Hindawi Publishing Corporation
VLSI Design
Volume 2011, Article ID 726014, 17 pages
doi:10.1155/2011/726014

Research Article

A High-Throughput, High-Accuracy System-Level Simulation Framework for System on Chips

Guanyi Sun,[1] Shengnan Xu,[1,2] Xu Wang,[1] Dawei Wang,[1] Eugene Tang,[1] Yangdong Deng,[1,2] and Sun Chan[1]

[1] *Tsinghua-Intel Center of Advanced Mobile Computing, Tsinghua University, Beijing 100084, China*
[2] *Institute of Microelectronics, Tsinghua University, Beijing 100084, China*

Correspondence should be addressed to Guanyi Sun, sun.guanyi@gmail.com

Received 21 November 2010; Revised 23 February 2011; Accepted 20 May 2011

Academic Editor: Matteo Reorda

Today's System-on-Chips (SoCs) design is extremely challenging because it involves complicated design tradeoffs and heterogeneous design expertise. To explore the large solution space, system architects have to rely on system-level simulators to identify an optimized SoC architecture. In this paper, we propose a system-level simulation framework, System Performance Simulation Implementation Mechanism, or SPSIM. Based on SystemC TLM2.0, the framework consists of an executable SoC model, a simulation tool chain, and a modeling methodology. Compared with the large body of existing research in this area, this work is aimed at delivering a high simulation throughput and, at the same time, guaranteeing a high accuracy on real industrial applications. Integrating the leading TLM techniques, our simulator can attain a simulation speed that is not slower than that of the hardware execution by a factor of 35 on a set of real-world applications. SPSIM incorporates effective timing models, which can achieve a high accuracy after hardware-based calibration. Experimental results on a set of mobile applications proved that the difference between the simulated and measured results of timing performance is within 10%, which in the past can only be attained by cycle-accurate models.

1. Introduction

As the backbone for embedded systems, System on Chips (SoCs) play an increasingly important role in the modern society. Today a typical SoC consists of one or more processors, a large capacity of memory, an interconnection fabric, as well as a set of accelerators (e.g., [1]). Besides the complex hardware, generally a complicated software system is composed by a real-time operating system, a given protocol stack, and various applications that will be running on the SoC hardware. To develop an efficient and effective SoC solution, designers have to explore a large solution space by comparing a huge number of different design options, so as to identify an optimized system configuration. Since modern SoCs are usually built with a set of IP blocks, an optimized system interconnection is essential for the success of SoC design. Such an interconnection fabric can be as simple as a shared bus or as complex as a networks on chip [2]. Once again, a good communication architecture can only be derived through detailed performance profiling and analysis. Currently, the explorations of the communication and computation architectures can only be performed with a system-level simulator [1]. In fact, running applications on an executable SoC model can identify arbitrary internal details of a design and thus provide key insights for SoC designers.

As part of a joint project between Tsinghua University and Intel Corporation to develop the next-generation mobile computing infrastructure, we are developing an SoC as the vehicle to deliver future wireless computing applications. By taking a software-as-a-service (SAAS) approach [3], the SoC design is now under a new set of constraints. Especially, many applications can leverage the computing power of servers within such a framework, and thus the SoC design will involve a large number of tradeoffs. It is thus essential to construct an efficient system-level simulation framework for SoC architects to rapidly explore the solution space.

Specifically, the simulation framework has to meet the following requirements.

(i) Accuracy. Traditionally cycle-accurate simulation is the dominant way to derive performance evaluations for decision making at the SoC architecture design stage. Transaction Level Modeling (TLM) technique now gradually gains popularity due to its higher simulation throughput [4]. However, a careful validation of the correlation between TLM models and real hardware, especially under an industry setup, is still largely missing in the current literature. Based on an extensive survey on industry experts, we believe a simulation accuracy of 15% will be a threshold to justify a TLM level simulator. In other words, the performance derived from the simulation must be within 15% of the measured results of real hardware.

(ii) Throughput. The target SoC of this project will serve as a reference design for a new generation of mobile computing technology. A large number of applications have to be simulated to evaluate the performance of a specific SoC architecture. The simulation throughput, therefore, has to be sufficiently high (e.g., cannot be slower than the simulated system by two orders of magnitude).

(iii) Heterogeneity. An SoC would install one or more CPUs, various accelerators or offloading engines, a large number of peripherals, and a potentially complex communication fabric. These components follow different computation and communication models and have to be modeled and simulated with varying methodologies.

(iv) Extendibility. A large number of computational IPs and communication protocols need to be evaluated to choose an optimal SoC microarchitecture. In addition, current SoC design typically will develop into a family of products with varying costs and/or performance levels. So the simulator must allow easy plug-in of different computation and communication IPs.

(v) Comprehensiveness. A simulation framework should cover all major design metrics including application execution latency, processing throughput, and power/energy consumption. In addition to overall performance measurements, SoC designers must use the simulator to expose key details such as bus occupation ratio and workload distribution among IPs.

There is already a large body of research dedicated to modeling of CPUs (e.g., [5–8]). These works cover the whole spectrum of CPU designs from functional level to cycle-accurate micro-architecture level. In the past few years, system-level modeling of SoCs attracts considerable research efforts (e.g., [9–12]). These works provided important insights into various aspects of SoC designs and paved the road for a systematic modeling methodology. Moreover, the release of SystemC Transactional Level Modeling (TLM) 2.0 [4] provides a unified foundation to build high-throughput SoC models. The TLM approach attains a higher simulation speed by skipping many timing details, and thus its timing accuracy still can be a concern to guide micro-architectural level design decisions.

On the basis of previous research on system-level modeling, we developed a comprehensive SoC modeling framework to pursue the above objectives. Especially, this work is aimed to achieve a high simulation throughput on real industry applications, while at the same time attaining a good correlation between simulated results and real hardware execution. Our simulation framework, System Peripheral Simulation Implementation Mechanism, or SPSIM, consists of 3 major components, a simulation tool chain, an executable SoC model, and an overall methodology. The target SoC is modeled at the transaction level to expedite the simulation process. The major contributions of this work are as follows.

(i) Integrating state-of-the-art SoC modeling techniques, the SPSIM framework offers comprehensive SoC evaluation and exploration capabilities. It has been used in real-world, industry-strength SoC designs.

(ii) Adopting an executable-driven and trace-driven hybrid simulation technique, our simulator provides superior simulation efficiency by safely skipping instructions that have little impact on system level timing. According to our experimental results on real-world embedded applications, the simulation time is within 35x of the native execution time.

(iii) This work emphasizes a good correlation between simulated and hardware-executed results. Leveraging our cutting-edge hardware development infrastructure, we performed extensive experiments to validate the accuracy of the simulator. Experimental results proved that the performance difference between the simulated results and the real hardware execution is within 10%. Our work proves that a transaction level model can achieve both high simulation throughput and good timing fidelity on real-world applications.

(iv) Our framework incorporates a series of effective techniques to ease the SoC design effort. Such techniques include generic template modules that can easily be configured and extended for real-world SoC modules. In addition, a power modeling methodology is constructed to allow early exploration on energy efficiency. The effectiveness of these techniques has been proved by industry practices.

SPSIM proved to be valuable for the design of SoCs. Its typical use cases include both system architecture exploration and early software development. It offers a highly efficient and sufficiently accurate modeling framework for SoC architects to quickly evaluate complex tradeoffs in hardware and/or software architecture development. Meanwhile, due to its high simulation throughput, SPSIM can also serve as a virtual platform on which software programmers can construct system and application software before real hardware is available. Currently, SPSIM has been deployed in a few leading edge SoC design projects. The whole

modeling framework will be made publicly available as an open-source package. In the released version, we will also include templates for master and slave devices as well as communication fabrics.

