.
- *Select(pred)*. This helper function provide a way to define named predicates that can be reused in other constraints.

In addition, *Exists* can also be assigned to a named predicate allowing more succinct constraint specification. The predicates are mainly defined using the *lambda* notation, e.g. $lambda\ x\ :\ x.type\ ==\ arithmetic$. is satisfied by any arithmetic instruction block.

Example Constraints: An example for a constraint-based workload scenario is shown in Fig. 2. The example describes an abstract (symbolic) application workload that start with CPU-intensive code followed by mixed instructions. The first two lines specify that the initial five instruction blocks have arithmetic type. The next two lines require two of these five arithmetic instruction blocks to be executed with high interrupt frequency from peripherals and IO devices. Line 7 and 8

```
 1 // initial five instruction blocks have arithmetic type
 2 A = Select(lambda x: x.pos <= 5)
 3 Ensure(A, lambda x: x.type == InstrType.Arithmetic)
 4
 5 // two of the blocks from A have a small irq scaler
      (frequency at which interrupts arrive) - please
      note numbers starting with '0x' are in
      hexadeximal format
 6 B = Select(A, lambda x: x.irq.scaler != 0 &&
      x.irq.scaler <= 0x50)
 7 Assert(Size(B) == 2)
 8
 9 // the position of DeviceIO blocks is an odd number
10 C = Select(lambda x: x.type == InstrType.DeviceIO)
11 Ensure(C, lambda x: x.pos & 1)
12
13 // at least to memory intensive instruction blocks are
      available
14 D = Select(lambda x: x.type == InstrType.Memory)
15 Assert(Size(D) >= 2)
16
17 // at least on block has arithmetic type and more than
      10000 instructions
18 Exists(lambda x: x.type == InstrType.Arithmetic &&
      x.arithmetic.num_instr > 10000)
19
20 // there shall be an IO-device access with fast
      processing and another with slow processing of
      incoming/outgoing data (device.scaler denotes the
      processing speed) and the fast block appears
      before the slow one (specified by last constraint)
21 E = Exists(lambda x: x.type == InstrType.DeviceIO &&
      x.device.scaler < 0xfff)
22 F = Exists(lambda x: x.type == InstrType.DeviceIO &&
      x.device.scaler > 0x7ffff)
23 Ensure(E, F, lambda a,b: a.pos < b.pos)
```

Fig. 2. A constraint-based workload scenario



type = arithmetic	type = arithmetic	type = arithmetic	type = arithmetic	type = arithmetic
num-instr = 20000	num-instr = 10000	num-instr = 20000	num-instr = 10000	num-instr = 10000
op-type = int-add	op-type = int-add	op-type = int-mult	op-type = float	op-type = int-mult
irq-scaler = 0	irq-scaler = 0x40	irq-scaler = 0x40	irq-scaler = 0	irq-scaler = 0x80
pos = 1	pos = 2	pos = 3	pos = 4	pos = 5

type = memory	type = io-device	type = arithmetic	type = io-device	type = memory
num-chars = 400	num-bytes = 256	num-instr = 40000	num-bytes = 16	num-chars = 200
io-mode = READ	scaler = 0xff	op-type = int-add	scaler = 0xffffff	io-mode = WRITE
irq-scaler = 0	irq-scaler = 0	irq-scaler = 0x60	irq-scaler = 0x80	irq-scaler = 0
pos = 6	pos = 7	pos = 8	pos = 9	pos = 10

Fig. 3. Application workload (constraint solution)

specify that IO devices are not accessed immediately one after another, there is always processing time in-between. The next two lines require that at least two memory intensive instruction blocks are present in the list. Line 13 requires that at least one arithmetic block exists with more than 10000 instructions. The last three lines ensure some specific instruction blocks are present. In particular, there shall be an IO-device access with fast processing and another with slow processing of incoming/outgoing data. The last constraint determines the order of these IO accesses: fast before slow.

