Trace-driven System-level Power Evaluation of System-on-a-chip Peripheral Cores

Tony D. Givargis, Frank Vahid*
Department of Computer Science and Engineering
University of California, Riverside, CA 92521
{givargis,vahid}@cs.ucr.edu

Jörg Henkel
C&C Research Laboratories, NEC USA
4 Independence Way, Princeton, NJ 08540
henkel@ccrl.nj.nec.com

(*Also with the Center for Embedded Computer Systems at UC Irvine.)

Abstract

Our earlier work for fast evaluation of power consumption of general cores in a system-on-a-chip described techniques that involved isolating high-level instructions of a core, measuring gate-level power consumption per instruction, and then annotating a system-level simulation model with the obtained data. In this work, we describe a method for speeding up the evaluation further, through the use of instruction traces and trace simulators for every core, not just microprocessor cores. Our method shows noticeable speedups at an acceptable loss of accuracy. We show that reducing trace sizes can speed up the method even further. The speedups allow for more extensive system-level power exploration and hence better optimization.

Keywords

System-on-a-chip, low power system design, intellectual property, cores, system-level modeling, parameterized architectures.

1. Introduction

Minimizing power consumption of a system-on-a-chip (SOC) is an important design goal, especially with respect to the fast growing market of complex mobile computing and communication devices. Those systems are typically designed as systems-on-a-chip in order to reduce the overall costs as well as minimizing the time to market. An SOC may consist of numerous parameterized cores, including microprocessors, caches, co-processors and peripherals like DMA controllers, UARTs, codecs, floating-point units, etc. A parameter may represent a variable feature of a core, like bit-width, buffer size, alternative algorithms, and so on. We denote a selection of parameters for all cores in an SOC as a system configuration. Power must be evaluated for hundreds or thousands of possible system configurations of such core-based systems in order to find the design point that best satisfies given constraints. Unfortunately, accurate gate-level power evaluation is too slow for large SOC's, i.e., > 100 million transistors, requiring an infeasible amount of computation time to evaluate just one single system configuration.

Consequently, researchers have focused on fast power evaluation techniques. Most initial work concentrated on the register-transfer-level (RTL), but this level may still be too slow for a large SOCs. Therefore, more recent work has focused on system-level power evaluation. Much emphasis has been on instruction-level power evaluation for microprocessors, and on trace-based cache evaluation. Recent work has also focused on an instruction-based approach for evaluating general cores (not just programmable microprocessors) [4], demonstrating high accuracy with computation times much faster than RTL approaches. The key to that work was treating any core, such as a UART, as executing a small set of high-level instructions. Power-per-instruction could then be pre-determined, as is done for microprocessors, and then such power would be accumulated during a system-level simulation of an executable specification representing an SOC.

In this paper, we extend this previous instruction-based general core approach to execute even faster, by using instruction traces and trace simulators for every core. The approach is harmonious with existing approaches for microprocessors and caches, and complements existing system-level modeling standards. We show noticeable speedups at an acceptable loss of accuracy[1].

2. Previous work

Previous power evaluation work has been done at various abstraction levels. Logic-level, or gate-level, approaches simulate a gate-level design, and calculate power by considering switching activity of internal nodes [8][18]. Logic-level approaches are orders of magnitude too slow to be used in SOC configuration exploration, requiring days to obtain data for even one configuration.

RTL power evaluation operates at a higher level of abstraction, modeling power consumption of more abstract circuit components, such as adders and multipliers. Simulation is performed at the RT-level and power is obtained by using power macro-models. The approaches taken here can be divided into two categories, macro-modeling using table-lookup techniques and analytical models. Using table-lookups, each component is modeled via an N-variable characterization (input density, output density, switching-probability, etc.) of its power consumption [5][1]. An N-dimensional lookup table is used to lookup the power consumption of an RTL component during

[1] Hence, the absolute accuracy of the power/energy number is secondary in importance to fidelity, or relative magnitudes, of the various estimated numbers with one another.