The rest of this paper is organized as follows. Section 2 briefly introduces the preliminaries of SystemC and TLM2.0. We present SPSIM's 3 major components, the simulation tool chain, the modeling methodology, and the target SoC model, in Sections 3, 4, and 5, respectively. In Section 6, we report experimental results to demonstrate the efficiency and accuracy of SPSIM. Finally, Section 7 concludes the talk and outlines future research directions.

2. Preliminaries

SPSIM is developed on top of the SystemC language and the TLM2.0 package. In this section, we briefly review the related background information.

SystemC is an extension to the standard C++ language by providing a series of new classes and application programming interfaces (APIs). As a result, SystemC has the capability to model both hardware and software. Hardware can be modeled at various abstraction levels from untimed algorithmic level down to cycle-accurate register transfer level, while software can be represented as pure functional specification or real application code. With specially tailored data structures and encapsulation mechanisms, SystemC is designed to decouple computation and communication [4]. In other words, the communication modules do not need to be changed when switching to a different communication fabric and vice versa. To derive a performance evaluation, a SystemC-based hardware/software description can be simulated via its discrete event simulation kernel (released as a linked library).

Transaction Level Modeling 2.0 is a class library layered on top of the SystemC basic classes. The key idea is to ignore pin-accurate and clock-accurate information, but model data transfer among system modules as transactions, which are represented as function calls. The timing is only accurate at the boundary of transactions. By smoothing out the internal details of transactions, the simulation speed can be significantly improved. TLM2.0 provides complete language structures for modeling module interfaces, sockets, generic payload, and basic communication protocols.

In our simulation framework, we use QEMU [13] and Skyeye [14] for functional simulation of both the X86 and ARM CPU architectures, respectively. Released as an open source package, QEMU is a processor emulator that relies on dynamic binary translation to achieve a relatively high emulation throughput and to minimize the effort of porting to new host CPU architectures. QEMU supports the emulation of a wide range of architectures, including IA-32 (X86) PCs, X86–64 PCs, MIPS R4000, Sun SPARC, and PowerPC. In this work, we only use it to interpret X86 instructions.

Originated from GDB/ARMulator [15], Skyeye is also an open source software package. The goal of Skyeye is to provide a unified CPU simulation environment on both Linux and Windows machines. Skyeye environment simulates typical embedded computer systems such as ARM and MIPS. This simulator can boot embedded operation systems like Embedded Linux and uClinux [16]. It provides fast flexible and convenient mechanisms for full-system emulation as well as debugging.

3. Design and Implementation of SPSIM Simulation Tool Chain

An SoC simulator has to capture the complicated interaction between a CPU and the remaining components. The CPU will boot a software stack, which typically consists of operating system, middleware, and applications. The remaining components of SoC will be driven by the system calls initialized by the CPU. During program execution, data will be transported among CPU, memory blocks, and other functional blocks through a communication fabric.

3.1. Fundamental Considerations of SPSIM Design. Executable driven and trace driven are two major approaches to build a system-level simulator platform. With the executable driven method like QEMU [13], the simulator directly handles an executable binary and can easily capture dynamic run-time information. However, simulators constructed in this way tend to be more complicated due to overhead of dynamically processing the input binary. With the trace driven method, a simulator is driven by events stored in a trace file containing all necessary information for performance simulation. Such an approach generally leads to simpler and faster simulators. An extra advantage is that system architects can reproduce the interested phenomena with little effort. The negative side is that trace driven simulators only manipulate static information of a program, and thus certain dynamic information such as the interaction among different threads cannot be simulated.

Another basic consideration is to perform functional or performance simulation. For software design and functional validation, functional simulation is sufficient. On the other hand, performance simulation is necessary for tradeoff analysis in SoC architecture design and software-hardware codesign.

The next question is at which abstraction level the simulator should be constructed. Cycle accuracy level simulator like ASIM [8] offers good accuracy at the cost of long simulation time. This work adopts the recent popular transaction level modeling methodology. Compared with the large body of existing work on transaction level modeling of SoCs, our focus is on simulation accuracy.

As discussed in Section 1, our simulator needs to estimate the performance of the whole system, capture the dynamic interaction among different threads and different peripherals, and precisely reproduce key operation scenarios. Accordingly, we take a novel hybrid simulation methodology that decomposes the simulation process into executable and trace-driven stages, while the trace-driven stage practices performance simulation at the transaction level.

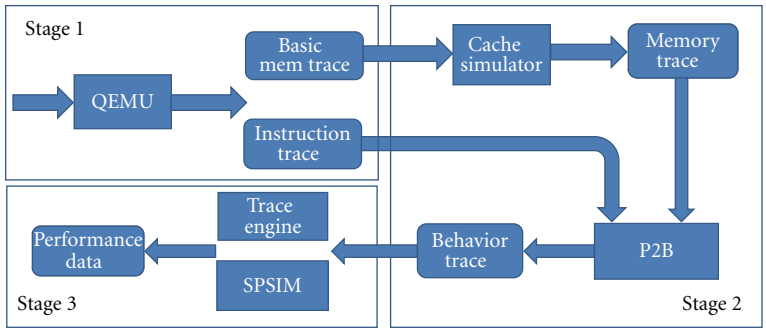

Figure 1: Overview of whole simulation flow.

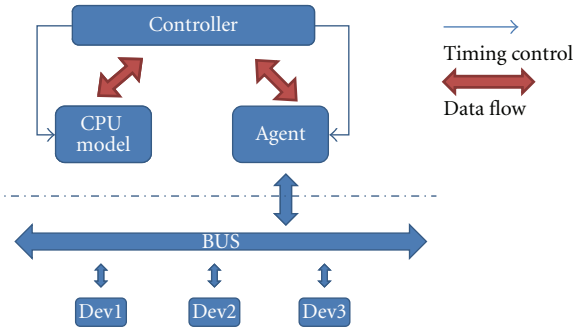

Figure 2: Fundamental simulation architecture.

Therefore, SPSIM utilizes a hybrid modeling methodology by integrating executable and trace-driven simulation techniques. A complete simulation cycle is illustrated in Figure 1. At the beginning, a target application is directly executed on a CPU executable simulator to extract instruction and memory operations, which potentially would trigger peripheral operations. Such operations are recorded into instruction and memory traces, respectively. After proper filtering and preprocessing, the above two traces are compiled into a special trace, or so-called behavior trace, which will be fed into the event-driven SystemC simulation engine to derive accurate performance statistics. For each application, the executable simulation only needs to be run once to create the trace. The trace-driven performance simulation process, however, can be executed many times with different system configurations. This very methodology has been used in fast micro-architecture simulation (e.g., [17, 18]). To the best of the authors' knowledge, this work is the first to adopt such a methodology under the context of SoC design. The remaining of this section presents the details of our simulation tool chain.

3.2. CPU Modeling. CPU plays a central role in an SoC. Therefore, it is critical to capture the behavior in an SoC model. On the other hand, manually creating a CPU model can be a labor-intensive and error-prone process. It should also be noted that we usually care more about the CPU output, which will trigger all SoC peripherals/accelerators. In other words, SoC architects usually do not need the internal details of a CPU. In addition, it is a common practice for SoC designers to try different processors at the prototyping stage.

As a result, SPSIM is designed for easy plug and play of different executable CPU models/simulators. In this work, we port QEMU [13] to capture the functional behavior of an X86 CPU core as well as its cache memory. The original QEMU simulator is extended so as to interpret the instructions of an Intel Atom processor [19]. For ARM binaries, we use Skyeye [14] as the simulator. We also implemented a recording mechanism to keep track of all instruction and memory access of a given application.

