PowerMonitor: a Versatile API for Automated Power-Aware ESL Design

Yasaman Samei, Rainer Dömer
Center for Embedded Computer Systems
University of California, Irvine, USA
ysameisy@uci.edu, doemer@uci.edu

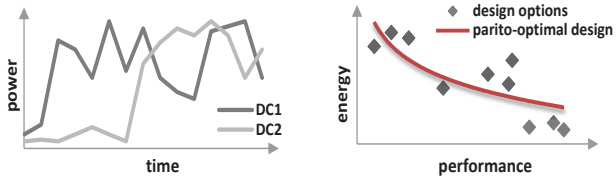
***Abstract*—System Level Description Languages (SLDL) emerged a decade ago for high-level System-on-Chip (SoC) design and efficient design space exploration. Initially performance and area constraints were the major concerns. Nowadays the shrinking of transistor size has brought power and temperature to top of the list of designer concerns. Although SLDLs are well-defined for functional and timing modeling, the dimension for power- and temperature-aware modeling is missing and needs to be added manually by the user in an ad-hoc fashion. In this paper we introduce; PowerMonitor as an Application Programming Interface (API) capable of (a) power annotation in the executable model, (b) power-aware simulation, and (c) monitoring and graphically representing power dissipation at the system level with flexible granularity and compositions.**

***Index Terms*—System Level Description Language; Power-aware Design; Annotation and monitoring; Profiling;**

I. Introduction

Power and temperature are now one of the main concerns in SoC design and directly affect each and every one of the design decisions. Each of the design components have different behaviors in terms of power dissipation and temperature. In other words there is no unique formula for power consumption of each unit, and in reality it is even more complex since different applications cause different power dissipation behavior in a single unit, which makes power dissipation data-sensitive. Hence, thorough power estimation needs detailed information about power dissipation of every unit while executing a certain set of operations as well as precise profiling results of their interactions, communication details, memory accesses, and cache hits/misses in addition to their operating conditions.
Currently many-core architectures have become the default in SoC design and extracting the above mentioned information is now even more essential and at the same time more complex. Additionally by considering power in the early design stages, the device life-time is significantly extended and the reliability and performance of the system improves, revealing another aspect of power evaluation importance.

Power modeling and estimation can be implemented in all design levels with different accuracy and speed trade-off. Desired power related information for design verification at different design stages are presented in Figure 1. Any proposed approach for system level power evaluation should support monitoring power for peak and average power evaluation of the entire system as well as for each design component separately as it is shown in Figure 1(a). Moreover, design space exploration and time-power constraints trade-off are required to start at the system level regardless of degraded accuracy, Figure 1(b).



(a) Power dissipation over time in design components, DC1 and DC2

(b) Design space exploration

Fig. 1. Power & Time Analysis at System-Level

II. Related works

Although power aware design is crucial in Electronic System Level (ESL) design, SLDLs are not supporting this feature natively. The initial solution to this deficiency, which is still being used in some cases, was the use of spreadsheets. The spreadsheet approach is fast for average power evaluation, however verifying power dissipation over time and securing the system against power peaks is only possible when timing is taken into account [1]. There has been some research on extending SystemC or providing new libraries for power aware simulation. PowerSC [2] is a library proposed for power aware simulation of SystemC-based TLMs; it can gather switching activity information during the simulation. TLM Power 3.0 [3] counts bit level activities in a TLM model. A similar approach is presented in PKtool [4].
A limitation of these approaches is that the annotation of power and timing functions to TLMs is performed manually, which is a tedious task for the designer and not scalable. Another drawback is that modifying the design with power aware functions changes the pure specification of the model. In [5] a methodology to convert RTL designs to power and timing annotated TLM models is presented. In this work power modeling is base on existing RTL models. In this paper, we present PowerMonitor, a library for ANSI-C based SLDLs such as SystemC [6] or SpecC [7]. PowerMonitor allows power-aware design to be orthogonal to conventional performance-aware design. Power evaluation, measurement, annotation and visualization are developed and adapted for SLDL. The proposed API can collect all components switching

activities, interactions and communication details. The proposed extension and methodology for power modeling can easily be applied for any other SLDL as well.

III. Overview

To develop PowerMonitor, we defined *PowerMeter* as a virtual tool for power evaluation. *PowerMeter* is designed similar to an actual power meter that is used for power monitoring and measurements at the physical level. *PowerMeter* can monitor and measure power, and generate an on-line log of power activities during simulation. The collected information can be used for power optimization, power-aware schedulers, and ultimately increasing life-time and reliability of the system. We have developed *PowerMeter* as an API and a library called PowerMonitor which offers power analysis along with timing log and graphical reports. *PowerMeters* are annotated to the specification model with energy dissipation and time consumption information. The implemented functions enable *PowerMeter* to monitor and measure power consumption over time, and within different system components or application segments. Table I shows an overview of existing time evaluation features along with added power features. In order to support power analysis the PowerMonitor library is developed with specific power and energy related units as well as functions for power consumption and monitoring. Owing to PowerMonitor, the design exploration process can be initialized at the system level via verifying both timing and power constraints as it is shown in Figure 1.

