Automated Estimation of Power Consumption for Rapid System Level Design

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

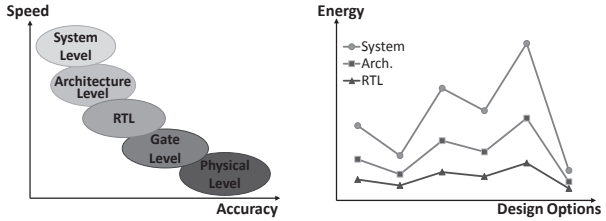
***Abstract*—This paper describes an early power estimation method for Electronic System Level(ESL) design, which provides a scalable API to support automated power profiling and analysis at the early stages of the design process. The proposed framework utilizes a high-level power modeling mechanism along with an automated profiler to extract energy activity from the simulated system model. These two features are integrated into *PowerMeter*, a framework that automatically annotates power meters as well as energy and performance functions into the executable model. This integrated profiling helps the designer to rapidly explore the design space, trading off performance against power cost in order to make best design decisions. Our approach also provides the designer with the ability to quantify the effect of revisions in the ESL design models, in terms of both power and performance. Despite the high abstraction level, our results show that the *PowerMeter* delivers rapid estimates with high fidelity and at minimal cost.**

***Index Terms*—Power; Performance; Profiling; System Level Design; Fidelity;**

I. Introduction

For the past few decades, semiconductor capabilities have been improving as Moore's law predicted. Transistor size has been shrinking and technology size will be less than 20nm in the near future. These improvements enable the designer to come up with more complex systems. However, this has made power dissipation a major design obstacle. The fact that power dissipation in small technology sizes increases due to high leakage power makes power optimization a primary target of the design process.

Conventionally, power consumption is considered in the later stages of the design process, like the architecture level [1], RTL [2] [3], gate level [4], and physical level, where detailed information about the design is available. Although there are many power-aware design tools at these lower levels, the simulation and evaluation time are high and often beyond the time-to-market requirements. To tackle the long simulation time as well as avoiding time consuming design modifications at lower levels, designers are changing the level of abstraction to the system level, typically by means of trading accuracy in favor of speed. Fig. 1 shows a prospect of power estimation at the system level. The speed-accuracy trade-off in power estimation at different design levels is demonstrated in Fig. 1(a). The accuracy and time trade-off is the main challenge in power estimation. Here, a powerful and automated API for system level early estimation, as proposed in this work, can shorten the design cycle of low-power systems tremendously. The earlier the optimization starts, the more efficiently devices can be produced. Consequently, design constraints such as power, performance and die size are ought to be taken into account from the early stages of the design flow.



(a) Speed-accuracy trade-off in power estimation (b) High fidelity energy estimation at different levels of design

Fig. 1. Power Estimation at Different Design Levels

The System level is the starting point for design constraints characterization as well as design space exploration. Therefore design decisions, such as component selections, Hardware(HW)/Software(SW) partitioning, communication schemes, number of cores, and power reduction techniques, are ideally all made at the system level. Here, it is critical to make correct decisions. To achieve this goal, a structured ESL tool suite is required to perform assessments. In order to select the best design options at the system level, *relative* accuracy and high *fidelity* [5] are essential. Fig. 1(b) presents the notion of power estimation from a desirable system level estimator where the comparative relations of the design options are accurate. Thus, our goal in this work is to develop a system level performance estimator with a high level of *fidelity*.

For system level power estimation, the two available design inputs are 1) the specification model implemented in a System Level Description Language (SLDL), such as SystemC [6] or SpecC [7]; and 2) the power models of different system components, such as processors and IPs. Many studies have been performed on characterizing power dissipation for memories [8], communication channels [9], and processing elements (PE) [1] using experimental and statistical analyses at lower levels, by actual measurement or by applying power model builders, such as PowerMixer [10]. The fidelity and accuracy of the system level power estimation directly depends on available functional information and extracted

power activity details in the design, as well as applying effective power models.

In this work, *PowerMeter*, a rapid and automated system level power estimator, is introduced to monitor the energy consumption of a system model by extracting comprehensive power activity of the modules without any manual model modifications. Although a scheme for ESL power models creation is described and used in *PowerMeter*, power modeling is not the focus of this work.

In the next section, we look at the available tools and strategies for power estimation in related work, followed by an introduction to the proposed approach for rapid power estimation. Our power model database is presented in Section IV. Section V explains the development steps of the power estimator; Section VI presents a case study on JPEG image encoder, Section VII evaluates the approach in terms of speed, fidelity and accuracy; and Section VIII provides a summary and conclusion.

II. RELATED WORK

Power estimation and modeling has been the focus of many research efforts. Although power aware design is crucial in Electronic System Level (ESL) design, SLDLs are not supporting this feature natively. Most of the proposed power estimation methods rely on similar foundations that can be categorized in to three main groups: cycle accurate, instruction based, and functional level based models.