0-7803-6633-6/01/$10.00 ©2001 IEEE.

simulation. Similarly, analytical models have been devised that compute power consumption of an RTL component given the actual input patterns or some form of input pattern characterization [10][14]. Lookup-tables and the coefficients of the analytical models are often derived from the gate-level circuit structure or lower-level power evaluation and simulation. While RTL power evaluation is shown to be accurate to within 5% of actual power consumption [11], it too suffers from simulation times too slow for extensive system-level exploration. Furthermore, just synthesizing an RTL design for a given configuration can take hours, independent of simulation.

Previous behavioral-level approaches seek to estimate power of a behavioral HDL description before a synthesized design is obtained. An abstract notion of physical capacitance and switching activity is used. Switching is estimated using entropy from circuit input to circuit output by quadratic or exponential degradation [13][14]. While such behavioral approaches can provide fast evaluation of power for custom designs, they will not be nearly as accurate for cores as approaches that take advantage of the fact that cores can be pre-designed and pre-analyzed.

Several researchers have focused on fast system-level models for cache, memory and bus power consumption [2][3], consisting mostly of equations that compute power consumption as a function of usage/traffic and core parameters. A system-level approach for power consumption that takes into consideration the interdependencies of various cores has been proposed in [11] using instruction traces. Work has been done to evaluate power consumption of microprocessor cores. One approach, instruction-level power modeling, is proposed by [17]. Given a program execution trace, energy is computed as the sum of the energy consumed by each instruction that is executed, circuit state energy consumed when a particular instruction is followed by another, and energy consumed by *other effects* such as stalls and cache misses. An even more general approach is described in [16]. An instruction-set simulator is extended with equations to compute power not only for the microprocessor, but also for cache, memory, bus, and even a DC-DC converter, with good accuracy results. These approaches can be sped up further by techniques in [6], by deriving a shorter program trace that results in equal power dissipation when compared to the original trace. In [16], techniques for speeding up simulation for power purposes are presented, in which code regions that give similar power upon repeated execution can be skipped, using the previously-seen power value. All of these techniques have emphasized microprocessors, memories and buses, while ours is generalized for all cores including peripheral cores, and thus the techniques are complementary.

3. Trace-driven core power evaluation

3.1 System simulation approach

The work presented in this paper is an extension to a system level simulation approach given in [4]. Their system level simulation approach works as follows. First, a core provider selects a set of appropriate instructions covering the possible actions of a core. Then they perform gate-level power analysis

Figure 1: Augmenting the functional model of an instruction for power estimation.

```
// functional implementation before here
power-mode = NextPowerMode(power-mode, current-inst);
p = this core's current parameter values;
m = power-mode;
i = current-instruction's identification number;
d = data passed into the current-instruction;
if( i is independent of its data ) {
   total-power += PowerLookupTable[p][m][i];
}
else if( i is statistically dependent on its data ) {
  total-power +=
  PowerLookupTable[p][m][i][[GetStats(d)];
}
else if( i is dependent on it data ) {
   total-power += PowerLookupTable[p][m][i][d];
}
```

to construct power lookup tables for each instruction, and create a system-level core model, written in a high-level language like C++ or Java, that utilizes the lookup-tables for power evaluation through an executable specification. The core user[2] connects the system-level core models, executes the whole system (executable specification) and thus obtains power data after a system execution/simulation. The core provider may need to provide data for different technologies, or provide a means for a core user to re-compute power-per-instruction data for different technologies.

Therefore, the core provider must first break the core's functionality into a set of appropriate instructions. Given an RTL model of a core called *C*, one first determines the system-level instructions $i_1, i_2, i_3, \ldots i_n$, of *C*. As with the instructions of an instruction-set microprocessor, each instruction i_j operates on some input data and produces some output data. During this step, the core provider must also determine the dependency of an instruction's power on the instruction's data, the inter-instruction power dependencies, and the different power *modes* in which the core may operate.

The second task consists of using gate-level simulation to obtain per-instruction power data for the lookup-tables. Given an RTL model of a core called *C*, its instructions $i_1, i_2, i_3, \ldots i_n$, and its modes $m_1, m_2, m_3, \ldots m_k$, one follows a procedure that gives a methodical way of creating a set of testbench models. When simulated at gate-level, these testbench models capture the power consumption of a particular instruction, in a

[2] We use the terms "core user" and "system designer" interchangeably. In contrast, the term "core developer" is used to denote the designer of a core who has no specific application in mind. Rather, a core developer's goal is to design a core such that it can be used in as many as possible different applications.