A solution to these workload constraints with 10 instruction blocks, i.e. a concrete workload of the example scenario is shown in Fig. 3. It satisfies all workload constraints, i.e. Assert, Exists and Ensure constraints. For example, the first five blocks are of arithmetic type. Other options that have not been constrained such as the type of arithmetic operation *op-type* (e.g. integer multiplication or floating point operation) are randomly generated. The generated list of instruction blocks allows the Program Generator to build a concrete application by randomizing the instructions within a block according to the block properties. The concrete application can then be simulated on the target VP.

C. Constrained Random Generator

The constraint language described above is implemented as a Domain Specific Language in Python (version 3). For a fully integrated SystemC-based flow, it would be better to build the language on top of a CRG framework for SystemC/C++ such as CRAVE [18]. A further advantage is that one could benefit from sophisticated CRG algorithms already implemented in such framework (see e.g. [19]). However, this would require substantial more implementation efforts. Furthermore, state-of-the-art CRG for SystemC/C++ only support constraints on bit-vectors, while our workload constraints can be formulated more naturally and efficiently on integers (not to be confused with their representation as bit-vectors). Therefore, for rapid prototyping and exploring our ideas, we decided to use Python at this stage and leave the option of a SystemC/C++ implementation for the final stage. The CRG as well as the overall flow is implemented completey in Python3. For the generation of concrete workloads of a scenario, the CRG starts with a predetermined number NB of symbolic instruction blocks. Then, it initializes NB symbolic blocks and maps the specified constraints for these blocks to the SMT fragment QF_LIA (i.e. quantifier-free linear integer arithmetic). In the next step, the CRG employs the state-of-the-art QF_LIA solver Z3 (v4.5.0) to solve the resulting SMT formula and generate N different solutions. Note that Z3 can by-default generate only one solution, our CRG contains an all-solution-solving layer over Z3 that adds additional constraints after the generation of a solution to block this solution from being considered in the future. If less than N different solutions can be found, the generator will increase NB to generate more solutions until the number N is reached.

In the next section we demonstrate the proposed approach for a LEON3-based VP.

IV. SoCRocket Case Study

This section presents a case study where we apply the proposed approach to the open-source VP SoCRocket [10]. Since there is no freely available VP with power modeling and estimation and SoCRocket is open-source, we first extend the base VP with power management features (see Section IV-A) and implement a FW-based dynamic power management strategy (see Section IV-B). Then, in Section IV-C we demonstrate our approach by providing an extensive validation of the power-management strategy.

A. Power Management Extensions

SoCRocket already includes basic power models. There are three types of power consumption values for each component: 1) static power, 2) internal power, 3) switching power. Both static and internal power can be considered application independent, thus their value only depends on the simulation

```
1 virtual double get_sta_power(PM_STATE s) {
2   std::map<PM_STATE, double>::const_iterator it =
        int_power_coefficients.find(s);
3
4   if (it == int_power_coefficients.end())
5     throw std::runtime_error("Unknown power state
          (get_sta_power)");
6
7   return it->second * pm->int_power;
8 }
```

Fig. 4. Retrieve the static power of a component based on its current power state

```
virtual double get_and_reset_swi_power(PM_STATE s) {
  std::map<PM_STATE, double>::const_iterator it =
      swi_power_coefficients.find(s);

  if (it == swi_power_coefficients.end())
    throw std::runtime_error("Unknown power state
        (get_swi_power)");

  double ans = it->second * pm->swi_power +
      pending_state_change_power;
  pm->reset_swi_power();
  pending_state_change_power = 0;
  return ans;
}
```

Fig. 5. Retrieve and reset the switching power of a component based on its current power state

time. Switching power will increase when the component is actively working, e.g. it depends on the number of executed instructions of the CPU and the number of bytes accessed by the memory and so on. Every component in SoCRocket possesses these power information, see [20] for more information. Adding up those power values for all components allows to compute the total power consumption at every simulation time step. We extend this basic scheme to support multiple power states and discuss how this information can be tracked. Furthermore, we add a lightweight *Power Interface Unit* (PIU) connected to the AHB bus of the system to act as a power interface for the firmware.

