Platune: A Tuning Framework for System-on-a-Chip Platforms

Tony Givargis and Frank Vahid

***Abstract*—System-on-a-chip (SOC) platform manufacturers are increasingly adding configurable features that provide power and performance flexibility in order to increase a platform's applicability. This paper presents a framework, called Platune, for performance and power tuning of one such SOC platform. Platune is used to simulate an embedded application that is mapped onto the SOC platform and output performance and power metrics for any configuration of the SOC platform. Furthermore, Platune is used to automatically explore the large configuration space of such an SOC platform. The versatility, in terms of accuracy and speed of exploration, of Platune is demonstrated experimentally using three large benchmark examples. The power estimation techniques for processors, caches, memories, buses, and peripherals combined with the design space exploration algorithm deployed by Platune form a methodology for design of tuning frameworks for parameterized SOC platforms in general.**

***Index Terms*—Author, please supply your own keywords or send a blank e-mail to keywords@ieee.org to receive a list of suggested keywords.**

I. Introduction

THE GROWING demand for portable embedded computing devices is leading to new system-on-a-chip (SOC) platforms intended for embedded systems. Such SOC platforms must be general enough to be used across several different applications, in order to be economically viable, leading to recent attention to parameterized SOC platforms. Different applications often have very different power and performance requirements. Therefore, these parameterized SOC platforms must be optimally configured to meet varied power and performance requirements of a large class of applications.

A platform is a predesigned computing system, typically consisting of a parameterized microprocessor [3], parameterized memory hierarchy [1], parameterized interconnect buses [14], and parameterized peripherals [9]. An intellectual property (IP) platform comes in the form of a hardware description language. An integrated circuit (IC) platform comes in the form of a chip. An IC platform can be oriented toward prototyping or can be oriented toward implementation in a final product. In recent years, a number of commercial platforms have become available and studied in the literature [12], [16], [18].

Manuscript received September 26, 2001; revised March 26, 2002. This paper was recommended by Associate Editor R. Camposano.

T. Givargis is with Information and Computer Science, Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA (e-mail: givargis@ics.uci.edu).

F. Vahid is with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA. He is also with the Center for Embedded Computer Systems, University of California, Irvine, CA 92697 USA (e-mail: vahid@cs.ucr.edu).

Publisher Item Identifier 10.1109/TCAD.2002.804107.

As a specific example, Motorola has announced a version of an MCORE processor IC with a configurable cache [11]. The MCORE cache is a four-way set-associative unified cache, in which one or more of the ways can be disabled. In the past, when power was not a key issue, there was not a strong reason to disable ways, since this could only hurt performance. However, with power becoming a key issue, shutting down ways can reduce power per cache access by reducing the number of power costly tag comparisons per access and eliminating the power necessary to drive the bit-lines and word-lines. If this reduction is greater than the increase caused by more cache misses and hence power costly accesses to the next level of memory, then overall power is reduced.

Platform developers typically provide extensive software development and debug environments for their platform users but often leave the platform user on his/her own when it comes to tuning. However, as more configurable features get added to platforms, we argue that platform developers should also provide a tuning environment to assist the user in finding the best configuration for his/her application and constraints. Platune is one such tuning environment and the subject matter of this paper.

The remainder of this paper is organized as follows. In Section II, related work is introduced. In Section III, the Platune framework is outlined and the underlying SOC platform is described. In Section IV, the simulation models as well as the power models used in Platune are described. In Section V, the exploration techniques utilized by Platune are described. In Section VI, experimental results are given. In Section VII, concluding remarks are stated.

II. Previous Work

There has been considerable effort in designing tools that enable a designer to measure various performance metrics of instruction-set processors and memory subsystems. We have examined *WARTS*, *SimpleScalar*, *SimICS*, *SimOS*, *WATTCH*, *SimplePower*, *Avalanche*, and an approach based on the *TOSCA* codesign framework. We are unaware of any framework/tools for measuring the performance metrics of a complete parameterized SOC composed of peripherals in addition to the instruction-set processor and memory subsystem.

The Wisconsin Architectural Research Tool Set [7] (WARTS) is a collection of tools for profiling and tracing programs and analyzing program traces, mostly intended for cache and memory hierarchy exploration. The collections of programs included in WARTS are *QPT*, a profiler and tracing system, *CPROF*, a cache performance profiler, and *Tycho/Dinero* cache simulators. Trace-driven simulation has the advantage of being fast and accurate for measuring certain performance metrics, such as cache hit or miss rates, but is too coarse for measuring other types of performance metrics, such as power and timing behavior of processors as well as peripheral devices.

The SimpleScalar [4] toolset is a set of architecture simulators. *SimpleScalar* simulates a MIPS-like (actually, a superset of the MIPS-IV instruction set) architecture at the instruction level. It provides five different simulators that focus on different aspects of the architecture (i.e., high-to-low abstraction level). At the highest abstraction level, *Sim-Fast* is a functional simulator providing quick results without too much statistics and without timing information. At the lower abstraction level, *Sim-Outorder* is a detailed low-level simulator that simulates the microarchitecture cycle by cycle. The SimpleScalar toolset provides the basic simulation infrastructure that is necessary to evaluate modern processor architectures and memory subsystems as well as the effect of particular design implementations (e.g., pipelining, branch prediction, etc.). SimpleScalar does not support power analysis.

SimICS [10] is an instruction level simulator developed at the Swedish Institute of Computer Science. The design objectives of SimICS are to be fast and memory efficient, support complex memory hierarchies, and simulate multiple processors (i.e., simulate interprocessor interrupts, message passing, and external TLB invalidations). The statistics that are gathered by SimICS are memory usage, frequency of important events, and instruction profiling. As with SimpleScalar, while these metrics can be used to explore the memory hierarchy design space, the tool does not provide power consumption estimates.

The SimOS [13] simulator is designed to enable the study of uniprocessor and multiprocessor systems. The SimOS simulator is capable of simulating the computer hardware at the right amount of detail to run an entire operating system. The tool does provide flexibility in the tradeoff between simulation speed and the amount of detail and statistics that is collected. However, power consumption is not directly modeled. Instead, the focus of the simulator design has been to enable the simulation of privileged operating system software, as SimOS is intended to allow a researcher to gain insight in the behavior of a processor system given a realistic application workload.

WATTCH [2] is a system that extends the SimpleScalar simulator for power analysis. Here, power analysis is done at the architectural level and the simulation is built on top of the Sim-Outorder simulator of the SimpleScalar toolset. The WATTCH simulator provides a framework to analyze different configurations, optimizations, and strategies to save power. Since WATTCH is based on SimpleScalar, it focuses on the processor and memory subsystem.