Figure 2: (a) Functional model, (b) trace-driven model.

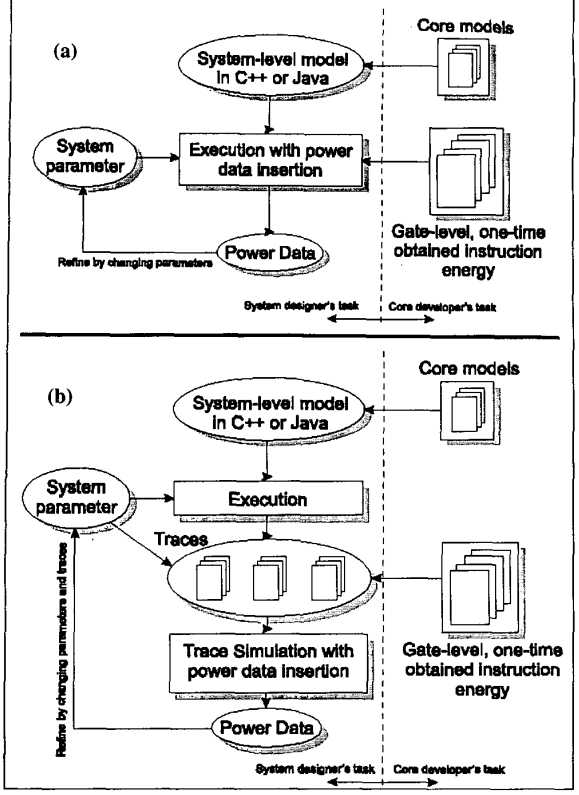


Figure 3: Trace-size reduction approaches: (a) full trace with instructions + data (unreduced), (b) reduced trace via characterized-data, (c) reduced trace via instructions only (d) reduced trace via instruction-frequency.

(a)

Reset	--
Quantize	$P_1, P_2, \ldots, P_{64}$
IDCT	$P_1, P_2, \ldots, P_{64}$
Quantize	$P_1, P_2, \ldots, P_{64}$
IDCT	$P_1, P_2, \ldots, P_{64}$

(b)

Reset	--
Quantize	.80
IDCT	.72
Quantize	.93
IDCT	.63

(c)

Reset	--
Quantize	--
IDCT	--
Quantize	--
IDCT	--

(d)

Reset	*1
Quantize	*2
IDCT	*2

particular mode with a particular parameter setting. Note that previous RTL power estimation approaches, such as macro-modeling, could be used to speed up this task.

The next step is to develop a system-level model of each core that enables rapid power evaluation when executed. Given an RTL model of a core called *C*, its instructions $i_1, i_2, i_3, \ldots i_n$, and its modes $m_1, m_2, m_3, \ldots m_k$, one implements a functional model of *C* in terms of its instructions. If using method-calling objects [19], the interface to the object representing *C* would have the instructions $i_1, i_2, i_3, \ldots i_n$, as methods and the instruction's input/output data as parameters to the corresponding methods. To each object-oriented model, one adds two data objects, called total-power, initialized to zero, and power-mode, initialized to *reset*. One then augments the implementation of each method of *C*'s system level model with the code outlined in Figure 1.

The above three steps, performed by the core developer, may take days to complete, forming part of the months required to develop the core. The core user connects the core models and simulates. Simulation of a complete SOC, using system-level models, may take on the order of seconds or minutes. Thus, hundreds or thousands of configurations can be evaluated. The top-level simulation model will be designed to output the value of the total-power variable, for each core of the system, at the end of each simulation. The sum of these total-power values represents the system-level estimate of the system's power consumption for a given configuration of its parameters.