1) Power Modeling: For power modeling we add a power layer for every component. This layer stores the power states the component supports together with the currently active state and component specific delays due to power state changes. For example the CPU supports the full power mode (RTM), some power save modes (PS0, PS1, PS2) where it is still able to execute instructions, and sleep modes (DS0, DS1, DS2) where the CPU only waits for interrupts. In general every component supports some power save and sleep states in addition to the obligatory full power mode. For every power state the CPU power layer specifies how many extra cycles are added during instruction processing compared to the base value provided by SoCRocket. The memory power layer specifies how many extra cycles are necessary due to power save modes to process read/write instructions and so on. Similarly the base power consumption values for static, internal and switching power (which are provided in SoCRocket already) are modified based on the active power state of the components. As example the static power retrieval is shown in Fig. 4. Based on the current state s of the component pm a scaling factor is retrieved from a lookup table (Line 2) and applied to the base static power of the component (Line 7).

2) Power Tracking: For power tracking, every component is registered in the power monitor before simulation. Tracking static and internal power is straightforward because it is application independent. Therefore, at registration the static and internal power for every supported power state of every component is retrieved as shown in Fig. 4 and dumped to a log file. These power values describe how many static and respectively internal power is consumed by the component per second in each power state. Switching power depends on the application code and therefore is periodically read and reset for every component, as shown in Fig. 5. In order to compute the total power at each time step, the power monitor also dumps all power state changes of every component together with a simulation timestamp.

3) Power Interface Unit: The Power Interface Unit (PIU) act as a power interface for the firmware. Therefore, it provides memory mapped addresses to the firmware, which the firmware can write and read. The PIU is connected ot the AHB bus of the system. Every component is registered in the PIU. The PIU has two tasks: 1) decode firmware commands for power state changes and sent it to the corresponding component, 2) provide hardware performance characteristics to the firmware. In particular the CPU and memory controller track their idle and active times, i.e. their duty cycle. The firmware can access this information through the memory mapped addresses of the PIU.

B. *Firmware-based Power Management*

The power layer on top of SoCRocket does not have any logic to decide power state changes of the components. The power management strategy is completely implemented in firmware. Therefore, the firmware manages a set of data structures. Essentially, its the current power states of the components as well as counters and auxiliary data structures for guiding the power management and synchronizing firmware code called from application code and firmware code asynchronously triggered by interrupts. In the following we describe the duty cycle based power management strategies for the CPU and memory controller as well as firmware code to access IO devices in more detail.

1) Duty Cycle-based Power Management: The power state transitions of the CPU and memory controller are based on duty cycles (i.e. active and idle times) obtained from the hardware. Therefore, the PIU periodically triggers an interrupt. The interrupt handler is shown in Fig. 6. It retrieves the duty cycle bitvector from a memory mapped address and updates the power states of the CPU, memory and IO devices. A duty cycle of 75 for the CPU means the CPU spends 75% of the last time interval being active and was therefore idle 25% of the time.

As an example we will describe how theses duty cycles are used in the CPU power management strategy in the following.

```
void pm_irq_handler(int irq) {
  uint32_t dc = *DUTY_CYCLE_ADDR;
  update_leon3_power_state(CPU_DUTY_CYCLE(dc));
  update_memory_power_state(MEMORY_DUTY_CYCLE(dc));
  update_devices_power_state();
}
```