As with WATTCH, SimplePower [17] too is based on the SimpleScalar simulator. SimplePower augments the SimpleScalar simulator with power models in order to estimate the processor core power consumption, memory subsystem power consumption, bus power consumption, and I/O pad power consumption. The SimplePower tool is intended to provide means for optimizing power consumption of the circuits, architecture, and the application software. However, this tool mainly focuses on the processor, memory, and the bus subsystem.

Avalanche is a system-level power estimation framework making use of a trace-based approach, which in turn is done by WARTS, and deploying a mix of analytical models (for instruction cache, data cache, and main memory) and instruction set simulators (ISS) [6]. The Avalanche framework optimizes system parameters in order to minimize energy dissipation of the overall system. Moreover, the tradeoff between system performance and energy dissipation is also explored. This framework focuses on the processor and memory subsystem.

An approach based on the TOSCA codesign framework is proposed by [5]. Their approach performs register–transfer level analysis of power for control-oriented embedded systems, implemented into a single ASIC. The main goal has been to offer a power-oriented codesign methodology, with particular emphasis on power metrics, to compare different design solutions described at high abstraction levels. Unlike the other work presented here, their approach starts with the application specification and seeks to derive a power optimal design by integrating power estimation techniques into a codesign synthesis environment. Platune, on the other hand, assumes a fixed but parameterized SOC platform and explores parameter configurations with respect to a fixed application mapped onto the architecture.

Most of the tools outlined so far are designed for evaluating research ideas pertaining to predominantly performance issues, or tuning of design parameters and subsystems toward optimum performance. Specifically, the focus has been geared toward the memory subsystems. The later efforts have extended the earlier efforts to account for power and system-level design exploration as well and have included the bus power consumption to the other metrics of interest. Platune extends such work further by allowing for power and performance analysis of an entire parameterized SOC platform, including the peripheral components found on the SOC platform. Furthermore, while the earlier work has focused on simulation of a user-selected configuration, Platune allows for automatic search and exploration of Pareto-optimal configurations.

III. Platune Framework

A. Overview

Platune is an environment (i.e., framework or tool) for enabling an embedded system designer (i.e., user of the SOC platform) to select appropriate architectural parameter values, for a given application that is to be mapped on the parameterized SOC platform, in order to meet performance and power goals. This process is also referred to as platform tuning, architecture optimization, parameter selection, and so on. We distinguish between an *embedded system designer* (e.g., builder of a digital camera) and desktop *computer architecture designer*, which is what most previous tools have targeted. Platune is closely tied to a specific architecture, namely the architecture that is to be tuned. This architecture is depicted in Fig. 1. Platune is composed of the following two components.

- Closely integrated simulation models for each of its SOC components (e.g., processors, memories, interconnect buses, and peripherals). These simulation models capture dynamic information essential for computing power and performance metrics.

- Power models for each of its SOC components. Each of these power models must be parameterized according to the parameterization of the respective SOC component.

Platune is capable of performing the following tasks.

- Compilation of a C program and linking of runtime support libraries in order to map an application to the SOC platform prior to simulation or exploration.
- Simulation for the purpose of generating a report on power consumption and execution time given a particular configuration of the SOC platform that is determined by the designer.
- Simulation for the purpose of generating a report on power consumption and execution time given a range (i.e., a subset) of configurations of the SOC platform that is determined by the designer.
- Exploration (automatic) of all possible configurations of the SOC platform for the purpose of generating a report on the possible power and execution time tradeoffs available to a designer.

Platune is designed with the following implementation goals in mind.

- To compute power and execution time metrics that are of high enough accuracy to distinguish inferior configurations from superior ones. The goal is *not* to compute metrics that are accurate in an absolute sense but rather in a configuration-to-configuration *relative* sense.
- To simulate the SOC platform at the highest possible abstraction level for rapid simulation and exploration.
- To explore the configuration space in an efficient manner since the configuration space is exponential in size.
- To avoid simulation whenever possible and use information gathered in previous simulation runs, in order to compute power and execution time for new configurations.
- To provide a single tool that is composed of closely coupled (i.e., integrated) components that interoperate with each other efficiently (i.e., shared memory instead of trace/toggle files, etc.) in order to achieve high-speed simulation and exploration.

In the next section, we will introduce the parameterized SOC platform and explain various tunable parameters.

B. Parameterized SOC Platform

The underlying parameterized SOC platform that Platune is based on is depicted in Fig. 1. The SOC platform works as follows. A MIPS R3000 processor, instruction cache, and data cache communicate over two processor-local buses, namely the CPU-instruction-cache bus and the CPU-data-cache bus. The on-chip memory is connected to the two caches via another bus, namely the cache-memory bus. The universal asynchronous receiver and transmitter (UART) peripheral and the discrete cosine transform (DCT) CODEC peripheral are connected to the peripheral bus. The peripheral bus is in turn bridged over to the CPU-data cache. The various components of this platform are configurable. Likewise, the Platune framework allows for modifying these parameters in its representation of the SOC platform.

The MIPS can be set to run at 32 different voltage levels (1.0 to 4.2 V in increments of 0.1 V) and thus 32 different frequencies. Each of the two caches are five-way set-associative (1, 2, 4, 8, and 16), there are five line-sizes settings (4, 8, 16, 32, and 64 byte), and ten total cache sizes (128 to 64 Kb in power of 2 increments). The four interconnect buses (CPU-instruction-cache, CPU-data-cache, cache-memory, and peripheral bus) are each in turn composed of a data bus and an address bus. Each one of the buses can be set to one of four different widths (4, 8, 16, and 32 wires) and three different encodings (binary, bus-invert, and gray). The UART peripheral's transmitter as well as the receiver buffer sizes can be set to one of four values (2, 4, 8, and 16 bytes). The DCT CODEC peripheral's pixel resolution can be set to one of two widths (16 or 24 bits).

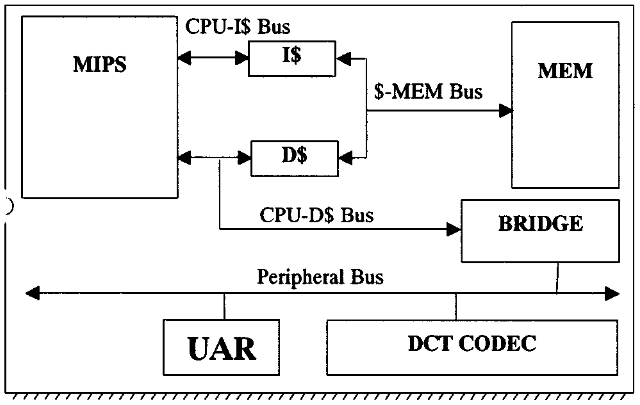


Fig. 1. Parameterized SOC platform.

In summary, there are a total of 26 parameters and a configuration space in excess of 10^{14} configurations. In the following sections, the simulation model and power models of Platune will be described in detail.