Figure 2 illustrates the basic thinking for plugging an arbitrary CPU model into SPSIM. While the CPU model takes care of the internal behavior, the interaction between CPU and the other parts of a SoC is through an agent. Both parts are coordinated by a controller.

The first task of the controller is to synchronize the CPU model and the remaining components modeled in SystemC. Here the major challenge is that the CPU model and the SystemC model generally follow different clocks. In this work, we adopt a synchronization protocol as follows.

(1) The controller activates the CPU model to run one basic instruction block, that is, a number of instructions with a single exit point.

(2) After the basic block completes its execution, the controller calculates how many cycles have passed by taking into account the number of simulated instructions and a premeasured cycles-per-instruction (CPI) value. Then the SystemC models will also need to be simulated for this number of cycles.

The second task of the controller is to build and maintain the memory channel. The original memory accesses, especially those fell into I/O space, will be bypassed in CPU functional simulation. Meanwhile, the agent module provides a series of memory access APIs. Upon receiving a bypassed memory request, the controller calls a proper API function through the agent. The API function will then activate corresponding SystemC module(s) to serve the original memory request. The agent guaranteesthat

each request can be encapsulated as one transaction packet and proper responses are created after a transaction completes.

3.3. Simulation Flow. In this subsection, we present the details of our system-level simulation flow and the corresponding tool chain. As illustrated in Figure 1, the overall simulation process is divided into three consecutive stages: (1) performing executable simulation to collect instruction and memory traces, (2) processing the traces in stage 1 to generate behavior trace, and (3) obtaining performance data via behavior trace-driven simulation. The essence of this 3-steps simulation flow is to decompose functional and timing simulation processes. The first step tries to simulate the target SoC behavior with the host CPU and find the correct results of a target application. Then the second step creates a trace by skipping instruction execution details that have less significant impact on final time information. Finally the 3rd step finds the exact time performance by going through the detailed behavior of SoC components.

In the first stage of the workflow, a ported CPU simulator such as QEMU will load application program(s) and interpret instructions. Note that our simulation platform is able to boot an unmodified target OS like Linux or Android [20]. A complete run of this step creates two basic traces, an instruction trace and a memory trace. The instruction trace records the whole instruction sequence of the input application including program counter (PC) values, instruction words, and op-codes. The memory trace contains such information as effective address, operation value, and operation code for every memory access.

The second stage can be further decomposed into steps. In the first step, the basic memory trace will be filtered by an in-house cache simulator. In fact, our focus is system-level activities happening on the system level interconnects, accelerators and peripherals. Most memory accesses will be serviced by cache and will not actually go through the system level interconnects. Accordingly, the cache simulator will extract L2 cache misses and uncacheable memory accesses and store them into a reduced memory trace. The memory operations in this trace have a real impact on the system interconnect and peripherals. The advantage of this filtering step is twofold. First, the size of memory trace can be reduced significantly, usually by an order of magnitude. Second, removing the pure CPU and cache interactions also simplifies the operations of tools in the succeeding flow.

Today a typical workload on SoC involves multiple threads. The dynamic behaviors of multiple threads are much more complex, especially when two or more threads are competing for the same executable unit. For instance, if two threads want to access the same memory with single port, one thread has to stall until the other threads are serviced. Therefore, following the spirit of simultaneous multithreading [21], we skip the cache filtering stage for multithreaded applications to keep more detail and maintain the simulation accuracy.

In the second step of this stage, we implement a tool to synthesize the instruction trace and the reduced memory trace into a so-called behavior trace. The tool is designated as P2B, or program-to-behavior converter. P2B analyzes the two traces, with special emphasis on memory operations and branch instructions, since these two types of instructions impact the system performance from both timing and power perspectives. To be specific, an access to memory or I/O space will trigger a transaction and its timing is decided by current system conditions including current occupation of the system bus throughput of the target peripheral. In addition, the power consumption of a memory access varies when it is a cache hit or miss. For branch instructions, the instruction latency and power consumption depend on the actual execution path of the program as well as the results of branch prediction.

The processing flow of P2B is shown in Figure 3. P2B reads records in both traces in a sequential manner. When handling nonmemory access instructions, P2B directly dumps them to the resultant behavioral trace. For memory access instructions, P2B combines the memory operation details with the corresponding information of basic instruction block and then dumps a new record into the resultant behavioral trace.

In the original trace, actually many instructions such as those for arithmetic operations only incur a fixed delay averagely and do not affect system-level performance. These instructions are designated as regular instructions by us. Therefore, a postoptimization procedure in P2B is performed to organize such regular instructions into one single record so as to improve the simulation throughput. In the multithreaded mode, we do not use this optimization due to the fact that adjacent instructions might belong to different threads.

In the last stage of the simulation flow, the behavior trace generated in the previous stage is consumed by a trace engine, which is implemented as a special SystemC module. The whole SPSIM model, including system interconnect, accelerators, and peripherals, is driven by events generated by the trace engine according to content in behavior trace.

Figure 4 shows the basic workflow of the trace engine. The whole behavior trace contains three types of instructions: blocks of regular instructions, blocking memory access instructions, and nonblocking memory access instructions. For a block of regular instructions, the trace engine first identifies the total number of internal instructions from the trace record. With precomputed tables of cycle per instruction (CPI) and working frequency collected from real hardware, the time spent on a processor to execute this block can be derived. The handling of the remaining two types of instructions is more complex. Every memory access instruction in the behavior trace initializes a system-level transaction. The mechanism to handle blocking and non-blocking memory access instructions differs in how to trigger the next instruction or instruction block. After producing a transaction payload, the non-blocking access can trigger next instruction block immediately. However, a blocking memory instruction will suspend the succeeding processing until the necessary feedback is received. Based on the operation principle of trace engine, (1) can be

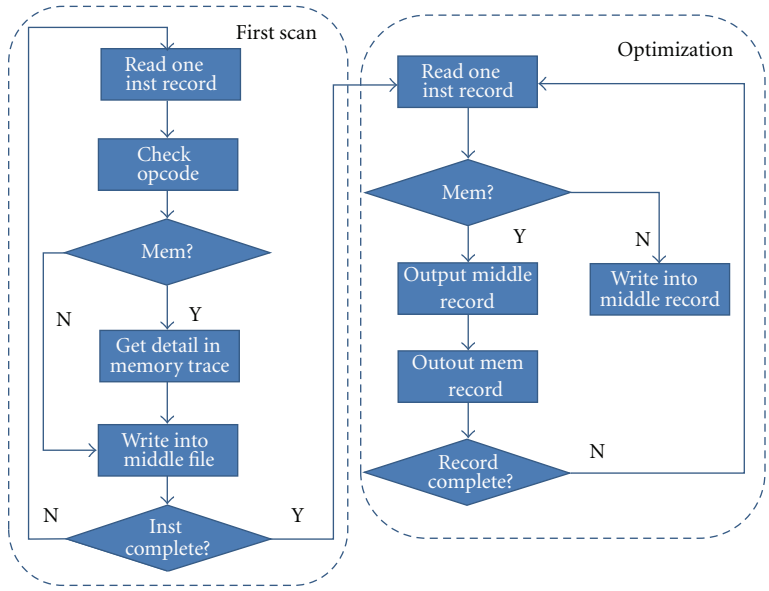

Figure 3: Analysis flow of behavior trace.