In cycle accurate methods, such as McPAT [1], Wattch [11] and Simplepower [12], power is analytically quantified by monitoring operations and transactions cycle by cycle and applying power models from the micro-architectural level for each involved unit. Generally, the simulation time in this group is long while the accuracy is high.

Powersim [13], an example of an instruction-based power model, presents a C++ library that monitors a limited set of SystemC operators and applies an energy model to the monitored operations.

A functional level power analysis approach is used in PETS [14]. PETS uses generic power models while extracting micro architectural activity to tackle the accuracy-speed trade-off. COMPLEX [15] is a framework for HW/SW co-design at system levels and allows applying hybrid combination of power models from various works for different design components. The final group is functional level power analysis which is applied in several tools, such as TLM POWER3 [16], and PKtool [17]. In [16], bit level activities are counted in TLM models, while PKtool is presented in the form of a class library for SystemC by means of power estimation and analysis at the system level. Both of these tools help to embed the power details and abstract power related information, such as Hamming distance of the signals, to the design. However, the process is manual and user-oriented, and therefore neither easy to apply nor scalable.

These power estimators at system level generate general power reports in form of an average power consumption, performance, or the trace of total power of the design only. However, our proposed system level power estimator enables the designer to concentrate on any HW or SW part of the design, their working intervals, their power consumption over time, and modes of operation, with user-defined granularity. This feature allows to make critical design decisions and find power optimization solutions easily and rapidly with clear understanding of the design.

III. OVERVIEW

A design space exploration platform named System-on-Chip Environment (SCE) [18], has been developed for SpecC language. This framework uses a top down design approach,

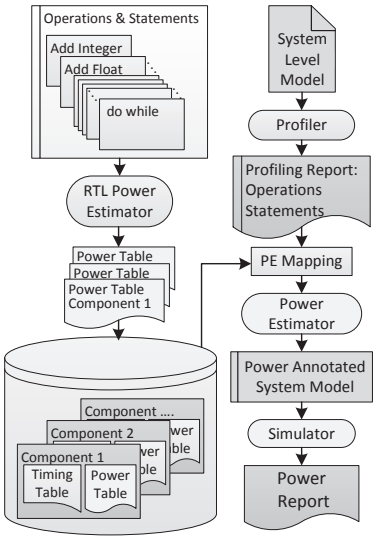


Fig. 2. Power Estimation Flow

and enables refinement of the specification model all the way down to an Instruction Set Simulation (ISS) model. We have integrated the proposed power estimation API into this framework. Our approach for power analysis introduces power as a new design constraint for design space exploration using SCE. Fig. 2 shows an overview of this methodology. The automated estimation of power consumption at the system level is performed through the following steps.

A. Profiling the Design Model

The design process of embedded systems starts with the original specification model of the design. This model only contains system functionalities, without any timing or architectural information. In order to evaluate power dissipation, the specification model is profiled by means of capturing its energy consumption activities. Similar to SpecC Profiler [19], our power tool extracts the energy activity details on operations, communications, and memory characteristics of each behavior. Our profiler automatically annotates the design source code to generate both *static* and *dynamic* reports. In a *static* report, the evaluation is based on pure code, while

in a *dynamic* report the actual number of executed operations and statements during simulation is taken in to account. For power estimation, the *static* report of each basic block within the model is used which contains sufficient information on the ESL model running its application. Monitoring these basic blocks is also shown to be a fine level of granularity for performance estimation in [19].

B. Mapping the Design to PEs

Once the specification model is profiled, the next step is architecture exploration. There are many design choices, such as processing elements, communication methods and memory elements, to be evaluated before refining the architecture. At this stage, energy dissipation and performance are of high importance. The proposed power estimator uses the profiling report from the previous step, along with the power models of the selected PEs from the database, to generate power reports. The power models can be suggested by the user or selected from the power model library, which is described in Section IV.

C. Adding Power Annotation to the Design

Once the PEs are selected, the power estimator can generate the power dissipation reports. Our power estimator has an API that automatically annotates the design to extract the energy dissipation values of each component. The estimator attaches power meters to monitor the energy dissipation over time. It also inserts energy and performance functions into each basic block in order to mimic the energy and time consumption of the component. The granularity of the power meters within the model can be selected by the designer. For instance, a global power meter can be assigned to the entire model, or power meters can be assigned to each behavior or component separately. The power and timing parameters, such as frequency, can also be set by the user.

D. Generating Power Reports

At the final step, the annotated model is simulated at the system level to produce the report of power dissipation, energy consumption and performance of the model under designated PEs. Since our power models are light weight, the slow down in simulation performance is negligible.