IV. Simulation and Power Models

Platune is composed of a collection of closely coupled and integrated simulators, each corresponding to the various component (CPU, cache, interconnect buses, and peripherals) of its SOC platform. In addition to performing a functional simulation, each simulator 4pis designed to gather activity information that, combined with power models, is used to compute the dynamic power consumption of the SOC platform.

The simulator and power models of each of the components will be described in the subsequent sections. We first give the general CMOS power model and technology related parameters that lie beneath all power models described in this paper. Dynamic power for a switching element is given by the following:

$$P = 1/2 \times C \times A \times F \times V^2.$$

In this equation, C is the average capacitance of the switching element. The term A (a number between 0 and 1.0) is a measure of switching activity of the element. The term F is the clock frequency applied to the switching element. The term V is the supply voltage applied to the switching element. The term F, clock frequency, is a (near linear) function of the supply voltage V, as shown in the following [8]:

$$F(V) = ((V - V_{\text{thresh}})^2/V) \times (F_{\max} \times V_{\max}/(V_{\max} - V_{\text{thresh}})^2).$$

TABLE I
TECHNOLOGY PARAMETERS

Parameter	Value
V_{thresh}	0.8 Volt
V_{max}	4.2 Volt
F_{max}	480 MHz
S_{tran}	1e-6 meter
S_{wire}	1e-6 meter
C_{tran}	5e-15 Farad
C_{wire}	1e-13 Farad
B_{len}	1e-2 meter

The term V_{thresh} is the threshold voltage of the underlying CMOS technology. The term $V_{\max}$ is the maximum supply voltage that can be applied to the SOC platform. The term $F_{\max}$ is the maximum allowable clock frequency of the SOC platform at maximum supply voltage, i.e., $V_{\max}$.

Other technology related parameters include S_{tran} (the width/length of a transistor), S_{wire} (the unit length of the shortest interconnect wire), C_{tran} (the average capacitance of a single transistor), C_{wire} (the average capacitance of the shortest interconnect wire, i.e., capacitance per unit length), and B_{len} (the approximate length of the on chip interconnects buses). Typical values for all of the technology parameters are given in the Table I.

In general, each simulator within Platune collects and computes the switching activity, i.e., the A term, during a simulation run. Subsequent to simulation, power consumption of each component (which in turn may be composed of subcomponents) is computed and ultimately summed up to obtain the total power consumption. It must be noted that power consumption and (or) execution time of the SOC platform for pockets of configurations is often obtainable without the need for repeated simulation. As an example, a single simulation can be used to obtain performance metrics for all configurations where the supply voltage is the only varying parameter. This is as a result of our power models analytically linking voltage, power, and clock frequency. Platune is optimized to avoid simulation whenever possible by reusing simulation results of previous runs. In the next sections, we examine each component individually.

A. Processor

The processor simulator maintains detailed statistics on its internal activity, e.g., fetches, stalls, instruction execution frequency, register file access, floating-point activity, etc. Such statistics is used in a post simulation analysis to compute power and performance metrics. The technique used here is an extension to previously given instruction-based approaches [15]. The power consumed by the processor core (excluding caches, interconnect buses and peripherals) can be broken down into the following components

$$P_{CPU} = \left(\sum (E^i_{instruction} \times V^2) + \sum (E^i_{reg\text{-}file} \times V^2)\right) \Big/ T.$$

The summations are over all instructions that where executed by the processor (in the case if $E^i_{instruction}$) and register file access (in the case of $E^i_{reg\text{-}file}$) during the simulation run. The term T is the simulated time (in seconds) of the application running on the SOC platform. The term $E^i_{instruction}$ is the average energy consumption of the ith instruction that is executed during simulation. The $E^i_{reg\text{-}file}$ is the average energy consumption of the ith register file access. The $E^i_{reg\text{-}file}$ is assumed to be constant for any read or write access i and is derived from gate-level simulation and power analysis and normalized for a supply voltage of one Volt. The term $E^i_{instruction}$ is given in the following equation

$$E^i_{instruction} = E(i,k)\colon k \text{ is the previously executed instruction.}$$

The function $E(i,\, k)$ is derived from gate-level simulation and power analysis of the CPU and assumed to be a constant two-dimensional lookup table that is normalized for a supply voltage of one Volt. Both i and k are integers in the range of $[0 \cdots n]$, where n is the number of instructions supported by the processor. Note that it's important to consider inter-instruction dependencies as the currently executing instruction can have an impact on the energy consumption of the next instruction. For example, a multiply successor to a branch instruction may require more energy to compute!

The time complexity of the processor power estimation approach is linear with respect to the length of the application software running on the target processor.

B. Caches

The cache simulators of Platune are fully parameterized modules that operate on a stream of memory references that is generated by the processor simulator during simulation. In addition to the standard cache metrics, such as number of misses used for execution time evaluation, the Platune cache simulators maintain additional activity statistics, e.g., number of tag comparisons, word-line activity, and bit-line activity, that is used to compute the power consumption. The power model for caches is defined as follows

$$P_{cache} = \sum (E_{storage} + E_{word\text{-}line} + E_{bit\text{-}line} + E_{decode})/T.$$

The summation is over all cache accesses performed during a simulation run. The term T is the simulated time (in seconds) of the application running on the SOC platform. The term $E_{storage}$ is the average energy consumption of the storage, tag, and house keeping (valid and dirty) transistors during a single cache access, i.e., a single cache line (block) read/write operation. This value is a function of the cache parameters L (line size), A (associativity), S (total size), and supply voltage V as shown

$$\begin{aligned}
N_{tag\text{-}bit} &= 32 - \log_2(S/(L \times A)) - \log_2(L) \\
N_{data\text{-}bit} &= L \times 8 \\
N_{ctr\text{-}bit} &= 2 \\
N_{total\text{-}bit} &= (N_{tag\text{-}bit} + N_{data\text{-}bit} + N_{ctr\text{-}bit}) \times A \\
N_{\text{tran}} &= N_{total\text{-}bit} \times 2 \\
E_{storage}(L,A,S,V) &= 1/2 \times (C_{\text{tran}} \times N_{ran}) \times 1/4 \times V^2.
\end{aligned}$$

The term N_{tran} is the total number of transistors that are invoked during each cache access. The factor 2 is based on the assumption that a one-bit memory cell in the cache is composed of a pair of transistors. In the energy equation, the 1/2 term and the V^2 term follow from the power equation given earlier. The $(C_{\text{tran}} \times N_{\text{tran}})$ term is a measure of the average switching capacitance. The 1/4 term is a measure of switching activity of the transistor. It is assumed that, on the average, half of the accesses to the cache cause half of the bits in question to switch. This metric can be improved by keeping more detailed simulation data at the expense of slowing down the simulation time. The random data assumption, however, has been shown to work very well [14]. Note that the clock frequency term is left out since we are computing the energy and not the power.