Fig. 6. Regularly triggered by interrupts to update power states of the hardware components

```
 1 void update_leon3_power_state(uint8_t leon3_dc) {
 2   switch (leon3_stat.pm_state) {
 3     case PM_STATE_RTM:
 4       if (leon3_dc < 75)
 5         leon3_change_power_state(PM_STATE_PS0);
 6
 7       break;
 8
 9     case PM_STATE_PS0:
10       if (leon3_dc < 50)
11         leon3_change_power_state(PM_STATE_PS1);
12       else if (leon3_dc >= 75)
13         leon3_change_power_state(PM_STATE_RTM);
14       break;
15
16     case PM_STATE_PS1:
17       if (leon3_dc < 25)
18         leon3_change_power_state(PM_STATE_PS2);
19       else if (leon3_dc >= 50)
20         leon3_change_power_state(PM_STATE_RTM);
21       break;
22
23     case PM_STATE_PS2:
24       if (leon3_dc >= 25)
25         leon3_change_power_state(PM_STATE_RTM);
26       break;
27
28     default:
29       assert (0 && "unkonwn power state");
30   }
31
32   if ((leon3_stat.pm_state == PM_STATE_RTM) &&
          (leon3_stat.num_rtm > 3)) {
33     leon3_change_power_state(PM_STATE_PS1);
34   }
35
36   if (leon3_stat.pm_state == PM_STATE_RTM)
37     ++leon3_stat.num_rtm;
38   else
39     --leon3_stat.num_rtm;
40 }
```

Fig. 7. Update CPU power state based on the CPU's duty cycle - regularly triggered by interrupts

The power management strategy of the memory controller and memories is similar to that of the CPU. For IO devices we use a strategy that will sent them to sleep mode in case they are currently not in use (i.e. there is no pending operation by the application code, which has been interrupted by this interrupt handler) and have not been used in the last time interval.

Fig. 7 shows the LEON3 power management strategy (i.e. the CPU core). The strategy will slowly increase the power save modes in case the CPU is idle, but it will immediately go into full power mode when sufficient work is available (*"ondemand"* policy). For example consider the case in Line 17-Line 21. The CPU is already in PS1 mode. The action depends on how much time the CPU has been idle in the last time interval:

- at least 75% : the CPU will go into PS2.
- between 75% and 50% : the CPU will stay in PS1.
- less than 50% : the CPU will change into full power mode (RTM).

```
 1 void io_device_read_data(int device_id, char *buf,
        unsigned int to_recv) {
 2   uint32_t n = leonbare_disable_traps();
 3   io_device_stats[device_id].pending_io = 1;
 4   leonbare_enable_traps(n);
 5
 6   io_ensure_power_up(device_id);
 7
 8   unsigned int num_recv = 0;
 9   while (num_recv < to_recv) {
10     if (io_get_available_chars(device_id) > 0) {
11       buf[num_recv] = io_read_char(device_id);
12       num_recv++;
13     } else {
14       io_wait_for_data(device_id);
15     }
16   }
17
18   n = leonbare_disable_traps();
19   io_device_stats[device_id].pending_io = 0;
20   leonbare_enable_traps(n);
21 }
```

Fig. 8. Firmware function to read data from an IO device

The lines Line 32-Line 39 ensure that the CPU will not stay for too much time in full power mode. In case the CPU does not become idle within three time intervals, it will change to a power save mode to avoid extensive power consumption (and also heat dissipation).

2) Read IO data: The application code does not access IO devices (e.g. peripherals, UART, ...) directly but only through a function layer provided by the firmware. For example to read 12 bytes from IO device 1, the application code will call io_device_read_data(**int** device_id, **char** *dst, **int** num) with device_id=1, num=12, and provide a char pointer *dst* to store the bytes. Fig. 8 shows the code to read data from an IO device. Interrupts are disabled while accessing data structures shared with the interrupt handlers The function *leonbare_disable_traps* disables interrupts and *leonbare_enable_traps* re-enables them again.

The function will power up the IO device if necessary, i.e. the function *io_ensure_power_up* will power up the io device in case it is currently in sleep mode (this happens when the io device is not used for some time intervals), and then iterate until to_recv chars have been received into *buf*. In each iteration the firmware will try to receive a single char (Line 10-12) or put the CPU to sleep mode in case no data is available (Line 14).

The *io_wait_for_data* function puts the CPU to sleep mode. Please note, that this is a shared operation with the interrupt handler who also updates the power state of the CPU. Therefore interrupts are disabled before sending the CPU to sleep mode. In SoCRocket we ensure that interrupts are automatically re-enabled when the CPU goes into sleep mode. Otherwise the CPU would not wakeup again as no interrupts would come in.