IV. POWER MODELS

To work with the proposed automated power estimator, any power model can be applied during IP integration and mapping to the behaviors. Therefore, power characterization and modeling is not the main focus of this work. However we used a simple power model generation method to generate some default power models. Our system level power models are based on power reports and simulation information from lower levels of design. The main idea behind these power models, is similar to power modeling presented in Tiwari et al. [20]. Later in the experimental results we show that our proposed power API is able to deliver high fidelity even with these basic power models. In these models, each expression and statement has been measured using power simulators [21] [22]. This process is only performed once for each PE and the resulting power tables are added to the database as shown in Fig. 2. During design space exploration within SCE, each PE power model is automatically added to the design while mapping design to PEs.

In our studies, we have used ARM-based and Intel Nehalem processor architectures. Each expression and statement in the source code of the model owes a dynamic and static energy consumption value. The dynamic energy is the energy spent as *dynamic switching* and *short circuit* power, and static energy represents the energy dissipation due to *leakage*.

TABLE I
DYNAMIC AND STATIC ENERGY VALUES (*nJ*) FOR OPERATIONS & STATEMENTS FOR ARM7 PROCESSOR

Operations Expressions	Types					
	integer		float		long long integer	
	Dynamic	*static*	*Dynamic*	*static*	*Dynamic*	*static*
Add	4.9	0.07	6.0	0.1	13.8	0.2
Division	57.0	1.3	82.1	2.3	298	6.9
do while	6.7	70.1	6.7	0.1	6.7	0.3

Table I shows a section of the generated energy tables for an ARM7 processor. In order to generate dynamic and static values, each expression/statement is simulated multiple times. Furthermore, simulations are statistically analyzed through regression analysis to even out the effect of cache misses, pipeline stalls, or any other situation that possibly increases the simulation time as well as energy consumption. For instance, the integer *"add"* operation has been tested with 10^1, 10^2, 10^4, 10^5, 10^6 integer *"add"* operations and the energy consumption is evaluated for each unit of the target architecture. Fig. 3 shows the energy consumption in different units for a series of *"add"* operations. The results show that for a large number of *"add"* operations the consumed energy will not vary significantly. Hence, we picked the average energy dissipation of 10^4 *"add"* operations as the reference energy.

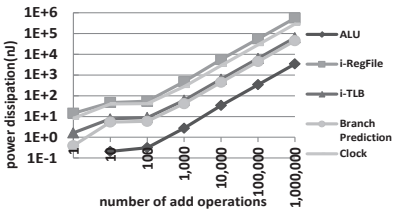


Fig. 3. Power dissipation (*nJ*) for *n "add"* operations

Other expressions, statements, and operations were similarly studied for each type in order to obtain dynamic and static power parameters.

V. AUTOMATED POWER ESTIMATOR

In this section, our power estimator API is explained in detail using an example.

A. Execution and Operator Profiling

The system level model of the design describes the functional blocks and communication channels plus their execution order(FSM, parallel, sequential, pipeline, etc.). Having a comprehensive profiler is a key factor in developing an accurate and fast system level power estimator. In this work we developed a profiler that summarizes all the execution counts of all expressions and statements with their associated types. An example of a profiling report for a specification model is presented in Fig. 4. In this model, two instances of behavior *A*, *A1* and *A2*, are running in parallel with an instance of behavior *B* named *B1*. The profile also shows the number of times that behaviors *A* and *B* are executed, along with their operations and corresponding data type. Behaviour *Main* contains *A1, A2* and *B1* as child behaviors; hence, its profiling report covers its own operations and expressions plus its child instances.

B. Power API

To evaluate the power consumption of the design at the system level, we insert the power activity information in to the system level model. The power activity information is calculated by applying the obtained profiling information to the power models. In order to evaluate power dissipation, we designed a C++ API called *PowerMeter* [23]. *PowerMeter* is automatically attached to each basic block of the system model to measure and monitor energy and delay. A graphical notion of basic block is shown in Fig. 5.

```
behavior A()
{
 void main()
 {
  int x;
 for(x=1;x<11;)
 {
   waitfor 1;
   x++;
 }
 }
};
behavior B()
{
 void main()
 {
  float y=100;
 do{
    waitfor 2;
    y/=2;
 } while(y>10);
  y++;
 }
};
behavior Main()
{
 A A1,A2;
 B B1;
 int main()
{
 par
 {
  A1.main();
  A2.main();
  B1.main();
 }
  return 0;
 }
};
```

behavior: A

executed : 2	**dynamic**	**static**
constant, *int*	44	3
identifier access, *int*	44	3
post-increment, *int*	20	1
less than, *int*	22	1
assignment, int	2	1
expression statement	20	1
for statement	2	1
wait for statement	20	1

behavior: B

executed: 1	**dynamic**	**static**
constant, *int*	12	3
identifier access, *float*	9	3
post-increment, *float*	1	1
greater than, *int*	4	1
division-assign, *float*	4	1
expression statement	5	2
do-while statement	1	1
wait for statement	4	1

behavior: Main

executed: 1	**dynamic**	**static**
constant, *int*	57	10
identifier access, *int*	44	6
identifier access, *float*	9	3
post-increment, *int*	20	2
post-increment, *float*	1	1
less than, *int*	22	2
greater than, *int*	4	1
assignment, *int*	2	2
division-assign, *float*	4	1
expression statement	25	4
do-while statement	1	1
for statement	2	2
return statement	1	1
par statement	1	1
wait for statement	24	3