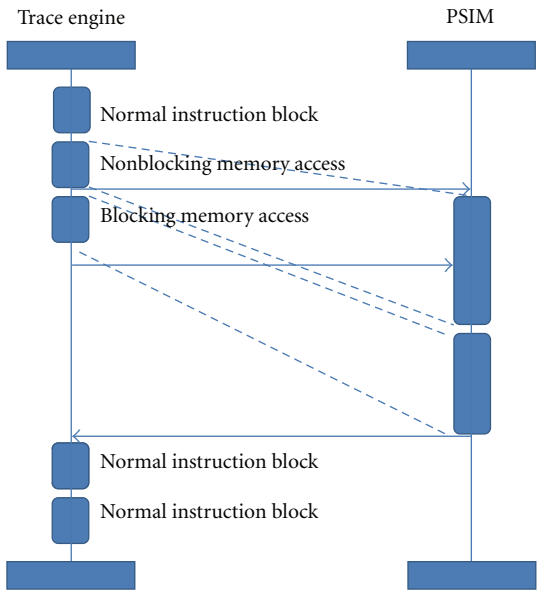

Figure 4: Processing flow of trace engine for 3 types of instruction.

used to calculate the total time spent on execute the input application:

$$T_{\text{total}} = \sum \left(I_{\text{num}} * \frac{\text{CPI}}{\text{freq}_{\text{cpu}}} \right) + \sum T_{\text{blocking}}. \quad (1)$$

Finally we summarize the operations of each stage as follows.

Stage 1. Derive raw instruction and memory traces of an application program on a given CPU.

Stage 2. Compile the original trace files into an integrated behavior trace with a unified format. Typical operations at this step include adding cache hierarchy and TLB information into final trace if necessary, compressing regular instructions to one block to enhance final simulation speed, and maintaining the correctness of memory access sequence.

Stage 3. Launch a trace engine to drive the whole SoC model according to the behavior trace. The trace engine performs the following operations: (1) simulate regular instruction blocks and find the corresponding execution latencies by considering both CPI and CPU clock frequency; (2) organizes I/O device accesses into transactions according to a preestablished memory map; (3) controls the memory (or I/O space) access mode to distinguish blocking and non-blocking requests; (4) injects dynamic actions such as the data movement from outside memory to internal memory upon a TLB miss; (5) coordinates the behavior of multithreaded applications by scheduling multiple threads and resolving conflicts on internal resources usage.

4. Modeling Technology

Future SoCs have to deliver increasing functionality under strict power constricts. One essential goal of this project is to equip SoC architects with the proper set of modeling tools. In this section, we introduce two enabling technologies, generic module template and power estimation, for the above objectives.

4.1. Generic Module Template. In this work, we have already developed an SoC model targeting a future wireless computing infrastructure. However, we certainly cannot cover all possible SoC designs. As a result, a generic modeling methodology is critical, and the remaining of this subsection explains our generic module, which can serve as a foundation for arbitrarily complex modules.

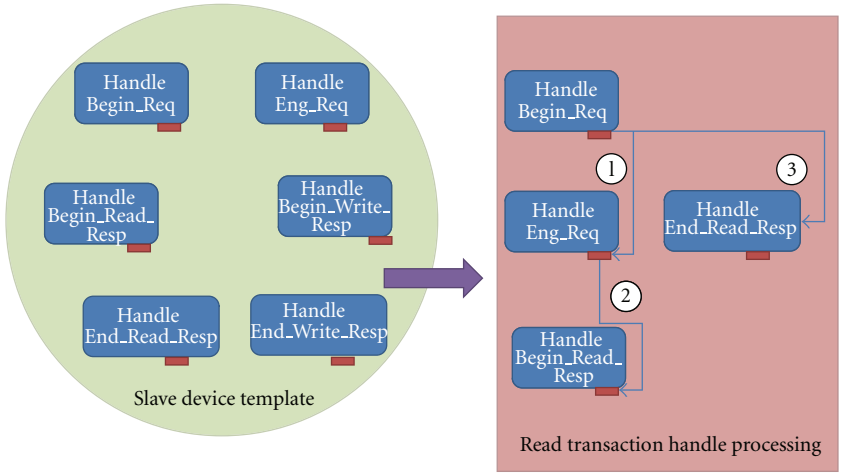

Figure 5: Template for a generic target.

Accordingly, we develop generic target and initiator models by extracting typical communication and computing patterns of a large number of peripherals. Supporting a basic communication flow, the generic module provides a common socket interface as well as bus interfaces such as the receiving and transmission FIFOs. While in the generic module, a set of registers are deployed to control the throughput behavior such as size of I/O buffer and processing capability of one data unit in a given target device. An arbitrary peripheral or accelerator can be modeled by inheriting this basic class and properly configured by setting up the registers. Using template can bring two major advantages: one is that the developer can enhance working efficiency via only focus on modeling behavior of given peripheral module. The other is that peripheral module can be modeled quickly via configuring throughput characteristic control registers. Plus using queuing theory, some necessary performance data can be derived and with enough accuracy.

Next we use the generic class for a given target device as an example to show key points in implementation. The organization and data flow of this template are illustrated in Figure 5. This generic slave class uses six basic functions (left of Figure 5) for operations at various stages of communication protocol. On the right side of Figure 5 is the example of a read transaction flow, which can be fulfilled by calling corresponding functions from the template class. As illustrated in Figure 6, a virtual function is injected at the end of each stage function implementation. By overriding these virtual functions, the whole working processing can be built rather easily.

Meanwhile, using above binding mechanism, the operation flow based on the generic module can be straightforwardly customized for a new one. It only needs to simply change key events notified in a related virtual functions. The process is exemplified in Figure 7, where a new stage is inserted in the operation flow.

The other essential functionality of the template class is to maintain the basic communication protocols. In order to simulate hardware in a more realistic manner, we modified the standard TLM2.0 communication protocol by reversing the direction of BEGIN_RESP and END_RESP when handling write issue in the approximate time mode. Figure 8 shows this difference. The BEGIN_RESP action delivers written data from initiator to target, while the END_RESP action propagates the completion message from target to initiator. To also be able to communicate with IPs with standard TLM2.0 compatible protocols, we adopt a parameter in the constructor to indicate which protocol should be used.

4.2. Power Estimation. In this section, we introduce the details of another important aspect of system-level modeling, power estimation. SPSIM incorporates an effective power modeling methodology to provide energy estimation for SoCs.

4.2.1. Introduction. Today's SoCs, especially those for mobile applications, are under strict constraints for energy consumption due to limited capacity of battery. It is thus essential for SoC architects to accurately predict the energy consumption of a given SoC architecture. Both IP blocks and interconnect fabric in an SoC will contribute to the energy consumption. For an IP block, the IP vendor usually provides a spread-sheet-based power model. When a SoC simulator extracts transactions happened to the IP, a simple table lookup can readily give power estimation. On the other hand, the interconnect fabric, no matter it is proprietary or standardized, has to be customized to connect the specific set of IP blocks on a SoC. Meanwhile, system interconnect is consuming an increasingly percentage of power when the SoC complexity is rising. Our simulation framework, therefore, has to provide the capability of modeling the power consumption of a communication fabric with arbitrary topology.