TABLE I
Time and Power Modeling

Features	Time	Power
Meters	sim_time *time*;	PowerMeter *pm*;
Units	SEC, MILLI_SEC,..	JOULE, MILLI_JOULE,.. WATT, MILLI_WATT,..

Consumption	wait(*event*);
	waitfor(*time*);
	do{...}timing{...}
		pm_consume_dynamic(*&pm,dynamic*);
		pm_consume_static(*&pm,static*);
		pm_consume_total(*&pm,dynamic,static*);

Monitor	*time* = now();
	time = delta();
		dynamic=pm_dynamic(*&pm*);
		static=pm_Static(*&pm*);
		total=pm_power(*&pm*);
		display(*&pm*);
		printPower(*&pm*);

IV. Power Monitor

The PowerMonitor library uses *PowerMeters* (PM) to extract power dissipation information from the design, and perform power analysis. Each PM monitors static and dynamic power dissipation over time and with proper units. In order to support different operations over PM, such as power consumption, power monitoring and power constraint evaluation capabilities, multiple functions are developed in the PowerMonitor library which are presented in the following subsections.

A. PowerMeter

Each PM is capable of measuring power for its assigned part of the design. There is no limit on defining PMs nor associating them with particular part of the design. A PM can be defined for any block, component, and behavior of the code. The PM class objects can be defined with type `PowerMeter` within the specification model code as:

```
PowerMeter pm;
```

B. Static & Dynamic Power

The PMs are treated as an actual power meter in this work. Therefore all the expected functionalities from a power meter are implemented as built-in features in the PowerMeter library. For each PM the main required information encompasses the energy dissipated over time due to the switching activity and leakage. Hence these values are monitored and maintained internally for each PM.

```
long double Dynamic;
long double Static;
sim_time time;
```

C. Power Consumption

To represent the power dissipation in the system model a set of functions is presented; `pm_consume_dynamic` for spending dynamic power only, `pm_consume_static` for static power only, and `pm_consume_total` for spending both dynamic and static power.

```
void pm_consume_dynamic( PowerMeter *pm,
                         const long double Dynamic);

void pm_consume_static( PowerMeter *pm,
                        const long double Static);

void pm_consume_total( PowerMeter *pm,
                       const long double Dynamic,
                       const long double Static);
```

When these functions are called the `Dynamic` and/or `Static` energy values along with the associated time stamps are recorded for the specified PM. Then in the analysis phase this information is deployed to generate graphs and different forms of power reports.

D. Monitor Power Dissipation

To access the spent dynamic, static or total power value a set of functions are implemented. The function `now()` which returns current simulation time is already available in SLDLs.

```
void const long double pm_dynamic( PowerMeter *pm );
void const long double pm_static( PowerMeter *pm );
void const long double pm_total( PowerMeter *pm );
```

For monitoring dynamic and static power dissipation different functions are developed in PowerMonitor. The provided functions are `print_pm_dynamic`, `print_pm_static` and `print_pm_total`, which print current dynamic, static and both respectively. These functions evaluate the power values and output the power report with proper units:

```
void print_pm_dynamic( PowerMeter *pm );
void print_pm_static( PowerMeter *pm );
void print_pm_total( PowerMeter *pm );
```

In order to evaluate the power dissipation over time a power-time diagram is a convenient solution for the designer. Function `display` is developed to visually display power consumption during simulation:

```
void pm_display( PowerMeter *pm );
```

Figure 4 represent an example of using function `display`.

E. Power Analysis

When the model simulation is over, PMs contain power behavior of the design. At this point, the designer can ask for different power information such as peak power, average power, display multiple PMs power dissipation in one graph, adding power dissipation of different PMs, or printing all static, dynamic dissipated values. Some auxiliary functions are implemented in order to allow these analyses. Function `pm_multidisplay` gets PMs as input and returns a merged graph of power dissipation over time of all the input PMs.

```
void pm_multidisplay(PowerMeter *pm1, ...);
```

An example of using `pm_multidisplay` is represented in Figure 4.