Fig. 4. Dynamic and Static Profiling Report of Behaviors *Main, A* and *B*

```
PowerMeter PM_Global;

behavior A(){
  ...
  for (x=1;x<11;){
  ...                          basic block 1
  waitfor(10);
  consume(PM_Global, 15.1, 0.7);
  }
  .....
};
behavior B()
{
 .....
 do{
  ....                         basic block 2
  waitfor(12.3);
  consume(PM_Global, 3.7, .45);
 } while(y>10);
 ....                          basic block 3
 waitfor(1.4);
 consume(PM_Global, 1.2, .32);
 ....
};
```

(a) Global power meter

```
PowerMeter PM_A;
PowerMeter PM_B;
behavior A()
{
  ......
  for (x=1; x<11;){
  ....                         basic block 1
  waitfor(10);
  consume(PM_A, 15.1, 0.7);
  }
};
behavior B()
{
  .....
  do{
  ....                         basic block 2
  waitfor(12.3);
  consume(PM_B, 3.7, 0.45);
 } while(y>10);
 ....                          basic block 3
 waitfor(1.4);
 consume(PM_B, 1.2, .32);
 .....
};
```

(b) Power meter per behavior

```
PowerMeter PM_PE1;
PowerMeter PM_PE2;

behavior A_PE1()
{
  ....
  for (x=1;x<11;){
  ...                          basic block 1
  waitfor(10);
  consume(PM_PE1, 151, 0.7);
  }
  ...
};
behavior A_PE2()
{
  .....
  for (x=1;x<11;){
  .....                        basic block 2
  waitfor(10);
  consume(PM_PE2, 127, 1.2);
  }
 ...
};
behavior B()
{
  .....
  do{
  .....                        basic block 3
  waitfor(12.3);
  consume(PM_PE2, 3.7,0.45);
 } while(y>10);
 ...                           basic block 4
 waitfor(1.4);
 consume(PM_PE2, 1.2, .32);
 ....
};
behavior Main()
{
  A_PE1 A1;
  A_PE2 A2;
 .....
};
```

(c) Power meter per PE

Fig. 5. Power annotated system model with global powerMeter, power meter for every behavior, and power meter per PE

Power Meter per Behavior: In this case, a power meter is attached to each behavior of the design. During the design, the system designer may want to analyze each behavior in terms of computation, power and performance, compared to the rest of the system. This information may also be used to evaluate the peak power and power hungry behaviors of the design for possible thermal issues. This information allows the designer to balance the computation of the system and adjust the design at the system level, where altering and re-evaluations are quick and easy.

Power Meter per Architecture Component: Here power meters are assigned to each processing element of the design. In order to explore the design options, different architectures may need to be evaluated. Power dissipation of each component is monitored using the profiling information of the behaviors mapped to the component and shown by the corresponding meter.

Global Power Meter: In this option, a single power meter is attached to the entire ESL model, which measures the total power consumption of the design; this can be used to quickly determine if the design meets the overall power constraints of the target system.

Power meters are designed to capture dynamic and static power dissipation of their assigned component based on the power model. These components can be any CPU or hardware accelerator. In addition to the power meters, the *waitfor* [19] and *consume* functions for time and energy consumption are automatically annotated to each basic block of the design. The *consume* functions virtually spend energy and represent the energy consumption during execution.
Specifically, the *consume* function for a basic block *b* is:

$$consume(PowerMeter_i,\ DynEnergy_b,\ StaticEnergy_b)$$

where $PowerMeter_i$ can be one of the three types of power meters. $DynEnergy_b$ and $StaticEnergy_b$ are the dissipated dynamic and static energy, respectively. In order to employ the suggested default power models, the dynamic energy of each basic block is computed directly by applying the number of operations reported by the profiler to the power models:

$$DynEnergy_b = \sum_{j,k} OpCount_{jk} \times OpEnergy_{jk} \quad (1)$$

where the $OpCount_{jk}$ is the number of operations *j* with type *k* and $OpEnergy_{jk}$ is the energy consumption of that operation derived from the power model. For static values, the execution time of the block ($time_b$) is divided by the total static energy ($Energy_b$) spent at each behavior:

$$StaticPower_b = Energy_b \ / \ time_b \quad (2)$$

In Fig. 5, the power annotated specification model is presented for the profiled code shown in Section III-A. The specification model is presented for the three different formats of power meters.
At this step the profiling information and power models are applied to Equation 1 and Equation 2, and the resulting energy dissipation is automatically annotated to the model via the *consume* functions. Similarly, the performance results from [19] are annotated automatically to the design using *waitfor* functions. For inserting the power meters per PE, the annotation of behavior *A* is implemented differently from the other two types of power meters. Since behavior *A* is mapped to two different *PEs, PE1* and *PE2*, the *consume* functions use different power models, *PE1* and *PE2*. To resolve this problem, new behaviors with the same functionality as behavior *A* are inserted and named corresponding to their allocated *PEs, A_PE1* and *A_PE2*. All instantiations of behavior *A* are modified accordingly, as shown in Fig. 5(c). The duplications are performed automatically by our *PowerMeter* API during power meter insertion without any interaction by the user. The dissipated power and energy can be monitored both numerically and graphically using the available power functions of the *PowerMeter* API.

VI. Case Study: JPEG Image Encoder

The *PowerMeter* is implemented as described in the last section. Here the *PowerMeter* API is utilized for monitoring power consumption in JPEG, as a real-life application. The *PowerMeter* is applied at global, PE, and behavior levels.
Our case study uses the JPEG image encoder model shown in Fig. 7. The stimulus reads a BMP color image with 3216x2136 pixels and performs color-space conversion from RGB to YCbCr. Since encoding of the three color components (Y, Cb, Cr) is independent, our JPEG encoder performs the DCT, quantization and zigzag operations for the colors in parallel, followed by a sequential Huffman encoder at the end. The image is divided in 9 strips and fed in to JPEG model. The JPEG monitor collects the encoded data and stores it in the output file.

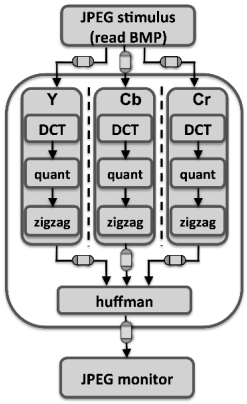
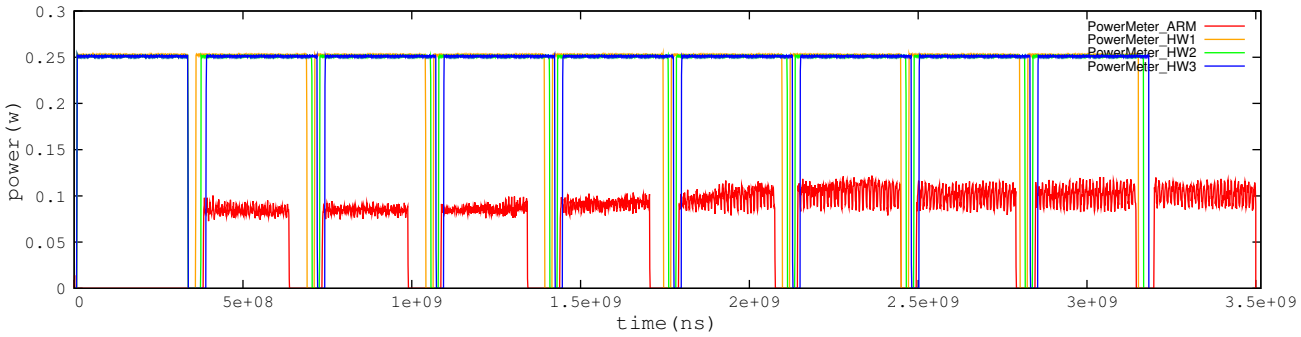
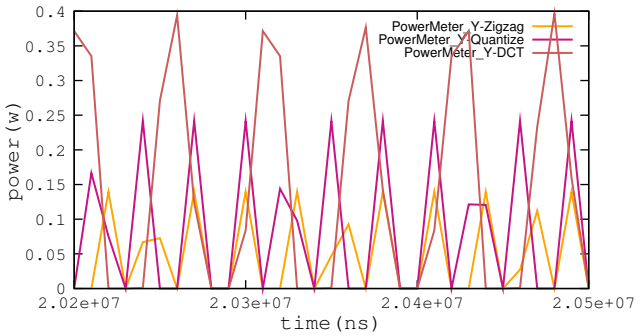