In this work, we propose a generic power modeling methodology that can be applied to any communication

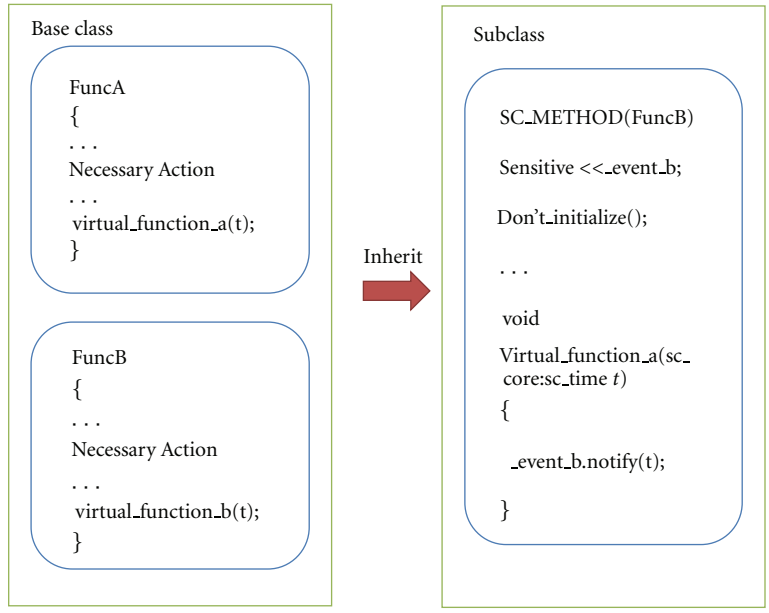

Figure 6: Hooking different functions via virtual interface.

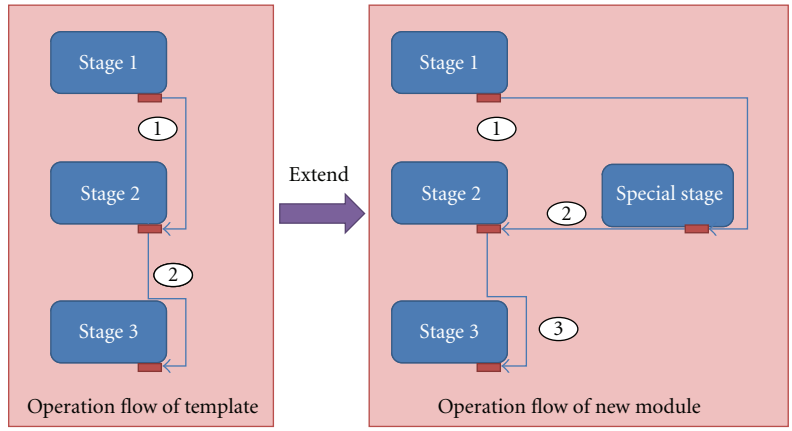

Figure 7: Extending operation flow in template class.

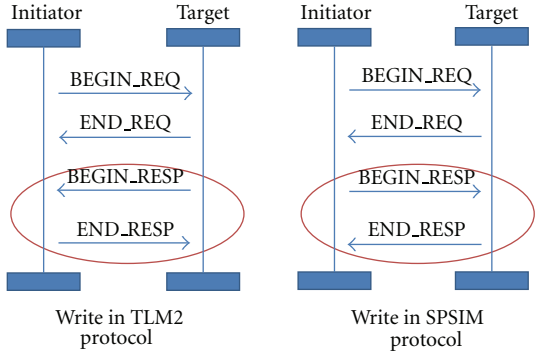

Figure 8: Basic write protocols supported by SPSIM.

fabrics modeled at the transaction level. Currently, SPSIM integrates this power model for AHB bus system [22]. In the remainder of this sub-section, we use the AHB bus to illustrate our generic methodology.

As illustrated in Figure 15, an AHB system will allocate a shared bus to every slave devices. Multiple masters can connect to a shared bus through a multiplexer (MUX). Generally a SoC has multiple slave devices, and thus an AHB system is often designated as a bus matrix. In Figure 9, the modules named as DW_ahb_icm_i ($i = 1, 2, \ldots, n$) is Synopsys' configurable Multilayer Interconnection Matrix (ICM) IP [23]. It is designed to facilitate multilayers access to one and only one specific slave. Each master needs one decoder to decide which ICM should be visited as well as one multiplexer to get the desired data back. In an AHB system, both the logic and interconnect (i.e., wires) need to consume power. These two sources of power have to be modeled in different manners.

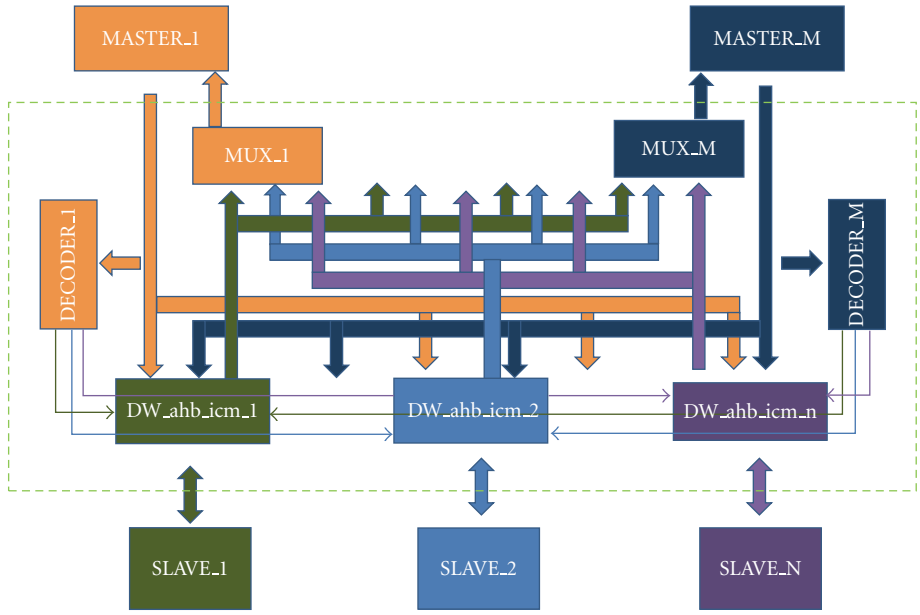

Figure 9: AHB bus matrix architecture.

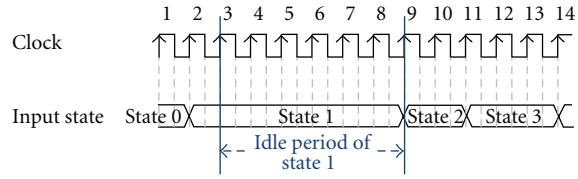

Figure 10: Decomposition of a multicycle state.

There is already a large body of research proposing various methodologies and algorithms for system-level power modeling (e.g., [24, 25]). Again in this work we focus on improving the correlation accuracy of a TLM model with real hardware.

The dominant sources of power consumption include both logic switching and leakage. However, the actual power consumed for even a standard bus significantly varies in different designs. Consequently, in this work we propose a power estimation methodology by fitting empirical data collected from RTL level power analysis. Our power model only depends on parameters reflecting characteristics of the process, design, and communication traffic.

4.2.2. Power Modeling for the Communication Fabrics. In this sub-section, we detail power modeling techniques for an AHB system interconnects illustrated in Figure 9.