3.2 Trace-simulator approach

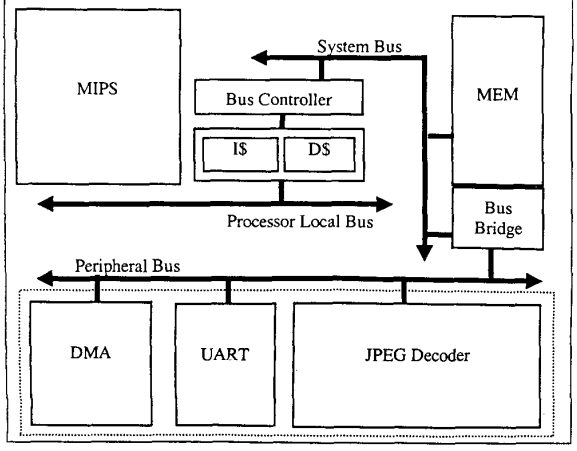
Provided the system simulation approach outlined in the previous sections, we define a *trace,* with respect to a core, to be a sequence of instruction/data items that are executed by that core during its functional simulation. We extend the above simulation-based approach by converting the functional models of the cores to non-functional (or partially functional) models. These models operate on a *trace.* We refer to such nonfunctional models as *trace simulators.* Processor architects use similar trace simulators, such as Dinero, for evaluating cache performance.

Figure 2 shows the functional-simulation-based approach as well as the trace-simulator-based approach. Using trace simulators, a core user simulates a system once to obtain the trace files for each core. These trace files are subsequently processed using trace simulators to obtain power and explore various core parameter effects. Trace-driven simulators are significantly faster than full functional simulators.

We now describe how to construct these trace-driven simulation models for general peripheral cores. First, given a system level functional model of a core *C*, the core developer augments the implementation of each method in that model with code that will append to a trace-file a unique id for that instruction and the corresponding input data. When executed, such a model will output a set of traces, one per each core in the system, that is subsequently used by trace simulators as described next.

Given an RTL model of a core called *C*, its instructions $i_1, i_2, i_3, \ldots i_n$, and its modes $m_1, m_2, m_3, \ldots m_k$, we implement a nonfunctional model of *C* in terms of its instructions. If using

Figure 4: Target architecture.



method-calling objects, the interface to the object representing *C* would have the instructions, i_1, i_2, i_3, ... i_n, as methods and the instruction's input/output data as parameters to the corresponding methods. To each object-oriented model, we add two data items: *total-power*, initialized to zero, and *power-mode*, initialized to *reset*. The implementation of each function consists of the code fragment presented in Figure 1, but excluding the functional implementation.

Each core's object-oriented model is then designed to read the corresponding trace file and execute the instructions of it accordingly. At the end, each core will output the value of its total-power variable. The sum of these total-power values represents the system-level estimate of the system's power consumption for a given configuration of its parameters.

3.3 Trace-size reduction

We can further speed up the power evaluation time for cores by reducing the size of the trace files, therefore reducing the processing time required to evaluate power consumption, as depicted in Figure 3. The technique is similar in idea to those in [6] intended for microprocessors, but simpler (since microprocessor instruction traces are more complex). In that work, they first capture a characteristic profile, which is a type of trace that includes instruction mix, memory references, cache misses, branch predictions, etc. Then, they use mixed integer linear programming and heuristics to transform the trace into a synthesized trace that exhibits the same power requirements but is much shorter.

Here, we outline several of our trace-size reduction approaches for cores. They are to be compared with the *full trace* approach of the previous section, shown in Figure 3(a), in which the traces store each instruction along with its complete input data.

In the *reduced trace via characterized-data* approach, shown in Figure 3(b), rather than storing complete parameter data, we store a statistical characterization of that data. For example, we can store the data-density, defined as the ratio between the number of bits that are set to the total number of bits. Density has been shown to be a good predictor of power in many components, and our own experiments support this.

In the *reduced trace via instructions only* approach, illustrated in Figure 3(c), we store the instruction only, without any parameter data. We can take this approach if we determine that power consumption is mostly independent of an instruction's data.

Note that we can apply the above trace reductions to the entire trace file, i.e., all instructions, or to selected instructions. Thus, Figure 1 shows code that can use a different method for each type of instruction.