Fig. 7. JPEG image Encoder

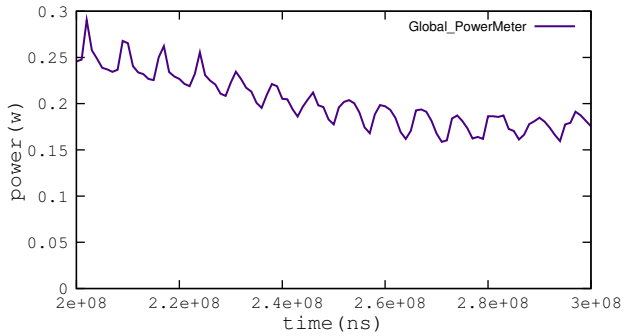
The JPEG model is examined on an ARM-based processor with 3 costum HW units. The 3 color components; Y, Cb, and Cr are mapped to separate HW units, along with their sub-behaviors (DCT, Quantize, Zigzag). All units are communicating through the AMBA BUS.
To comprehensively study power dissipation, we started with the specification model, and applied architecture, scheduling and communication refinement to the model, with increasing amount of implementation detail.
In order to control the size of power and energy log files, and adjust the precision of the analysis, the user can pick the sampling frequency. The user can also specify any simulation intervals to monitor as well. Moreover, *PowerMeter* supports merging the graphical reports or stacking up the power dissipation values in different PowerMeters over time.
Fig. 6(a) shows the power dissipation in each design elements over the whole simulation time with sampling frequency of 1ms. As it was expected from the JPEG model defined in Fig. 7, where the Y, Cb and Cr are running in parallel, the custom HW units are executing them in parallel as well. The Huffman encoder, which is mapped to the ARM processor, begins working once the Y, Cb and Cr color processing is over for a strip within all HW units. The encoding process is divided in to 9 different steps, through 9 sub-images, and the power dissipation intervals of design elements in Fig. 6(a) reflects the same behavior.
In Fig. 6(b) power dissipation in the Y-DCT, Y-Quantize, and Y-Zigzag, which are the leaf behaviors of Y is shown. The sampling frequency is 10μs. All these behaviors are mapped to HW1. As it shown, the user can easily pick any behavior



(a) Power dissipation at each PE



(b) Power dissipation at behavior level for behaviors mapped to HW1 (Y-DCT, Y-Quantize, Y-Zigzag)



(c) Average global power dissipation

Fig. 6. Power dissipation of JPEG Image Encoder visualized and optimized by *PowerMeters*

as well as any interval for power analysis. Using the power reports of each behavior, the user can easily verify the active behvaiors at each PE, computationally expensive behaviors, the woking intervals of each behavior, and modify the design model rapidly if needed. The *PowerMeter* allows merging the power reports in to one graph based on the user selections.

The global PowerMeter monitors the average power dissipation of the whole system. Fig. 6(c) represent the average power for selected simulation interval in JPEG application. The PEs, global and behaviors power dissipation reports support monitoring and analysing the system for power and performance, and provide a platform for initiating power optimization.

The power estimation API can also provide the energy dissipation graph for each power meter over time. For example, Fig. 8 displays the energy dissipation graph of a global power meter in JPEG with 9 sub-image. As shown in the diagram, for each image, the energy dissipation increases during the complex encoding process for each sub-image.

VII. EXPERIMENTAL RESULTS

In this section, we describe our experimental results and analyse the speedup, *fidelity*, and accuracy of *PowerMeter* with real life applications. We implemented the profiler and *PowerMeter* API, as well as a platform for automated back annotation of power meters, *consume* functions, and *waitfor*. Thus, when the specification model is ready, the system designer can instantly evaluate the design by simply mapping the ESL system model to architecture components and set the *PowerMeter* granularity.

For our experiments, we choose a JPEG image encoder application, a MP3 audio decoder and a H.264 video encoder and decoder application, all specified in SpecC SLDL. To evaluate the *fidelity* and accuracy of the proposed power estimation method, we modified the ISS model of JPEG encoder to simulate on SimpleScalar [24] and measured the energy and power consumption using the cycle-accurate SimPanalyzer [21] for the ARM based architecture. The MP3 and H.264 examples are simulated using the SNIPER 5.2 [22] simulator and their power is evaluated using the embedded McPAT [1] against the Intel Nehalem architecture. We also applied these applications to our power estimator. All results are measured on a host PC with a 4-core CPU (Intel(R) Core(TM)2 Quad) at 3.0 GHz.

Our results are presented in Table II. The comparison execution time shows significant speedup in *PowerMeter*, in comparison to the cycle accurate SimPanalyzer, as well as SNIPER, with the embedded McPAT power estimator as an architectural level simulator. The overall speedup shows that *PowerMeter* is about one order of magnitude faster than the alternatives. Furthermore, the *PowerMeter* is a system level power estimator and, with its low execution time, provides a practical platform for design space exploration. As shown in Table II for computationally expensive applications, a low-level simulator is not an applicable solution. For instance, power estimation of the H.264 application with 10 frames did not complete after two hours of runtime while its simulation