The term $E_{word\text{-}line}$ is the average energy consumption of the word-lines that get activated during a single cache access. This value is a function of the physical width of the cache, which is dependent on the parameters L (line size), A (associativity), and S (total size), as well as the supply voltage V as shown

$$\begin{aligned} W_{word\text{-}line} &= 2 \times N_{total\text{-}bit} \times S_{trans} \\ C_{word\text{-}line} &= C_{\text{wire}} \times (W_{word\text{-}line}/S_{\text{wire}}) \\ E_{word\text{-}line}(L, A, S, V) &= 1/2 \times C_{word\text{-}line} \times V^2. \end{aligned}$$

The term $W_{word\text{-}line}$ is the width of a word-line. Here, we assume that all transistors in a row are laid out side-by-side with the word-lines routed straight through them. The factor 2 is based on the assumption that a one-bit memory cell in the cache is composed of a pair of transistors. This is certainly not an exact assumption, but a reasonable one that allows us to compare two candidate cache architectures in a relative way. The $C_{word\text{-}line}$ term is the average switching capacitance of a word-line. The $E_{word\text{-}line}$ is based on the power equation given earlier. The switching activity parameter is one and hence removed from the equation. The clock frequency term is left out since we are computing the energy and not the power.

The term $E_{bit\text{-}line}$ is the average energy consumption of the bit-lines that get activated during a single cache access. This value is a function of the physical height of the cache, which is in turn dependent on the parameters L (line size), A (associativity), and S (total size), as well as the supply voltage V as shown

$$\begin{aligned} H_{bit\text{-}line} &= S/(L \times A) \\ C_{bit\text{-}line} &= C_{\text{wire}} \times (H_{bit\text{-}line}/S_{\text{wire}}) \\ E_{bit\text{-}line}(L, A, S, V) &= 1/2 \times C_{bit\text{-}line} \times N_{total\text{-}bit} \times 2 \times V^2. \end{aligned}$$

The term $H_{bit\text{-}line}$ is a measure of the height of a bit-line. Here, as with the word-lines, we assume that all transistors in a column are laid out side-by-side with the bit-lines routed straight through them. The $C_{bit\text{-}line}$ term is the average switching capacitance of a bit-line. The $E_{bit\text{-}line}$ is based on the power equation given earlier. The switching activity parameter is $N_{total\text{-}bit}$ since that many bit-lines are energized during each cache access. The factor 2 is based on the assumption that a one-bit memory cell in the cache is composed of a pair of transistors. The clock frequency term is left out since we are computing the energy and not the power.

The term E_{decode} is the average energy consumption of the index decode logic. This value is a function of the index range of the cache, which is dependent on the parameters L (line size), A (associativity), and S (total size), as well as the supply voltage V as shown

$$\begin{aligned} I_{range} &= S/(L \times A) \\ N_{tran\text{-}logic} &= \log_2(2 \times I_{range}) \\ E_{decode}(L, A, S, V) &= 1/2 \times C_{\text{tran}} \times N_{tran\text{-}logic} \times V^2. \end{aligned}$$

The term I_{range} is the index range of the cache. The term $N_{tran\text{-}logic}$ is an estimate of the number of transistors required to implement a decoding unit that can be used to index into the cache. The E_{decode} term is based on the power equation given earlier. The switching activity is assumed to be half of the transistors making up the decode logic switching per access.

It must be noted that the cache structure that is modeled here is very generic and simple in design. The power models given here are intended to for comparing relative quality of caches with different parameters but still identical in structure.

The time complexity of the cache power estimation approach is linear with respect to the length of the application software running on the target processor for the first configuration and constant for subsequent configurations. For the first configuration, the application software is simulated on the target processor to obtain the necessary statistics. For subsequent configurations, the statistics are used along with the above equations and new parameter settings. Certain system configurations may require a re-simulation of the applications software.

C. Memory

The memory simulator of Platune operates on a stream of memory references that are generated by the processor, cache, and bus simulators during simulation. The power models for memory component works as follows

$$P_{memory} = \left(\sum (E_{memory\text{-}first} \times V^2) + \sum (E_{memory\text{-}next} \times V^2)\right) \Big/ T.$$

The summation is over all memory accesses performed during a simulation run. The term T is the simulated time (in seconds) of the application running on the SOC platform.

The on-chip memory power model is simply a per access energy lookup based on weather the access is a first time memory read/write operation (e.g., first read/write from a page in burst mode) or a subsequent read/write operation from a page in burst mode. Note that the access type of a reference (first or next) is accounted for during the simulation run. The term $E_{memory\text{-}frst}$ (the energy of a first read/write operation) is derived from gate-level simulation and power analysis of the memory and assumed to be a constant normalized for a supply voltage of one Volt. Likewise, the term $E_{memory\text{-}next}$ (the energy of a subsequent read/write operation from a page in burst mode) is derived from gate-level simulation and power analysis of the memory and assumed to be a constant normalized for a supply voltage of one Volt.

The time complexity of the memory power estimation approach is linear with respect to the length of the application software running on the target processor for the first configuration and constant, for subsequent configurations. For the first configuration, the application software is simulated on the target processor to obtain the necessary statistics. For subsequent configurations, the statistics are used along with the above equations and new parameter settings. Certain system configurations may require a re-simulation of the applications software.

D. Interconnect Buses

Like the previous simulators, the bus simulators in Platune also operate on a stream of data and memory references that are generated by the processor, cache, and memory modules and accumulates bus wire bit toggle statistics that are used for subsequent power consumption computation. Without lack of generality, and for the following discussion, we assume that a bus is either an address bus or data bus of one of the four buses with the term $S(W, C)$, the bit switching activity for the parameter W (width) and C (encoding), computed during simulation. The following will compute the power consumption of the bus

$$P_{\text{bus}}(W, C, V) = (1/2 \times C_{\text{bus}} \times S(W, C) \times V^2)/T.$$

The term T is the simulated time (in seconds) of the application running on the SOC platform. The P_{bus} value is based on the power equation given earlier. To compute C_{bus}, we consider the relative spacing of the bus, i.e., the fewer the wires, the larger the spacing between adjacent bus wires. In our assumption, the routing area devoted to interconnect buses is constant

$$C_{\text{bus}}(W) = B_{len} \times C_{\text{wire}} \times (1 + (W/32)^2).$$

Here, the constant 1 factor accounts for the wire capacitance resulting from coupling to the ground and substrate planes. The $(W/32)^2$ factor accounts for the coupling capacitance to the neighboring wire. Note that the bus width is normalized such that at max width (32 wires) the capacitance ratio due to coupling to ground/substrate equals that due to coupling to neighboring wires.