During the power analysis, the user may want to add up power dissipation in multiple PMs and run further investigations. In `pm_add` a PM is returned that contains the summation of all input PMs

```
void pm_add( PowerMeter *result, PowerMeter *pm, ...);
```

For user convenience, in order to get a general report of all PMs the `report_power` function can be used. It iterates through all PMs and prints the static, dynamic, total, average and total time spent in each PMs.

```
void report_power();
```

Apart from the above mentioned functions, PowerMonitor provides functions so PMs can be initialized, reset, or set to certain values for dynamic power, static power and time. The `pm_assign_dynamic`, `pm_assign_static`, `pm_assign_total` functions are used to initialize or assign values to a PM at time `t`.

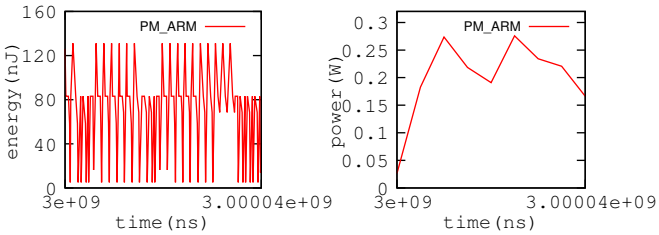
```
void pm_assign_dynamic( PowerMeter *pm,
                        sim_time t,
                        const long double Dynamic);

void pm_assign_static( PowerMeter *pm,
                       sim_time t,
                       const long double Static);

void pm_assign_total( PowerMeter *pm,
                      sim_time t,
                      const long double Dynamic,
                      const long double Static);
```

To reset the PM power values `pm_reset` can be called.

```
void pm_reset(PowerMeter *power);
```

V. PowerMonitor: Energy & Power

Each PM tracks the energy consumption due to switching activity and leakage. So the user can monitors how energy is spent at each PM over time and has the option to access the trace files with any requested timing resolution. An example of energy consumption at PowerMeter `pm` is presented in Figure 2(a). The power behavior is a direct function of energy consumption over time and can be obtained easily from energy consumption information. The PowerMonitor assists the user to generate any format of power report and diagrams for needed simulation intervals and timing resolution. An example of power dissipation at PowerMeter `pm` is shown in Figure 2(b), where the average power dissipation is obtained from energy consumption with a sampling frequency of 1 milli second.



(a) Energy consumption (b) Power dissipation

Fig. 2. Energy consumption and derived power dissipation

VI. Automated PowerMeter Annotation

The fact that applying power and time related functions to every basic block, behavior or component of the design is tedious and inefficient, we have provided an automated power annotation tool in the context of the SpecC-based System-level Design Environment (SCE) [8]. To offer a precise power report we decided to attribute every basic block of the system model code with its power information. The annotation tool attaches PMs to the system level model based on user choice of granularity and inserts `pm_consume_total` and `waitfor` functions to every basic block of the design for the associated PM. These two functions mimic energy and time consumption within each basic block of the code. In this case, the designer only needs to provide the power and time libraries for SCE and choose the granularity of the insertion.

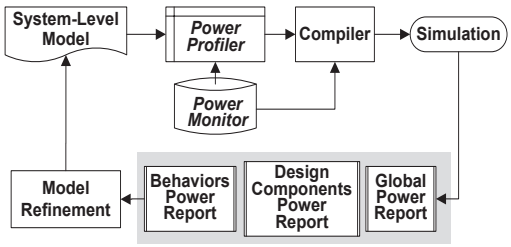


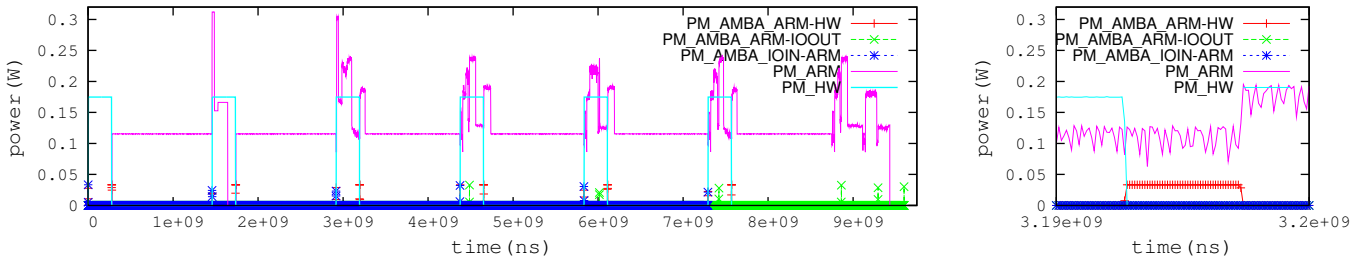
Fig. 3. Power-aware design flow

A. PM Granularity

From the user perspective, power evaluation is performed for certain architectural components of the design like a processing element, certain set of the behaviors within the system model, or globally over the whole system. Thus we provided same levels of granularity for all PM allocations. The designer can choose to allocate PMs to all the design components, behaviors, or only allocate a single PM for the entire system.