TABLE I
EXPERIMENT RESULTS (SIMULATION TIME IN SECONDS, POWER CONSUMPTION IN μJ) ON THE SOCROCKET PLATFORM

Scenario	Full Power Mode					Firmware-based					Difference	
	sta	int	swi	all	time	sta	int	swi	all	time	Power	Speed
S1) High CPU Load	110995	140838	3136	254969	2.03	40949	52009	1447	94405	2.42	-62.97%	+19.21%
S2) Interrupt Intensive	59572	75589	26113	161274	1.09	49681	63055	16609	129345	1.68	-19.80%	+54.13%
S3) Alternating Workload	151497	192229	53649	397375	2.77	83682	106234	21072	210988	3.74	-46.90%	+35.02%
S4) Memory and IO Intensive	416425	528388	59748	1004561	7.63	96399	122250	59748	278397	10.88	-72.29%	+42.60%
S5) Small Tasks	99540	126303	44813	270656	1.82	80772	102510	25473	208755	2.88	-22.87%	+58.24%

On average 8,000,000 instructions executed on the SoCRocket platform per concrete workload.

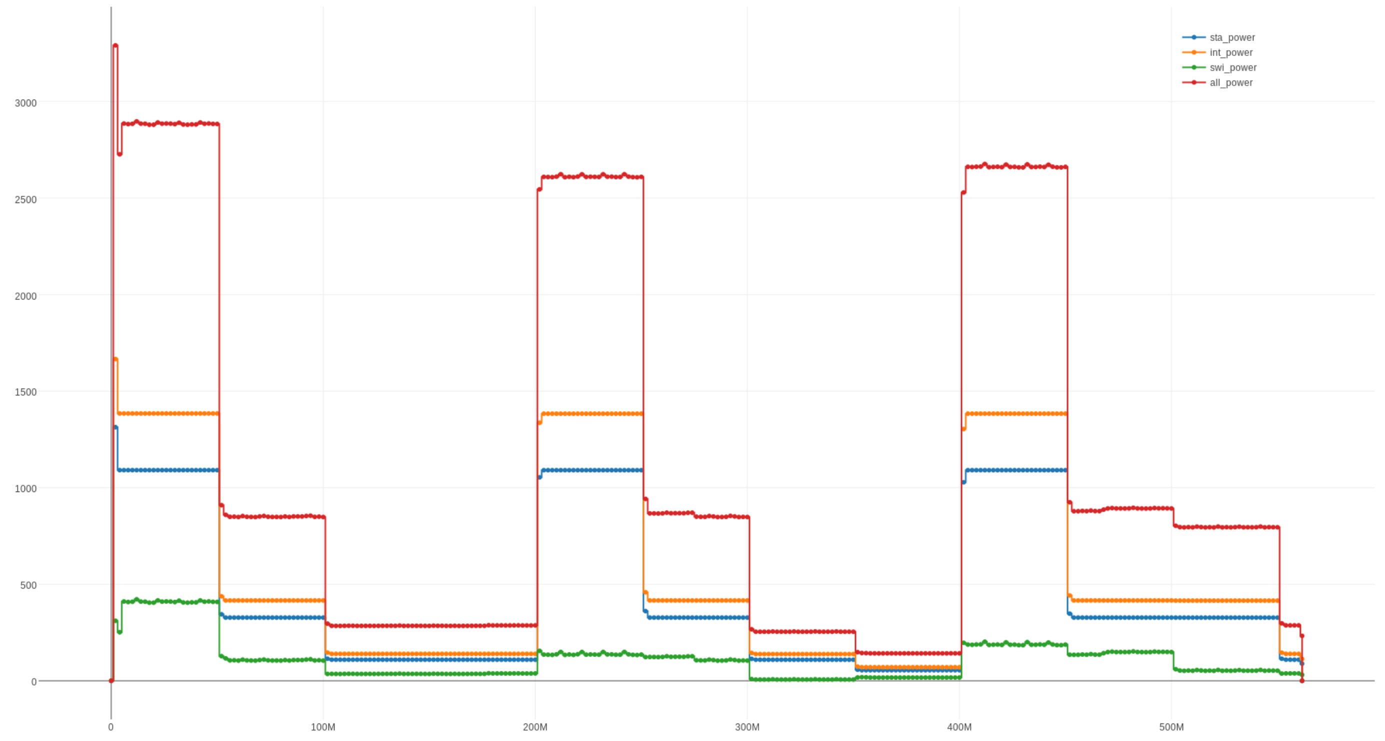


Fig. 9. Example power diagram of a concrete workload from scenario S3. It shows the static, internal, switching and total power consumed of all components at different simulation time steps.

C. Results

In this section we present results of applying our validation approach on the SoCRocket VP. Validation results for 5 scenarios are presented in Table I. For every scenario we generate 50 concrete workloads using our constrained random technique. For every concrete workload a concrete application is generated and executed in full power mode (RTM) and with firmware-based power management (FW) on the SoCRocket platform. Table I shows the average results over all runs. All power consumption values are specified in micro Joule (μJ) and simulation time in seconds. On average 8,000,000 instructions are executed on the SoCRocket platform per concrete workload. Validation of a scenario takes 15 minutes in average. All experiments have been run on a Linux machine with a 2,4 GHz Intel and 16 GB RAM. Please note, with simulation time we do not refer to the wall time, but the time it takes for the application code to execute on the SoCRocket platform, i.e. when the code will run on the real hardware (estimated at system level using the SystemC-based VP). Therefore, a higher simulation time directly implies a lower performance of application code. We define the power- and performance-budget of the firmware-based power management to be 80% of power consumption and 150% of simulation time compared to full power mode, i.e. the power management should save at least 20% power and should reduce the performance by no more than 50%.