The time complexity of the bus power estimation approach is linear with respect to the length of the application software running on the target processor for the first configuration and constant, for subsequent configurations. For the first configuration, the software is simulated on the target processor to obtain the necessary statistics. For subsequent configurations, the statistics are used along with the above equations and new parameter settings. Certain system configurations may require a re-simulation of the applications software.

E. Peripherals

Much work has concentrated on system-level modeling of microprocessors, caches, memories, and buses. But for general-purpose peripherals, such as the UART and DCT CODEC, there has been little previous work established. In Platune, peripherals are viewed as executing a sequence of *instructions*. Classically, an instruction represents an atomic action available to a microprocessor programmer. We use the term *instruction* more generally as an action that, collectively with other actions, describes the range of possible behaviors of a peripheral core. We have extended the instruction-level power modeling approach that was previously used for microprocessor cores, for use with peripheral cores. In building the power models for our peripherals we have performed a number of steps that are described in what follows.

The first step is selecting instructions for our peripherals. For the UART and DCT CODEC, we have first broken the functionality into a set of instructions. These instructions have the property that they collectively cover the entire functionality of the particular core. As with the instructions of an instruction-set processor, each instruction operates on some input data and produces some output data. For example, for the UART, one might select the following instructions: *Reset*, *Enable_tx*, *Enable_rx*, *Send*, and *Receive*. *Send* and *Receive* may operate on bytes of data.

The second step is instruction data-dependency modeling of our peripheral cores. For each instruction, we have determined how dependent that instruction's power consumption is on that instruction's input data. Let us define an instruction's *power-dependency characteristic* as one of: *dependent* directly on its input data, dependent on a *statistical* characterization of its input data (e.g., the density of 1s in a vector of bits), or *independent* of its input data. For example, for the UART core, we ran experiments that provided different data to each instruction. Then, we determined that the power-dependency characteristic for all instructions was "independent." In other words, the *Send* instruction consumed approximately a constant amount of power regardless of the data being sent; likewise for the *Receive* and other instructions.

The next step is core power-mode modeling. Very unlike microprocessors, certain instructions executed on a peripheral core can drastically change the power consumption of succeeding instructions. In particular, certain instructions change the *mode* of the peripheral core. This concept of mode is very different from that of measuring interinstruction power dependencies (e.g., a load following a store may consume more power than a load following an add). To account for this, we determined the set of modes of the two peripheral cores in question that caused the cores to consume significantly more or less power per each execution of their instructions. With our UART core, we found four power modes: *Idle*, *Tx_enabled*, *Rx_enabled*, and *Tx_rx_enabled*.

The final step is gate-level power evaluation. We use gate-level simulation to obtain per-instruction power data and capture this information in lookup tables. This procedure involves a methodical way of creating a set of testbench models that, when simulated at gate-level, capture the power consumption of each instruction, in each particular mode, with each particular parameter setting. As an example, Table II gives the lookup table for the UART peripheral. The rows correspond to instructions while columns correspond to the UART's buffer size parameter values. The entries are repeated for each one of the four modes.

The time complexity of our peripheral power estimation approach is linear with respect to the length of the application software running on the target processor for the first configuration and constant, for subsequent configurations. More specifically, for the first configuration examined, the time complexity of the estimation approach is a function of how frequently a particular peripheral device is accessed by the application software.

TABLE II
UART'S POWER LOOKUP TABLE

Buffer size (byte) →	2	4	8	16	2	4	8	16
	Mode 1: Idle				Mode 2: Tx_enabled			
Reset	11	13	14	14	1	13	14	14
Enable_tx	27	32	31	31	23	23	22	24
Enable_rx	17	18	19	18	19	20	19	19
Send	17	19	19	20	133	135	157	209
Receive	14	15	17	18	14	16	18	18
	Mode 3: Rx_enabled				Mode 4:Tx_rx_enabled			
Reset	13	14	15	15	13	13	14	14
Enable_tx	22	22	21	21	21	22	21	21
Enable_rx	18	19	18	18	19	19	19	19
Send	19	19	21	23	133	135	157	209
Receive	73	83	93	111	74	81	93	107

V. EXPLORATION

So far, we have described the simulation model and power analysis techniques used in Platune. In this section, we define the exploration problem and outline the techniques used for performing it automatically. Our SOC platform is composed of numerous parameters. We enumerate each of these parameters as $P_1, P_2, P_3 \ldots P_N$. Each of these parameters can be assigned a value from a finite set of values, namely its domain. A complete assignment of values to all the parameters is a *configuration*. The problem is to efficiently, compute the *Pareto-optimal* configurations, with respect to power and performance, for a fixed application executing on the SOC platform. In our problem, a configuration C_i is Pareto-optimal if no other configuration C_j has better power as well as performance than $C_i (i \neq j)$.

A. An Exhaustive Exploration Algorithm

We start by outlining an exhaustive algorithm to solve the exploration problem. In this exhaustive algorithm, first, power and performance are evaluated for all configurations. Then, configurations are sorted by nonincreasing execution time (i.e., higher performance). Last, in the sorted order, a walk through the space is performed while all configurations that result in power consumption above the minimum seen thus far are eliminated. The remaining configurations are Pareto-optimal. The algorithm is given below.

```
Algorithm 1:
list compute_Pareto_configurations(space
  $s$) {
  list all, Pareto;
  float min_power = 1e100; /* infinity */
  for each configuration $c$ in space $s$ {
    simulate_SOC($c$);
    all.push($c$);
  }
  all.sort(/* key is execution time */);
  while(!all.empty()) {
    $c$ = all.pop();
    if($c$.power < min_power) {
      min_power = $c$.power;
        Pareto.push($c$);
    }
  }
  return Pareto;
}
```

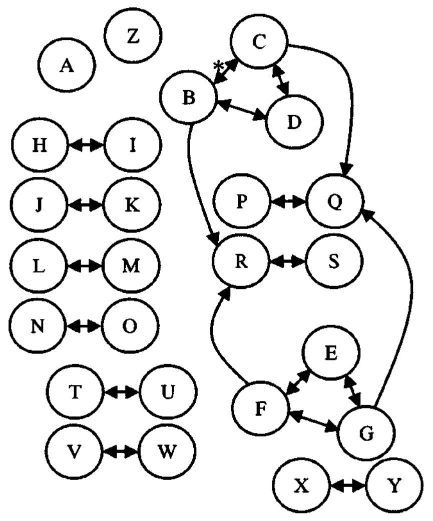
The problem with this approach is that the configuration space is likely to be very large, making the approach impractical in many cases. The exhaustive approach is practical when applied to a small subset of the solution space consisting of one or two varying parameters while all others held constant. We have found that many parameters in an SOC platform have little interdependency among each other. Two parameters are *interdependent* if changing the value of one of them impacts the optimal parameter value of the other. For example, it may be that the set-associativity and line size parameters of the instruction cache are interdependent. However, the set-associativity parameter of the instruction cache and the line size parameter of the data cache are not interdependent. An efficient algorithm can take advantage of such interdependencies of parameters (or lack of it) to prune the configuration space.