An AHB system bus needs to consume energy in both logic devices and interconnect wires. Note that AHB is a synchronous system. At the positive edge of each clock cycle, if any of the input ports switch, the system interconnect will enter a new state. If the state lasts for more than one cycle, say n cycles, the total energy consumption by the logic devices of system interconnect can be decomposed into two parts, $W_{\text{state-transition}}$, the energy consumed by the state transition in the 1st cycle, and $W_{\text{idle-state}}$, the idle energy within *the remaining* $n - 1$ cycles. Here T_{cycle} is the cycle period. Figure 10 is an illustration of the system states. For the interconnect wires, we need to further distinguish input wires (from master to bus logic) and output wires (from bus logic to slave). Accordingly, we can compute the complete energy as:

$$W_{\text{application}} = W_{\text{state-transition}} + W_{\text{idle-state}} + W_{\text{input-wires}} + W_{\text{output-wires}}. \quad (2)$$

$W_{\text{state-transition}}$ is the total energy cost due to the change of state and can be calculated with (3), while $W_{\text{idle-state}}$ is the energy consumed in the idle cycles:

$$W_{\text{state-changing}} = \sum_{i=1}^{\#\text{state-transitions}} W_{s_{i-1}-s_i}, \quad (3)$$

$$W_{\text{idle-state}} = \sum_{i-1}^{\#\text{state-transitions}} P_{s_{i-1}} \times (n_{s_{i-1}} - 1) \times T_{\text{cycle}}. \quad (4)$$

$W_{S(i-1)-Sn}$ is the single energy cost from state $i - 1$ to state i, and $P_{S(i-1)}$ is the power consumption during state $i - 1$. If we directly use (3) and (4), the $W_{S(i-1)-S(i)}$ *and* $P_{S(i-1)}$ values have to be stored in a pre-computed table by characterizing the underlying circuit at either hardware level or at a lowering

modeling level (e.g., register-transfer level or gate level). However, if the module has x input ports, there will be as many as $2^{2x} W_{S(i-1)-S(i)} s$ and $2^x P_{S(i-1)} s$, which imply a huge memory space to store them.

We performed extensive power analysis on an FPGA-proofed register-transfer level (RTL) implementation of an AHB bus system (i.e., Figure 9) with PrimeTime [26]. Our experiments make us reach the following observations.

(i) Multiplexers and decoders cost little power (less than 3%) and can be ignored for power analysis. Therefore, the ICMs (i.e., DW_ahb_icm modules) can be studied separately.

(ii) The idle power consumption in one ICM is proportional to the number of masters.

(iii) When multiple masters request for visiting the same slave (i.e., a conflict), it costs the same energy as the case when these masters access the slave one by one.

(iv) The energy consumed by the switching of address ports is a linear function of the number of switched bits, which can be described like this: $W_0 + H * W_{\text{ebit}}$, where W_0 is determined by the specific type of a transaction as well as the master and number of layers, H is the total number of switching bits, and W_{bit} stands for the energy consumption caused by every switching bit.

(v) The switching of data input ports consumes much less energy than the switching of the address ports does. As a result, the data ports switching can be ignored.

(vi) Since W_0 is determined by transaction type and its value is a linear function of the length of data in the present transaction, we can disassemble W_0 into $W_{00} + S_{\text{slope}} * B_{\text{beats}}$, so that W_{00} and S_{slope} only differ with master ID and number of masters.

(vii) Given a specific bus architecture, $P_{S(i)}$ is almost a constant at different stages static input ports. The reason is because there are a relatively large number of registers installed inside the ICM.

The above observations can be used to significantly reduce the storage requirements for $W_{S(i-1)-S(i)}$ and $P_{S(i-1)}$. As a matter of fact, a state of the AHB bus is now equivalent to a transaction. $W_{S(i-1)-S(i)}$ can be derived as

$$W_{s_{i-1}-s_i} = W_{00}(m, M_{\text{ID}}) + S_{\text{slope}}(m, M_{\text{ID}}) \times B_{i-1,\text{beats}} + \left(\sum_{q=1}^{B_{\text{beats}}} H_{q,\text{switch}} \right) \times W_{\text{bit}}. \tag{5}$$

Meanwhile, the idle energy is derived as

$$W_{\text{idle-state}} = P_{\text{idle}} \times T_{\text{cycle}} \times \left[\sum_{i-1}^{\text{final-state-number}} \left(n_{s_{i-1}} - B_{i,\text{beats}} \right) \right]. \tag{6}$$

The parameters of (5) and (6) are listed in Table 1.

P_{idle} can be written as

$$P_{\text{idle}} = m \times P_{\text{idle0}}, \tag{7}$$

where P_{idle0} is a real constant.

Besides the logic devices, the wires of a system interconnect also consume power. The wiring power can be calculated as follows:

$$W_{\text{input-wires}} = \frac{1}{2} \left(\sum_{i=1}^{\text{final-state-number}} C_{i,\text{total-wire-input}} \right) \times U^2,$$
$$W_{\text{output-wires}} = \frac{1}{2} \left(\sum_{i=1}^{\text{final-state-number}} C_{i,\text{total-wire-output}} \right) \times U^2. \tag{8}$$

Then we can deal with $C_{i,\text{total-wire}}$, which is the effective wire capacitance of both input and output wires. Nowadays, an increasing number of IPs are deployed in SoC designs. The area information of these IPs will be available from IP vendors. Hence, we can try to estimate the length of global wires by creating a "virtual floorplan" of the target SoC design. We use an open-source floorplanning tool [27] to derive the SoC floorplan and an open-source Steiner tree package [28] to estimate the wire length. Our model can also take into account the buffers to drive long wires. The wiring capacitance can be calculated as follows:

$$C_{i,\text{total-wire-input}} = \left[\sum_{q=1}^{B_{\text{beats}}} \left(\sum_{\text{switched-bits}} L_{\text{wire-length,input}} \right) \right] \times C_0,$$
$$C_{i,\text{total-wire-output}} = \left[\sum_{q=1}^{B_{\text{beats}}} \left(\sum_{\text{switched-bits}} L_{\text{wire-length,output}} \right) \right] \times C_0. \tag{9}$$

C_0 is the unit length capacitance determined by the target process. In fact, for a large VLSI system, the above capacitance values should be multiplied by a correction factor that is determined by the design goals (i.e., timing driven or power driven) [29].

5. SoC Model

We implemented a model for a template SoC configuration as shown in Figure 15.

The infrastructure configuration consists of a target processor (X86 or ARM), a multilayer bus matrix, a DMA controller, a DDR2 memory, a LCD module, and an NE2000 Ethernet interface that the trace engine is modeled as a special master in the trace-driven stage of system simulation. We also integrate a number of accelerators and peripherals into the SoC.

5.1. Trace Engine. Trace Engine plays an essential role in the SPSIM simulation platform. First, it interprets the behavior trace, creates transactions if necessary, and then delivers the transactions to drive the whole system. Second, it maintains

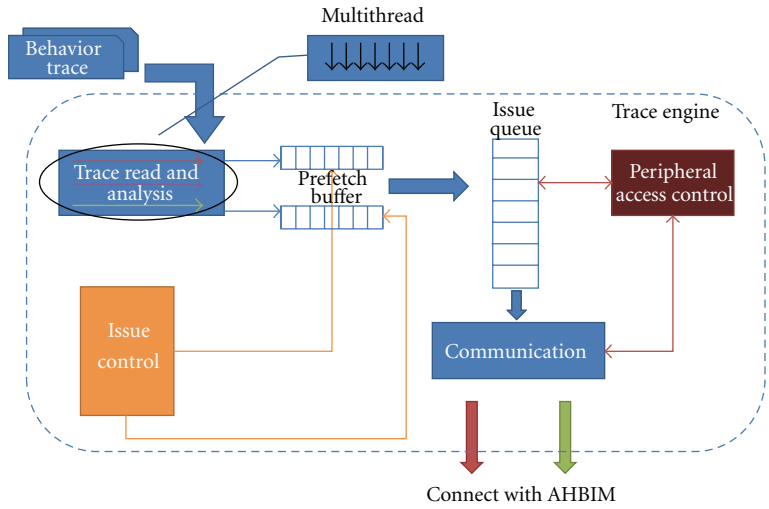

Figure 11: Handling multithreaded workload in SPSIM.