Table I shows the scenario name in the first column. The second and third columns show results for simulating the concrete application in full power mode (RTM) and with firmware-based power management (FW), respectively. For both modes we further report the static (sta), internal (int), switching (swi) and total (all) power consumed, as well as the simulation time (time) on the SoCRocket platform. The fourth column shows the difference in power consumption and simulation time on the SoCRocket platform between both modes. For example, it can be observed that the firmware-based power management strategy on average saves 62.97% power consumption at the cost of losing 19.21% performance compared to full power mode for the concrete workloads in scenario S1. In particular, we consider the following scenarios:

S1 describes workload that is very CPU intensive. It generates instructions with high CPU load.

S2 generates interrupt intensive workload. Application code is interrupted by incoming interrupts with very high frequency.

S3 describes workload with alternating instructions blocks. It ensures the neighboring code blocks do not have the same instruction type.

S4 generates workload that is very memory and IO intensive. The CPU load is comparatively low. It ensures that all IO devices are used with different processing speed.

S5 describes workload with many small tasks. This leads to application code with many small blocks of different instruction types.

It can be observed that the power- and performance-budgets are satisfied for most scenarios (S1,S3,S4). Scenario S2 slightly exceeds both the power- and performance-budget, and S5 the performance budget only. S2 describes interrupt intensive code which will interrupt the normal application flow and perform some computation before giving the control back. This can lead to inefficient sleep intervals when waiting for IO, as the CPU will wakeup from the interrupt (and also check the device for available input again) to be put back to sleep again. S5 changes the workload type very frequently. When changing from CPU intensive to IO or memory bound code and vice versa, the firmware will reduce the power state of the CPU and power it back up again. Therefore, both S2 and S5 can lead to an increased switching frequency of power states in firmware code. We can use our approach to generate additional workload for further investigation. Furthermore, we can plot power diagrams which show the power consumption (of the whole system or any particular component) at different simulation time steps to get further insight. An example power diagram is shown in Fig. 9.