B. Parameter Interdependency Model

We have used a graph model to capture the parameter interdependencies. Such a graph is constructed with its nodes representing parameters and edges representing interdependencies between parameters. Generally, a path from a node A to a node B indicates that the Pareto-optimal configurations of B should be calculated once the Pareto-optimal configurations of all the nodes from A to B, residing on the path, are calculated, in that order. During that calculation all other parameters not on the path can be temporally fixed to some arbitrary value. A path from a node A to a node B and back to A, which forms a cycle, indicates that the Pareto-optimal configurations of all the parameters on the cycle need to be calculated simultaneously. During that calculation all other parameters not on the path can be temporally fixed to some arbitrary value. The Pareto-optimal configurations of an isolated node can be computed by temporally setting all other parameters to some arbitrary value. The dependency graph used by Platune is depicted in Fig. 2. We assume that the designer of the SOC platform determines the interdependencies among the parameters. Often these interdependencies follow from the structure of the SOC platform. For example, given an optimal configuration of the instruction cache, one can tune the data cache parameters without affecting the optimality of the instruction cache, since an optimally performing instruction cache will maximize instruction cache hit rate and no data cache configuration can have an effect on the instruction cache. In our graph, there are no edges going from B, C, or D to E, F, or G, stating that the instruction cache and data cache are not interdependent. If the designer cannot establish the interdependency of two or more parameters, than he or she should conservatively assume that they are interdependent. In future work, we plan to automate this dependency determination.

C. An Efficient Exploration Algorithm

Given an interdependency graph, our algorithm works as follows.



Node	Core	Parameter		Node	Core	Parameter
A	MIPS	Voltage scale		L	CPU-D$-bus	Data bus width
B	I$	Total size		M		Data bus code
C		Line size		N		Addr bus width
D		Associativity		O		Addr bus code
E	D$	Total size		P	I/D$-Mem-bus	Data bus width
F		Line size		Q		Data bus code
G		Associativity		R		Addr bus width
H	CPU-I$-bus	Data bus width		S		Addr bus code
I		Data bus code		T	Peripheral-bus	Data bus width
J		Addr bus width		U		Data bus code
K		Addr bus code		V		Addr bus width
X	UART	Tx buffer size		W		Addr bus code
Y		Rx buffer size		Z	DCT CODEC	Pixel resolution

* Note that we have used double-arrowed lines in place of two single arrowed-lines for clarity.

Fig. 2. Target SOC interdependency graph.

Algorithm 2:
```
list compute_Pareto_configurations_2
  (graph $g$) {
  list sub_graphs, Pareto;
   sub_graphs $=$ strongly_connected_components($g$);
  // part 1
   for each subgraph $g$ in sub_graphs {
     Pareto $=$ compute_Pareto_configurations($g$.space);
    eliminate configurations. in $g$.space not in Pareto;
  }
  // part 2
  while(!sub_graphs.size()$!=$ 1) {
    $g1 =$ sub_graphs.pop_front();
    $g2 =$ sub_graphs.pop_front();
     $g = g1$ union $g2$;
    sub_graphs.push_back($g$);
    Pareto $=$ compute_Pareto_configurations($g$.space);
    eliminate configurations. in $g$.space not in Pareto;
  }
  return Pareto;
}
```

The algorithm can be broken down into two phases. The first phase performs a local search for Pareto-optimal configurations. The second phase iteratively expands the local search to discover global Pareto-optimal configurations.

The first phase of our algorithm performs clustering of interdependent nodes in the graph. This is the same problem as finding strongly connected components of a graph (e.g., a depth first search can be used to accomplish this). In addition, if two clusters are connected (but not strongly), then they are topologically ordered. Here, each cluster represents a disjoint subspace of the overall configuration space. We use our exhaustive algorithm for calculating Pareto-optimal configurations for each of the clusters. Then, we restrict possible configurations of that cluster to the Pareto-optimal configurations only. This pruning is justified since if a configuration is not Pareto-optimal within a cluster, it cannot be part of a Pareto-optimal configuration for the entire configuration space. Conversely, if a configuration is Pareto-optimal within a cluster, it may or may not be Pareto-optimal given the entire configuration space, and thus must remain. Our exhaustive approach applied to clusters is usually feasible since these clusters represent only a small subspace of the total configuration space. Nevertheless, heuristics such as probabilistic exploration techniques can be used to search within a cluster when the exhaustive method is too time consuming.

The second phase of our algorithm combines pairs of clusters into a single cluster and computes Pareto-optimal configurations within it. Then, it limits the space of this new cluster to the Pareto-optimal configurations only. This procedure is repeated until all the clusters have been merged and a single cluster remains. The Pareto-optimal configurations within this last cluster represent Pareto-optimal configurations of the entire configuration space.

The worse case time complexity of the algorithm is bounded by $O(K \times M^{N/K})$, where K denotes the number of initial strongly connected components (i.e., clusters) computed in part 1, N denotes the number of parameters, and M denotes the upper bound on the number of values each parameter can receive. Note that for our target architecture, depicted in Fig. 1, $K = 13$, $M = 32$, and $N = 26$. Here, the $M^{N/K}$ factor bounds the running time of the exhaustive computations of the Pareto-optimal points, while K bounds the number of times the first/second part of the algorithm iterate. In the worst case,[1]

TABLE III
EXPERIMENTAL RESULTS: ESTIMATION ACCURACY AND SIMULATION SPEED

		Average Power Consumption (Watt)						Execution Time (sec)
		CPU	Caches	Buses	Memory	Peripherals	Total	
Small Caches, 24-bit CODEC	Platune	0.031	0.0047	0.0017	0.00065	0.021	0.059	0.138
	Gate-level	0.029	0.0049	0.0017	0.00070	0.020	0.057	
Large Caches, 12-bit CODEC	Platune	0.033	0.0098	0.0012	0.00043	0.011	0.055	0.128
	Gate-level	0.029	0.0107	0.0012	0.00046	0.011	0.052	
Average Accuracy Error ⇒		10%	7%	3%	8%	5%	8%	

when $K = 1$ (all parameters are interdependent) the running time is exponential, namely M^N. In the best case, when $K = N$ (all parameters are independent) the running time is linear, namely N. For most practical cases the running time will be closer to the best case since the factor $M^{N/K}$ will decrease very rapidly as K increases.

VI. EXPERIMENTS

In this section, we intend to experimentally demonstrate three qualities of Platune, namely, accuracy of power and performance estimates, speed of simulation, and speed of exploration. In all our experiments, Platune is used to explore the configuration space of some application software executing on the hardware architecture depicted in Fig. 1. The architecture and its parameters are described in Section III.