Lastly, in the *reduced trace via instruction-frequency* approach, we combine a sequence of instructions that are identical or have identical power consumption into a single instruction augmented with a frequency value. An example is given in Figure 3(d). We could further annotate each instruction with a statistical characterization of the data accompanying the combined instructions. The instruction-frequency approach remains an area for future work, and could be further extended in the direction of [6].

4. Experiments

In order to evaluate our trace driven approach, we have experimented with several examples and compared metrics of simulation time, trace file size and power accuracy. We compared our full and reduced trace simulation approaches with full system-simulation, as well as with gate-level simulation. Below, we will outline our architecture, describe our experimental setup, and summarize data obtained from several examples.

4.1 Architecture

Our parameterized system-on-a-chip architecture is depicted in Figure 4. The architecture works as follows. A MIPS R2000 processor and instruction and data caches communicate over a high-speed processor-local bus. The on-chip memory and direct memory access (DMA) controller cores are connected to the system bus, which in turn is bridged to the processor-local bus via a bus controller. Universal Asynchronous Receiver and Transmitter (UART) and JPEG decoder cores are connected to the peripheral bus, which is bridged to the system bus. Both the UART and JPEG decoder cores are DMA capable. The DMA controller is capable of transferring data between peripheral cores and memory without the intervention of the processor. The processor can run concurrent to the DMA until a cache miss occurs at which point the processor is blocked waiting for the DMA transfer to complete. The UART and JPEG decoder cores in our architecture are parameterized. The UART core's transmitter/receiver buffer sizes can each be set to one of 2, 4, 8, or 16 bytes. The JPEG decoder core's pixel resolution can be set to one of 10 or 12 bits. This architecture is used to implement a JPEG image decode accelerator. JPEG images are input serially through the UART, transferred via the DMA to memory, Huffman decoded by the MIPS, transferred from memory to the JPEG decoder and back to the UART to be output to the host device. Most of these operations take place in a pipelined fashion for maximum throughput. We have RTL

Table 1: Trace file size, evaluation time and power result.

Param.	Trace File Size (Kb)			Power Evaluation CPU Time (sec)					Power (mJ)			
	Ftrc	Rtrc	Rtrci	Gate	Sys	Ftrc	Rtrc	Rtrci	Gate	Ftrc, error	Rtrc, error	Rtrci, error
Buff. Size	**UART-XMIT**											
2	7.2	-	3.1	123K	22.8	17.1	-	10.9	76.0	79.1,4.1%	-	80.0,5.2%
4	6.7	-	2.6	145K	21.4	16.9	-	10.8	81.2	84.9,4.6%	-	84.3,3.8%
8	6.4	-	2.3	155K	21.3	16.3	-	10.8	97.1	99.8,2.8%	-	100,3.0%
16	6.3	-	2.2	164K	21.7	15.8	-	10.9	113	115,1.8%	-	115,1.8%
Pixel Size	**JPEG-CAR**											
10	32	3.6	.5	290K	48	26	4.9	4.6	420	443,5%	451,7%	491,17%
12	39	3.6	.5	330K	49	27	5.1	4.6	531	569,7%	576,8%	632,19%
Pixel Size	**JPEG-EARTH**											
10	8.1	.90	.10	130K	20	11	2.3	2.1	235	252,7%	264,12%	289,23%
12	9.7	.90	.10	170K	21	12	2.5	2.1	297	308,4%	323,9%	355,20%

synthesis models for all three of the parameterized cores. We implemented system-level simulation models, as described in the previous section, for all components in the architecture. Our system-level simulation models can operate as fully functional models or as non-functional models. Each functional model can during execution generate full traces of instructions + data, reduced traces of instructions-only, or reduced traces of instructions + data-characterization (i.e., data-density). Likewise, our non-functional models can read in any of the three types of traces and output power consumption. However, the focus of this work is the peripheral cores, such as the DMA, UART and JPEG Decoder, depicted in the shaded area of Figure 4.