V. Conclusion and Future Work

In this paper we presented an approach for early power validation of firmware-based power management at system level using SystemC-based Virtual Prototypes (VPs). We employ constrained random (CR) techniques to generate concrete workloads and then validate that available power and performance budgets are satisfied, by using a power aware simulation on the VP. First experiments with the LEON3-based SoCRocket VP demonstrate the applicability and effectiveness of our approach.

For future work we plan to investigate power aware coverage metrics in order to evaluate the thoroughness of the validation. A viable solution might be to combine information about reached power states of all components of the system as well as transitions between power states with well known code coverage metrics. To speed-up the overall validation, we plan to distribute the processing of scenarios using multiple threads or even clusters connected by a network. This is a natural and very promising extension as all scenarios are processed independent of each other. Finally, we plan to extend our constraint language and program generator to support generation of multi-threaded workload.

References

[1] B. Bailey, "Power limits of EDA," 2016, http://semiengineering.com/power-limits-of-eda.

[2] *IEEE Standard SystemC Language Reference Manual*, IEEE Std. 1666, 2011.

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

[4] V. Herdt, H. M. Le, D. Große, and R. Drechsler, "Compiled symbolic simulation for SystemC," in *ICCAD*, 2016, pp. 52:1–52:8.

[5] H. M. Le, V. Herdt, D. Große, and R. Drechsler, "Towards formal verification of real-world SystemC TLM peripheral models - a case study," in *DATE*, 2016.

[6] 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*, vol. 37, no. 8, Part C, pp. 966 – 980, 2013.

[7] S. Schürmans, D. Zhang, D. Auras, R. Leupers, G. Ascheid, X. Chen, and L. Wang, "Creation of ESL power models for communication architectures using automatic calibration," in *DAC*, May 2013, pp. 1–6.

[8] G. Onnebrink, R. Leupers, G. Ascheid, and S. Schrmans, "Black box ESL power estimation for loosely-timed TLM models," in *International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)*, July 2016, pp. 366–371.

[9] J. Yuan, C. Pixley, and A. Aziz, *Constraint-based Verification*. Springer, 2006.

[10] 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.

[11] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: A cycle-accurate energy estimation tool," in *DAC*, 2000, pp. 340–345.

[12] J. Laurent, N. Julien, E. Senn, and E. Martin, "Functional level power analysis: An efficient approach for modeling the power consumption of complex processors," in *DATE*, 2004.

[13] S. K. Rethinagiri, O. Palomar, R. Ben Atitallah, S. Niar, O. Unsal, and A. C. Kestelman, "System-level power estimation tool for embedded processor based platforms," in *Workshop on Rapid Simulation and Performance Evaluation: Methods and Tools*, 2014, pp. 5:1–5:8.

[14] O. Mbarek, A. Pegatoquet, and M. Auguin, "Using unified power format standard concepts for power-aware design and verification of systems-onchip at transaction level," *IET Circuits, Devices Systems*, vol. 6, no. 5, pp. 287–296, Sept 2012.

[15] J. Karmann and W. Ecker, "The semantic of the power intent format upf: Consistent power modeling from system level to implementation," in *International Workshop on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, 2013, pp. 45–50.

[16] 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 Workshop*, 2016, pp. 102–113.

[17] P. Sayyah, M. T. Lazarescu, S. Bocchio, E. Ebeid, G. Palermo, D. Quaglia, A. Rosti, and L. Lavagno, "Virtual platform-based design space exploration of power-efficient distributed embedded applications," *ACM Trans. Embed. Comput. Syst.*, vol. 14, no. 3, pp. 49:1–49:25, Apr. 2015.

[18] F. Haedicke, H. M. Le, D. Große, and R. Drechsler, "CRAVE: An advanced constrained random verification environment for SystemC," in *SoC*, 2012.

[19] H. M. Le and R. Drechsler, "CRAVE 2.0: The next generation constrained random stimuli generator for SystemC," in *DVCon*, 2014.

[20] "HW-SW SystemC co-simulation SoC validation platform," TU Braunschweig, Tech. Rep., 2012.