To demonstrate the accuracy of power estimates generated by Platune, we compared the power breakdown of the various SOC components generated by Platune (e.g., processor, memory, buses, and peripherals) to gate-level power estimations. Since Platune is a cycle accurate simulator, the performance (i.e., execution time) statistics reported are exact in terms of the number of cycles. For this experiment, we used the *PowerStone* [11] *jpeg* benchmark, an implementation of the JPEG image decompression standard that is roughly 620 lines of C code. The inverse DCT (IDCT) function of this algorithm was mapped to the DCT CODEC peripheral while Huffman decoding and dequantization was mapped to the MIPS processor of our parameterized SOC platform. The results are given in Table III. Here, only two configurations of the SOC platform are shown. However, in the table, the average accuracy error is computed by looking at 48 different configurations of the SOC platform. In the first configuration, the instruction and data caches are set to 1024 byte, direct mapped, and 4 byte/line, and the CODEC is configured to perform 24-bit IDCT computations. In the second configuration, the instruction and data caches are set to 4096 bytes, two-way set-associative, and 4 byte/line, and the CODEC is configured to perform 12-bit IDCT computations. In all, we examined 48 random configurations and computed the average power estimation error. The average error for the total SOC power consumption was 8%.

To demonstrate the simulation speed, we compared the time taken to simulate 48 configurations of the *jpeg* example running on Platune with the time taken performing gate-level simulation. The results are shown in Fig. 3. On the average, Platune required 23 s, while gate-level simulation required 47 490 s (over 13 h) per configuration. Platune is over 2000 times faster than gate-level simulation. The experiments ran on a machine with dual 500-MHz Sun Ultra Spark II processors.

To demonstrate the exploration speed of Platune, in addition to *jpeg* we used two other PowerStone benchmark applications, namely *g3fax*, a group three fax decoder (single level image decompression) at roughly 652 lines of C code, and *v42*, a modem encoding/decoding algorithm at roughly 743 lines of C code. The results are given in Table IV. For the *jpeg* example, a total of 18 611 configurations where examined, which took about 3 days. Of those, 157 were Pareto-optimal with respect to power and performance. For the g3fax example, a total of 14 350 configurations where examined, which took about 1.6 days. Of those, 134 were Pareto-optimal with respect to power and performance. For the *v42* example, a total of 15 731 configurations where examined, which took about 1.7 days. Of those, 133 were Pareto-optimal with respect to power and performance. The power and performance tradeoff are shown in Fig. 4. We repeated our experiments for the remaining PowerStone benchmarks with similar conclusion. We discovered that the discontinuities in the plots occurred when the size, line or set-associativity of the caches crossed the working set boundary of the particular application that was being executed. In general, cache parameters unlike others (e.g., processor voltage setting) do not affect power and performance metrics monotonically. We note that the average of 1 to 3 days taken to perform the exploration is reasonable when considering that tens of thousands of design

[1]Our analysis only holds for the worse case because on the average, clusters will not always receive an equal number of parameters, thus the algorithm will be more accurately bounded by $O(K \times M^{(N-K)})$.

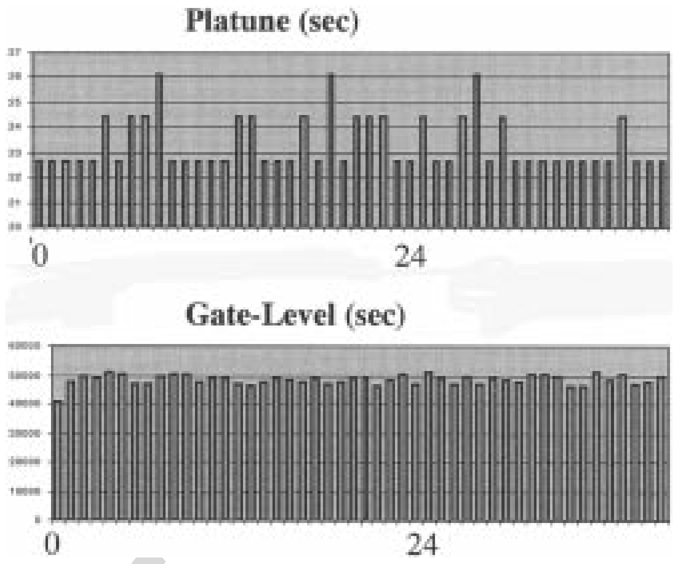
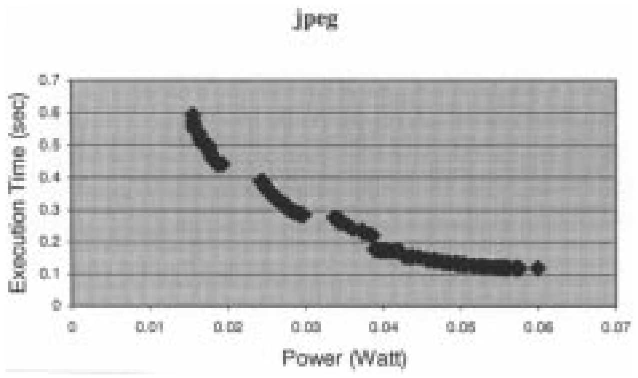


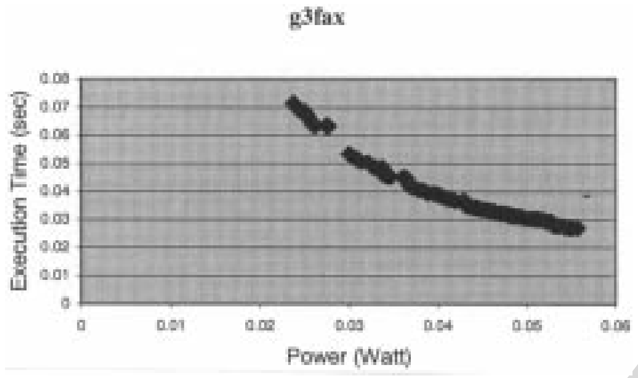
Fig. 3. Simulation speed of Platune and gate-level simulation.

TABLE IV
EXPERIMENTAL RESULTS: DESIGN SPACE EXPLORATION

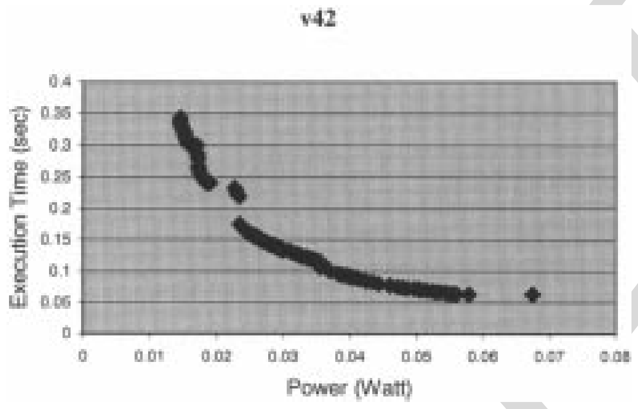
Application	Exploration Time	Configurations Visited	Pareto Configurations	Power Tradeoff	Execution Time Tradeoff
ipeg	2.9 days	18611	157	3.9 x	5.0 x
g3fax	1.6 days	14350	134	2.6 x	2.3 x
V42	1.7 days	15731	133	4.7 x	5.4 x



(a)



(b)



(c)

Fig. 4. Pareto-optimal set with respect to power and performance.

alternatives are examined and the optimal choices are discovered. Currently, we are investigating heuristics to speedup such exploration even further.