B. Design Flow

A general design flow using PowerMonitor is shown in Figure 3. Once the specification model is ready, a power model is associated with every design component. These power



(a) Canny edge detector 6 image-stream

(b) Zoomed in Canny

Fig. 4. Power dissipation of Canny Edge Detector application using PowerMonitor

models (equations, operating conditions,..) are user-defined, nevertheless applied to each component automatically through a modification to SCE, the tool for refinement of SpecC models. Next *PowerProfiler* performs power annotation and, followed by simulating the power-annotated system model, different formats of power reports can be generated.

The User does not need to modify the specification model in order to evaluate or test design constraints, new PE mapping, different power behavior, or power libraries. In other words, the cost of new design exploration is updating the mappings and/or libraries without any power annotation related changes in system model code. Thanks to this feature PowerMonitor significantly reduces design space exploration effort and power optimization.

The generated library source code can be unknown to the user as well. Therefore it is effortless for the designer to generate power reports. The only needed change is the inclusion of the PowerMonitor header file. No extra library or compiling feature is required for applying PowerMonitor. Moreover, the annotated power information can be added to the original design specification or it can only be used for power aware simulation and generating power reports. The power API has the capability of providing the trace of each behavior and PE utilization with different time precision, which is significantly helpful for verifying the peak power and power optimization.

VII. Case Study: Canny Edge Detector

We have examined the PowerMonitor together with the automated power-annotation tool on the Canny edge detector [9]. The Canny is a real-life image processing application implemented with a 4-stage pipeline configuration. In this architecture an ARM processor along with a custom hardware (HW) unit are communicating through an AMBA BUS and double-handshake channels. Figure 4 represents the power dissipation graph for a stream of 6 images. The automated PowerMonitor is applied at the PE level and the power graph within all the processing and communication components of the design is displayed. Figure 4(a) shows the application life-time in addition to associated power dissipation and illustrates how the pipeline is getting filled and drained, and when the images are being transferred between ARM and HW. A short time interval when the HW is sending the processed image 3 to the ARM is shown in Figure 4(b). Furthermore the peak power and working intervals of each component can be captured and altered as the designer desires. Apart from diagrams the detailed log of PEs power dissipation can be used for further analysis or power optimization.

VIII. Conclusion & Future Work

In this paper we presented PowerMonitor as a power extension to SLDL. Power evaluation at the system level adds a new dimension to the system level design process and improves the design exploration experience significantly. An overview of *PowerMeter*, its functions, along with the automated power annotator were described followed by an experiment on Canny edge detector application. Thanks to PowerMonitor and automated annotation feature a convenient framework for power analysis and optimization is offered for C-based SLDLs.

For future work we are planning to deliver power optimization techniques to be applied automatically at the system-level. Also we want to extend the PowerMonitor for temperature and reliability evaluation features.

References

[1] B. Fischer, C. Cech, and H. Muhr, "Power modeling and analysis in early design phases," in *Proceedings of the conference on Design, Automation & Test in Europe*, p. 197, European Design and Automation Association, 2014.

[2] F. Klein, R. Azevedo, L. Santos, and G. Araujo, "Systemc-based power evaluation with PowerSC," *Electronic System Level Design*, pp. 129–144, 2011.

[3] D. Greaves and M. Yasin, "TLM POWER3: Power estimation methodology for SystemC TLM 2.0," in *Models, Methods, and Tools for Complex Chip Design*, pp. 53–68, Springer, 2014.

[4] G. Vece, M. Conti, and S. Orcioni, "PK tool 2.0: a SystemC environment for high level power estimation," in *Electronics, Circuits and Systems, 2005. ICECS 2005. 12th IEEE International Conference on*, pp. 1–4, IEEE, 2005.

[5] D. Lorenz, K. Grüttner, N. Bombieri, V. Guarnieri, and S. Bocchio, "From rtl ip to functional system-level models with extra-functional properties," in *Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pp. 547–556, ACM, 2012.

[6] T. G. S. Liao, G. Martin, S. Swan, and T. Grötker, *System design with SystemC*. Springer, 2002.

[7] D. D. Gajski, J. Zhu, R. Domer, A. Gerstlauer, and S. Zhao, "SpecC: Specification language and methodology," 2000.

[8] R. Dömer, A. Gerstlauer, J. Peng, D. Shin, L. Cai, H. Yu, S. Abdi, D. D. Gajski, *et al.*, "System-on-chip environment: a SpecC-based framework for heterogeneous MPSoC design," *EURASIP Journal on Embedded Systems*, vol. 2008.

[9] R. Han, Y. Samei, and R. Doemer, "System-level modeling and refinement of a canny edge detector," *Center for Embedded Computer Systems*, 2012.