4.2 Experimental setup

For our experiments, we selected two cores, a UART and a JPEG decoder, and three small applications, XMIT, CAR and EARTH. XMIT is an application that transmits 2048 bytes of a Huffman encoded photograph using the UART core. The CAR example uses the JPEG core to decode a 320x80-pixel photograph of an automobile. Likewise, the EARTH example uses the JPEG core to decode an 80x80-pixel photograph of planet Earth. The CAR and EARTH examples differ in that the photographs that they process have very different levels of detail. In the case of the CAR example, the photograph is very detailed using near 256 different colors. The EARTH example, however, is less detailed and uses less than 100 colors. This difference in detail has an impact on the energy consumption of the system.

We have used our architecture to evaluate power consumption for each example for various core configurations, and have compared the results to gate-level power estimations done by Synopsys tools. Only the UART and JPEG cores are simulated at the gate-level, since those are the cores we are focusing on, and the remainder of the architecture is simulated at a behavioral VHDL level. For our system-level simulation model, all components (including the MIPS) of the architecture are simulated as an executable binary obtained from the C++ models. Gate-level power evaluation time includes synthesis using the Synopsys Design Compiler from RTL to gate, simulation at gate level using the Synopsys VHDL simulator, followed by cycle-by-cycle switching activity based power evaluation using the Synopsys Power Compiler.

4.3 Experimental results

We first experimented with the XMIT example. The results of this and all other examples are provided in Table 1. For each parameter value, in this case the possible buffer sizes for the UART, we measured the sizes of the trace files for the *full-trace* and the *reduced trace via instructions only* approaches. Also, we compared the CPU time required to evaluate power consumption, comparing gate-level simulation (*gate*), full system simulation (*sys*), full trace simulation (*ftrc*) and reduced trace simulation via instructions only (*rtrci*). We see that the full trace file was on the average 2.6 times larger than the instruction-only reduced trace files. We also see that, compared to gate-level power evaluation, full system simulation gave a speedup of 6800, while reduced trace-simulation using instructions gave a speedup of 13500.

Considering power results for the XMIT example, the full trace with instruction + data gave an average error of 3.3% compared to gate-level power data. Results for full system simulation were the same. The reduced trace with instructions-only gave an error of only 3.5%. Hence, for the UART core, we can see that the reduced trace with instructions-only gives excellent accuracy with the best speedup.

Next, we experimented with the CAR example. For each parameter value, in this case the possible pixel resolutions for the JPEG core, we measured the sizes of the full trace (*ftrc*), reduced trace via characterized data (*rtrc*), and reduced trace with instructions only (*rtrci*). We noted that the trace using characterized data was nearly an order of magnitude smaller than the full trace file. Compared with gate-level simulation

time for CAR, the system simulation gave a speedup of 6000, full trace a speedup of 12,000, reduced trace using data characterization of 62,000, and reduced trace using instructions only of 67,000.

Considering our power results for the CAR example, a full trace approach gave 6% average error compared to gate-level. A reduced trace approach using characterized data (i.e., data density) gave 7.5% average error. Using reduced traces with instructions only, the error was 18%. Thus, this example shows the usefulness of the characterized data approach to trace reduction, which gives the nearly the best performance with only minor loss of accuracy. We also see that for this compute-intensive core, the speedup of the reduced trace versus the system simulation approach is much greater than for the UART, in fact, there is an order of magnitude difference between the reduced trace and the system simulation.

Our last example, EARTH, is similar to CAR. Here too, the reduced trace was an order of magnitude faster than system simulation, with the characterized data approach giving the best accuracy/performance tradeoff. Once again, we see that the data characterization approach to trace reduction gives the best performance/accuracy tradeoff for the JPEG core.

Thus, for each peripheral core, a core designer must decide on which trace file approach to use. We found that for the UART core, a reduced trace file using instructions only would give high accuracy and fast simulation times. For the JPEG core, a reduced trace file using characterized data would give best accuracy/performance tradeoff.

A limitation of a trace-based approach is that changing certain parameters of certain cores may change the instruction trace for those cores or the cores they interact with. Thus, a trace-based approach must include an awareness of which parameters require re-simulation of the system-level model to generate new traces.

5. Conclusions

Previous work showed that an instruction-based system-level core simulation approach could evaluate core power 3 orders of magnitude faster than gate-level simulation. In this paper, we demonstrated that an additional order-of-magnitude speedup can be obtained by using the instruction-trace based techniques described, thus permitting even more extensive system-level exploration for low-power design of system-on-a-chip architectures exceeding 100 million gates.