We have demonstrated that Platune can achieve overall accuracy of 8% when compared to gate-level power estimates. We have experimentally shown that Platune can compute power and performance metrics over 2000 times faster than gate-level simulation. For three large examples, the exploration techniques used by Platune discovered the Pareto-optimal set by extensively pruning the design space of 10^{14} configurations.

VII. Conclusion

We have given an overview of the Platune parameterized SOC simulation and exploration framework. Platune is a tool to aid the system designer in selecting appropriate architectural parameter values, for a given application that is to be mapped on the parameterized SOC platform, in order to meet performance and power goals. We have shown that Platune is accurate in estimating power and performance metrics. Moreover, the exploration techniques used by Platune, based on parameter interdependency models, can discover the Pareto-optimal configurations (with respect to power and performance) by extensively pruning the design space of 10^{14} configurations. The search process for three large applications took of the order of 1–3 days. Our future work focuses on reducing this time down to hours using a combination of heuristics and parameter interdependency models. The techniques and algorithms used by Platune can be applied to tuning platforms targeted toward any parameterized SOC platform.

References

[1] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *Proc. Annu. IEEE/ACM Int. Symp. Microarchitecture*, Dec. 2000.

[2] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. Annu. Int. Symp. Computer Architecture*, June 2000.

[3] T. D. Burd, T. A. Pering, A. J. Stratakos, and R. W. Brodersen, "A dynamic voltage scaled microprocessor system," in *IEEE Int. Solid-State Circuits Conf.*, Nov. 2000.

[4] D. Burger and T. M. Austin, "The SimpleScalar tool set, Version 2.0," Univ. Wisconsin-Madison, Computer Sciences Dept., Tech. Rep. 1342, June 1997.

[5] W. Fornaciari, P. Gubian, D. Sciuto, and C. Silvano, "Power estimation of embedded systems: A hardware/software codesign approach," *IEEE Trans. VLSI Syst.*, vol. 6, June 1998.

[6] Y. Li and J. Henkel, "A framework for estimating and minimizing energy dissipation of embedded HW/SW systems," in *Proc. Design Automation Conf.*, June 1998.

[7] M. D. Hill, J. R. Larus, A. R. Lebeck, M. Talluri, and D. A. Wood, "Wisconsin architectural research tool set," *Computer Architecture News*, vol. 21, no. 4, Sept. 1993.

[8] I. Hong, M. Potkonjak, and M. B. Srivastava, "On-line scheduling of hard real-time tasks on variable voltage processor," in *Proc. Int. Conf. Computer-Aided Design*, Nov. 1998.

[9] J. Lahti, "A parameterized Reed–Solomon CODEC for ASIC implementation of forward error correction functions," in *Proc. NORCHIP Conf.*, Nov. 1999.

[10] P. Magnusson and B. Werner, "Efficient memory simulation in SimICS," in *Proc. Simulation Symp.*, Apr. 1995.

[11] A. Malik, B. Moyer, and D. Cermak, "A lower power unified cache architecture providing power and performance flexibility," in *Proc. Int. Symp. Low Power Electronics and Design*, June 2000.

[12] S. Ortiz, "New chips move networking onto silicon," *IEEE Comput.*, Feb. 1999.

[13] M. Rosenblum, S. A. Herrod, E. Witchel, and A. Gupta, "Complete computer system simulation: The SimOS approach," *IEEE Parallel Distributed Technol.: Systems Applicat.*, vol. 3, Winter 1995.

[14] M. R. Stan and W. P. Burleson, "Bus-invert coding for low power I/O," *IEEE Trans. VLSI Syst.*, Mar. 1995.

[15] V. Tiwari, S. Malik, and A. Wolfe, "Power analysis of embedded software: A first step toward software power minimization," *IEEE Trans. VLSI Syst.*, vol. 2, pp. 437–445, Dec. 1994.

[16] J. van Meerbergen, A. Timmer, F. Leijten, and M. S. Harmsze, "Experiences with system-level design for consumer IC's.," *IEEE Trans. VLSI Syst.*, Feb. 1998.

[17] N. Vijaykrishnan, M. Kandemir, M. J. Irwin, H. S. Kim, and W. Ye, "Energy-driven integrated hardware–software optimizations using SimplePower," in *Proc. Int. Symp. Computer Architecture*, June 2000.

[18] "Velocity product information," VLSI Technology, Inc.

Tony Givargis received the B.S. and Ph.D. degrees in computer science from the University of California, Riverside, in 1987 and in 2001, respectively.

He is currently an Assistance Professor in the Department of Information and Computer Science at the University of California, Irvine. His is also a member of the Center for Embedded Computer Systems at UC Irvine. He is a coauthor of the textbook *Embedded System Design* (New York: Wiley, 2002). His research interests include platform-based system design, real time resource management of embedded computing systems, and design space exploration.

Dr. Givargis was a GAANN (Graduate Assistance in the Area of National Need) Fellow in 2001 and received the department's best Thesis Award.

Frank Vahid received the B.S. degree in computer engineering from the University of Illinois, in 1988, and the M.S. and Ph.D. degrees from the University of California (UC), Irvine, in 1990 and 1994, respectively,

He is currently an Associate Professor in the Department of Computer Science and Engineering at the University of California, Riverside. He is also a faculty member at the Center for Embedded Computer Systems at UC Irvine. He is coauthor of the textbooks *Embedded System Design* (New York: Wiley, 2002) and *Specification and Design of Embedded Systems* (Englewood Cliffs, NJ: Prentice Hall 1994). His current research interests include architectures and design methods for low-power embedded systems, with an emphasis on tuning system-on-a-chip platforms to their executing programs.

Dr. Vahid was an SRC Fellow, in 1994, and he received the Outstanding Teacher of the College of Engineering award, in 1997. He was program and general chair for the IEEE/ACM International Symposium on System Synthesis in 1996 and 1997, respectively, and for the IEEE/ACM International Workshop on Hardware/Software Codesign in 1999 and 2000. He received the best paper award from IEEE TRANSACTIONS ON VLSI SYSTEMS in 2000.