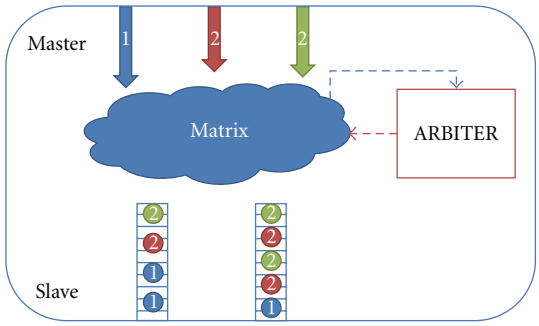

Figure 12: Diagram of an AHB bus scheduler.

Table 1: Definition of parameters.

	Parameter definitions
m	Total number of masters
M_{ID}	Master ID
$W_{00(m,M_{ID})}$	A constant component for master M_{ID} when there are m masters in the system
$S_{slope}(m, M_{ID})$	A constant slope data for master M_{ID} when there are m masters in the system
$H_{q,switch}$	Total switching bits from the beat q to beat $(q+1)$ within a certain transaction. Here one beat stands for one complete process of transmitting a single data word. The transmission includes both address/control phase and data phase, which are pipelined in AHB protocol and can take multiple cycles.
W_{bit}	The energy consumption by the logic switching (i.e., excluding wire energy) of a bit
P_{idle}	Average value of $P_{S(i)}$
T_{cycle}	Time spent in one clock cycle
n_{Si}	Total number of cycles in transaction i
$B_{i,beats}$	Data length of transition i

necessary timing information for the performance statistics. As illustrated in Section 3, we use the precharacterized CPI values under typical workloads and clock frequency to estimate the latency of a basic instruction block.

As mentioned previously, a key task of the trace engine is to support multithreaded workload. The basic mechanism for handling multiple threads is shown in Figure 11. In order to analyze multiple threads in parallel, we record the trace of every thread in an individual file. After being analyzed, the contents of trace files are fetched and stored in separate per-fetch buffers. Then, according to policies of instruction issuing and inner resource usage in the target processor, the issue control module moves qualified prefetch instructions to an issue queue, where the instructions will finally be translated into transactions and sent to the memories, peripherals, or communication fabric of a SoC. The action of the issue unit is also impacted by the feedback from some special peripheral module, such as the DMA controller.

5.2. Interconnect Matrix Bus. The multi-layer bus matrix follows the AHB protocol [22]. In the current configuration, every slave device is associated to a single virtual channel with multiple master devices connected. An arbitrator module is responsible for scheduling transactions and resolving conflicts according to a priority assignment. For different priority devices, the lower priority value component has higher priority. For the peers with same priority value, we use round-robin algorithm for scheduler. If a request from a device can be serviced without confliction, it will be pushed to a sending queue and then sent out after a certain delay. Otherwise it will be suspended until the conflicting condition disappears.

In Figure 12, there are three masters and two slaves connected with an AHB bus. The leftmost master has a higher priority, while the other two have an identical lower

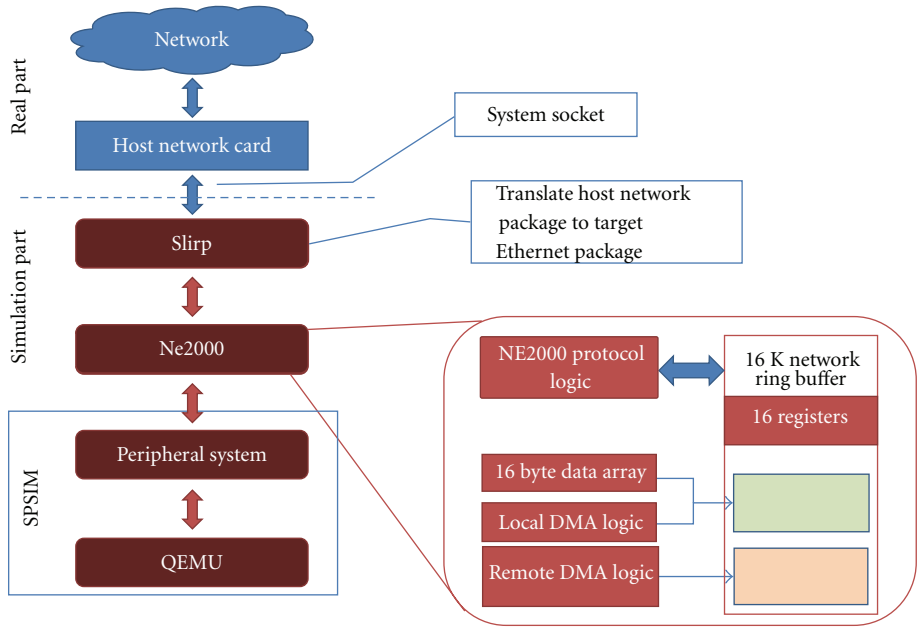

Figure 13: Diagram of the NE2000 network interface.

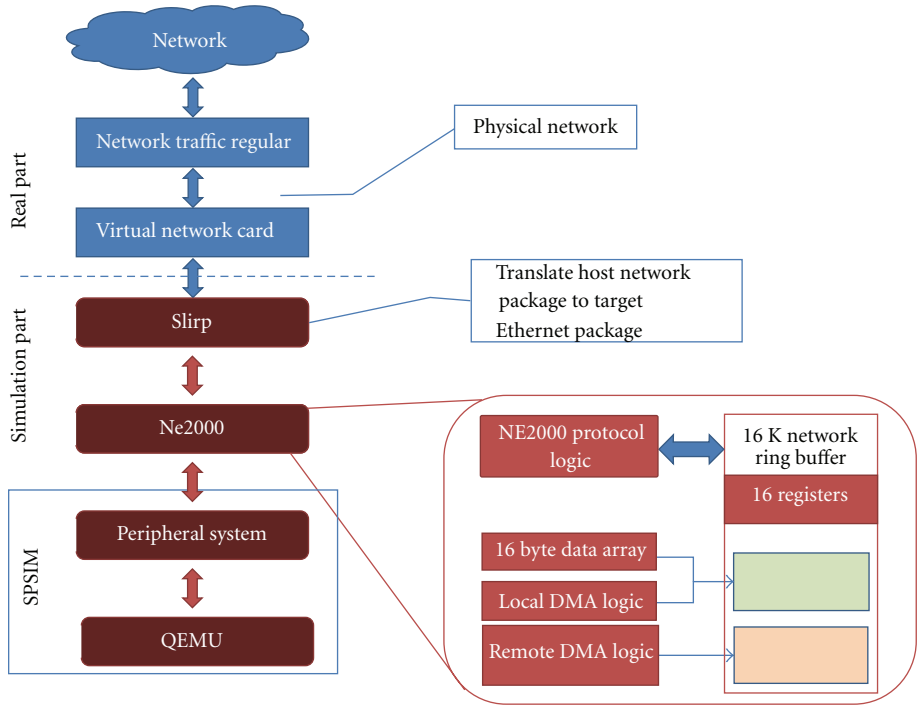

Figure 14: Diagram of the network traffic regulator.

priority. Suppose all three masters try to access the slaves at the same time. In the output channels associated with the two slaves, we can observe that the packages with higher priority are served first. Then the lower priority packages can be handled according to a round-robin protocol.

5.3. NE2000 Ethernet Interface. This section describes the implementation details of an NE2000-compatible Ethernet interface card [30]. Using the NE2000 SystemC module and a third-party application Slirp [31], SPSIM can connect to the real Internet.

The main task of NE2000 interface is controlling the data transmission between network and host. Our design philosophy is to cut down any unnecessary operations, because an embedded system only has limited computing resources.