6. Acknowledgement

This work was supported by the National Science Foundation (CCR-9811164), (CCR-9876006) and a Design Automation Conference Graduate Scholarship.

7. References

[1] M. Barocci, L. Benini, A. Bogliolo, B. Ricco, G. De Micheli. Lookup Table Power Macro-Models for Behavioral Library Components. Design Automation and Test In Europe, March 1998.

[2] R. J. Evans, P.D. Franzon. Energy Consumption Modeling and Optimization for SRAMs, IEEE Journal of Solid-State Circuits, Vol. 30, No. 5, pp. 571-579, 1995.

[3] T.D. Givargis, J. Henkel, and F. Vahid. Interface and Cache Power Exploration for Core--Based Embedded System Design. ICCAD 1999.

[4] T.D. Givargis, F. Vahid, J. Henkel, A Hybrid Approach for Core-Based System-Level Power Modeling, ASP-DAC, 2000.

[5] S. Gupta, F. Jajm. Power Macromodeling for High Level Power Estimation. Design Automation Conference, June 1997.

[6] C.T. Hsieh, M. Pedram, H. Mehta, F. Rastgar. Profile Driven Program Synthesis for Evaluation of System Power Dissipation. Design Automation Conference, June 1997.

[7] S.M. Kang. Accurate Simulation of Power Dissipation in VLSI Circuits. IEEE Journal of Solid-State Circuits, vol. CS21, no. 5, pp. 889-891, October 1986.

[8] T.H. Krodel. PowerPlay - Fast Dynamic Power Estimation Based on Logic Simulation. IEEE International Conference on Computer Aided Design, pp. 96-100, Oct. 1991.

[9] M. Lajolo, A. Raghunathan, S. Dey, L. Lavagno, A. Sangiovanni-Vincentelli. Efficient Power Estimation Techniques for HW/SW Systems, IEEE VOLTA, 1999.

[10] P. Landman, J. Rabaey. Architectural Power Analysis: The Dual Bit Type Method. IEEE Transactions on VLSI Systems, vol. 3, no. 2, June 1995.

[11] Y. Li, J. Henkel, A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems, IEEE/ACM 35th. Design Automation Conference (DAC) 1998, pp.188-193, 1998

[12] E. Macii, M. Pedram. High-Level Power Modeling, Estimation, and Optimization. IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, vol. 17, no. 11, November 1998.

[13] D. Marculescu, R. Marculescu, M. Pedram. Information Theoretic Measures for Power Analysis. IEEE Transactions on Computer Aided Design, vol. 15, no. 6, pp. 599-610, 1996.

[14] H. Mehta, R. Owens, M.J Irwin. Energy Characterization Based on Clustering. Design Automation Conference, June 1996.

[15] M. Nemani, F. Najm. Toward a High Level Power Estimation Capability. IEEE Transactions on Computer Aided Design, vol. 15, no. 6, pp. 588-598, 1996.

[16] T. Simunic, L. Benini, G. De Micheli. Cycle-Accurate Simulation of Energy Consumption in Embedded Systems. Design Automation Conference, June 1999.

[17] V. Tiwari, S. Malik, A. Wolfe. Power Analysis of Embedded Software: A First Step Toward Sofware Power Minimization. IEEE Transactions on VLSI Systems, vol. 2, no. 4, pp. 437-445, 1994.

[18] R. Tjarnstorm. Power Dissipation Estimate by Switch Level Simulation. IEEE symposium on Circuits and Systems, pp. 881-884, 1989.

[19] F. Vahid, T.D. Givargis. Incorporating Cores into System-Level Specification. International Symposium on System Synthesis, November 1998.

[20] Virtual Socket Interface Association, Architecture Document, http://www.vsi.org, 1997.

[21] G.Y Yacoub, W.H. Ku. An Accurate Simulation Technique for Short-Circuit Power Dissipation Based on Current Component Isolation. IEEE International Symposium on Circuits and Systems, pp. 1157-1161, 1989.