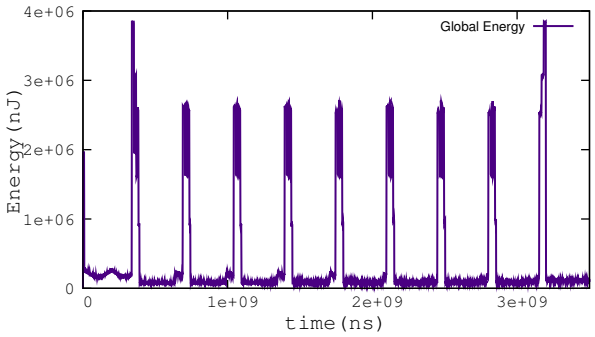
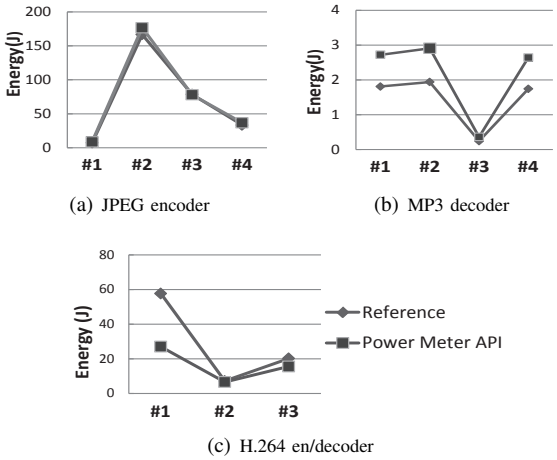


Fig. 8. JEPG Encoder energy consumption diagram over time



(a) JPEG encoder (b) MP3 decoder

(c) H.264 en/decoder

Fig. 9. Power Evaluation Fidelity for System Level and RTL

ended in less than 6 minutes at the system level.

The error reports of simulated applications are as expected in Fig. 1. The absolute accuracy is not as good as a low level estimator and goes up to 50% in MP3 application. This is simply owing to the fact that the accuracy of the estimated energy dissipation highly depends on the information captured during lower level simulations. Another reason is the accuracy of power models and the fact that the accuracy of power estimation directly depends on the accuracy of these models. In the applied power models, due to the fact that the models are not data-dependent and that even minor errors within the estimated power value for each operation will accumulate in total power estimation calculations, this results in higher errors specifically in large applications. For instance, in the MP3 example, the accuracy errors for all four audio streams are uniformly very close (only 1% difference). This clearly reflects the error in the original power model. However, the values of accuracy error using scaled energy for each sample application show the absolute maximum error is 3.5%, which reveals the high *fidelity* of our *PowerMeter* estimator. The *fidelity score* for the system models are also calculated using the Equation 3 proposed in [25].

$$F_{score}(model_m)=100 \times \frac{2}{n(n-1)} \sum_{\substack{i,j=1 \\ i<j}}^{n} \mu_{ij} \qquad (3)$$

Here, *fidelity score* F_{score} is calculated for system model *m* with *n* different design samples using predictor fidelity function μ. This function compares the referenced(*i*) and generated(*j*) design samples under multiple criteria, and returns 0 or 1 accordingly. As shown in Table II, this confirms the perfect *fidelity score* of 100 in all three of the applications.

The relative accuracy and high level of fidelity are the main achievements of our system level power estimator, which make SCE *PowerMeter* a solid starting point for power evaluations at the early system level.

Fig. 9 shows the results of *PowerMeter* API against McPAT and SimPanalyzer power estimators. As demonstrated, all pairs of curves, which represent energy values of different inputs for each application, all show the same shape, confirming the high fidelity of our approach.

VIII. SUMMARY AND CONCLUSIONS

In this paper, we have addressed power as a main challenge in the system design cycle. Power estimation at ESL is a method for evaluating power rapidly at the early stages of the design process. In order to have an ideal power estimator with high fidelity, it is essential to incorporate all possible information on the target system. In this work, we demonstrated that *PowerMeter*, a system level power estimator that automatically profiles the power activity of a ESL model, integrates a power model, and back-annotates the design with energy dissipation and performance functions.

The *PowerMeter* allows for fast power evaluation of functional blocks and integrated IPs without manual intervention. Power dissipation reports for JPEG image encoder application are presented as a case study.

Our experimental results show significant increase in power evaluation speed, up to one order of magnitude, with a high degree of fidelity. These results show that design space exploration at the system level can be easily equipped with power and performance estimation, and power optimization approaches can be initiated early and rapidly at the system level.

REFERENCES

[1] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "McPAT: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pp. 469–480, IEEE, 2009.

[2] "Design compiler, synopsys inc," 2000.

[3] S. Ravi, A. Raghunathan, and S. Chakradhar, "Efficient RTL power estimation for large designs," in *VLSI Design, 2003. Proceedings. 16th International Conference on*, pp. 431–439, IEEE, 2003.

[4] C. X. Huang, B. Zhang, A.-C. Deng, and B. Swirski, "The design and implementation of PowerMill," in *Proceedings of the 1995 international symposium on Low power design*, pp. 105–110, ACM, 1995.

[5] D. D. Gajski, F. Vahid, S. Narayan, and J. Gong, "Specification and design of embedded systems," 1994.

[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.

TABLE II
EXPERIMENTAL RESULTS FOR JEPG, MP3 AUDIO DECODER AND H.264 VIDEO CODEC

Model		Time(sec)		Speed Up	Energy(J)		Error	Scaled Energy		Accuracy Error	Fidelity Score
		simPanalyzer	*PowerMeter*		*simPanalyzer*	*PowerMeter*		*simPanalyzer*	*PowerMeter*		
JPEG Enc.	Image1	52.7	4.04	13x	7.3	8.76	+12.00%	0.0%	0.0%	+0.0%	100
	Image2	634	4.12	154x	166.7	176.73	+6.20%	100.0%	100%	+0.0%	
	Image3	287	4.08	70x	78.6	78.05	+1.60%	44.7%	41.3%	-3.5%	
	Image4	134	4.07	33x	33.5	36.81	+9.5%	16.4%	16.7%	+0.3%	
		McPAT	*PowerMeter*		*McPAT*	*PowerMeter*		*McPAT*	*PowerMeter*		
MP3 Dec.	Audio1	645.1	135.05	4.8x	1.81	2.72	+50.30%	92.4%	92.5%	+0.1%	100
	Audio2	682.6	137.68	5.0x	1.94	2.91	+50.30%	100.0%	100.0%	+0%	
	Audio3	118.1	18.21	6.5x	0.24	0.36	+50.20%	0.0%	0.0%	+0.0%	
	Audio4	612.1	128.35	4.8x	1.74	2.64	+51.50%	88.2%	89.4%	+1.2%	
H.264 Codec	Video1	9967.54	479.85	20x	57.72	27.02	-53%	100.0%	100%	+0.0%	100
	Video2	3209.2	70.23	45x	7.16	6.56	-7.9%	0.0%	0.0%	+0.0%	
	Video3	13942.63	191.29	70x	20.21	15.56	-23.00%	100%	100%	+0.0%	
	Video4	+2hr	353.53	-	-	33.74	-	-	2.17	-	

[8] X. Dong, C. Xu, Y. Xie, and N. P. Jouppi, "Nvsim: A circuit-level performance, energy, and area model for emerging nonvolatile memory," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 31, no. 7, pp. 994–1007, 2012.

[9] A. B. Kahng, B. Li, L.-S. Peh, and K. Samadi, "ORION 2.0: a fast and accurate NoC power and area model for early-stage design space exploration," in *Proceedings of the conference on Design, Automation and Test in Europe*, pp. 423–428, European Design and Automation Association, 2009.

[10] "PowerMixer User Manual." http://www.tinnotek.com.tw.

[11] D. Brooks, V. Tiwari, and M. Martonosi, *Wattch: a framework for architectural-level power analysis and optimizations*, vol. 28. ACM, 2000.

[12] W. Ye, N. Vijaykrishnan, M. Kandemir, and M. J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool," in *Proceedings of the 37th Annual Design Automation Conference*, pp. 340–345, ACM, 2000.

[13] M. Giammarini, M. Conti, and S. Orcioni, "System-level energy estimation with Powersim," in *Electronics, Circuits and Systems (ICECS), 2011 18th IEEE International Conference on*, pp. 723–726, IEEE, 2011.

[14] S.-K. Rethinagiri, O. Palomar, O. Unsal, A. Cristal, R. Ben-Atitallah, and S. Niar, "Pets: Power and energy estimation tool at system-level," in *Quality Electronic Design (ISQED), 2014 15th International Symposium on*, pp. 535–542, IEEE, 2014.

[15] K. Grüttner, P. A. Hartmann, K. Hylla, S. Rosinger, W. Nebel, F. Herrera, E. Villar, C. Brandolese, W. Fornaciari, G. Palermo, *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, pp. 966–980, 2013.

[16] 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.

[17] 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.

[18] 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.

[19] L. Cai and D. Gajski, "Introduction of design-oriented profiler of SpecC language," *Center for Embedded Computer System*, 2002.

[20] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: a first step towards software power minimization," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 2, no. 4, pp. 437–445, 1994.

[21] T. Mudge, T. Austin, and D. Grunwald, "SimPanalyzer: the simplescalar-arm power modeling project," 2004.

[22] T. E. Carlson, W. Heirman, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 52, ACM, 2011.

[23] Y. Samei and R. Dömer, "PowerMonitor: A Versatile API for Automated Power-Aware ESL Design," in *Specification & Design Languages (FDL), 2014 Forum on*, IEEE, 2014.

[24] D. Burger and T. M. Austin, "The SimpleScalar tool set, version 2.0," *ACM SIGARCH Computer Architecture News*, vol. 25, no. 3, pp. 13–25, 1997.

[25] F. J. Kurdahi, D. D. Gajski, C. Ramachandran, and V. Chaiyakul, "Linking register-transfer and physical levels of design," *IEICE Transactions on Information and Systems*, vol. 76, no. 9, pp. 991–1005, 1993.