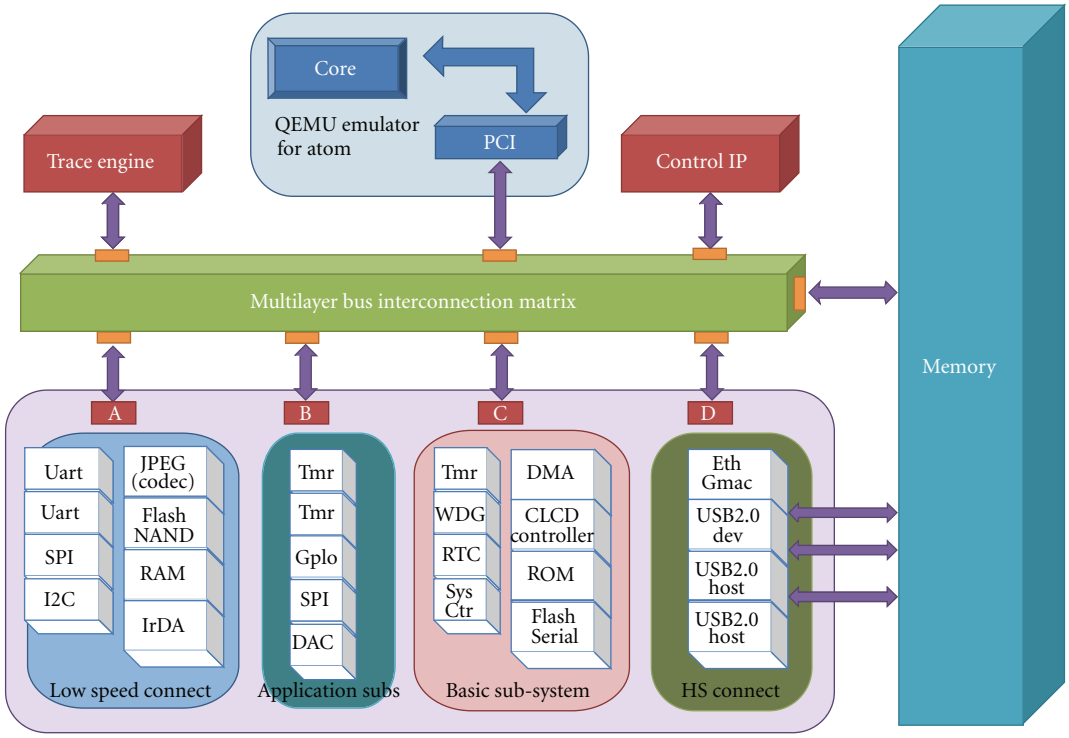

Figure 15: System diagram of the target SoC.

The processing flow of NE2000 model is illustrated in Figure 13. Upon receiving a data packet from the real network interface card on the host machine, the Slurp package translates the packet from the original format to target packet format. Then Slurp will deliver the translated packet to the SystemC-based NE2000 module. Our NE2000 module simulates every necessary step that needs to be done in a real network card. Such operations include updating descriptor ring, and sending DMA request, and so on. The NE2000 module also delivers the corresponding transactions onto the system interconnect and thus launches a series of transaction level processing steps. Here we only focus on behavior of NE2000 the standard communication processing with other peripheral modules in SPSIM is handled by generic module.

5.4. Network Traffic Regulator. As we are designing SoC for mobile devices, most mobile applications need to be evaluated in a mobile networking environment. Current mobile devices usually support multiple wireless networks, such as Wi-Fi, GPRS, EDGE, 3G, and 3G+. It is thus difficult to model so many wireless network interfaces in our simulator. In addition, it is also challenging to evaluate mobile applications under special scenarios, for example, a moving vehicle. In our work, we only implement a single NE2000 network interface as the fundamental mechanism of network connection. Meanwhile, we develop a tool, a so-called network traffic regulator, to emulate various mobile networks for SPSIM.

The network traffic regulator is a client/server-based network emulator. As illustrated in Figure 14, our network traffic regulator consists of two parts, virtual network card and network traffic regulator. The virtual network card is a virtualized network interface implemented on the simulation host. A leaky bucket algorithm [32] is implemented to limit bandwidth and business of the transmitted data. The network traffic regulator actually performs regulations on the traffic. It is connected to the simulation host through a high-speed link, such as 100M Ethernet. The regulator shapes the traffic from the simulator host according to the characteristic of the emulated wireless link.

Our network traffic regulator uses statistical models to emulate different wireless links under different scenarios. To be able to imitate a real wireless network, we first need to monitor mobile applications running on the network. Then the packet traces can be analyzed to extract the link characteristics under specific scenarios. The link characteristics will be represented as a loss and delay model. The loss-rate and delay are functions of packet rate, packet length and link state. This link model is implemented in the network traffic regulator. The network connection between the regulator and virtual network card is a high-speed link. Compared to the emulated wireless link, the loss rate and delay are nearly zero. Therefore we ignore the impacts of the high-speed link. The regulator maintains an input buffer and an output buffer for the uplink and downlink, respectively. When a packet arrives at the input or output buffer, the loss probability and delay time can be calculated according to the link model. Then the calculated loss probability determines whether a

packet needs to be discarded, while the delay time indicates how long the packet should be buffered.

Our network traffic regulator currently supports the emulation of Wi-Fi and 3G-WCDMA networks. For a given network, it can imitate various scenarios such as a 3G cell with a single subscriber and a busy network with several subscribers in the same cell.

5.5. Other Key Peripheral Modules. The DMA controller has two independent channels and realizes all basic functions as well as advanced functionalities such as scatter-gather. The DDRII memory module implements full read, write, and refresh functionalities defined in a DDRII protocol. The LCD module offers the normal display functionality and was enhanced to support file I/O during simulation.

6. Experiments and Validation

We have performed careful calibrations on our SoC model with measured data from real hardware. We run a set of micro-kernels on both the simulator and the real SoC chip so that we can use hardware data to fine-tune the parameters of the SoC model. We then simulated a group of real-world mobile computing applications. The simulation results are also compared with measured data to validate the accuracy and efficiency of SPSIM. At the end of this part, we provide a case study to demonstrate how SoC developers can take advantage of SPSIM to identify internal design details of interest.

6.1. Benchmark. Our experiments are performed on three real-world applications, MPEG4, G.729, and EFR. MPEG4 is a collection of patented methods for compressing digital audio and visual data. G.729 is an audio data compression implementation that compresses digital voice in packets of 10-millisecond duration. EFR, or Enhanced Full Rate, is a speech coding standard designed to improve the quality of GSM-Full Rate codec.

6.2. Performance Metrics. Performance of an application on a given architecture can be measured by instruction per cycle (IPC). After running the same application on both SPSIM and real hardware, the IPC values of both platforms can be derived. To evaluate the simulation accuracy, both IPC values can be plugged into (10) to extract the relative error. Here we use the average IPC of a whole application for the computation. For a higher precision, a set of IPC values for different stages of a program can also be utilized:

$$\Delta\text{IPC} = \frac{\text{IPC}_{\text{simulation}} - \text{IPC}_{\text{chip}}}{\text{IPC}_{\text{chip}}}. \quad (10)$$

To evaluate the simulation throughput of SPSIM, we selected the slowdown factor as the performance metric. As calculated in (11), the slowdown factor indicates how many times a simulated program execution is slower than the native execution. The variable simulation means how much time is spent in running a given application on SPSIM model. The other target indicates how much time should be used in target machine:

$$\text{Slowdown} = \frac{T_{\text{simulation}}}{T_{\text{target}}}. \quad (11)$$

Table 2: Accuracy of SPSIM.

Application	IPC measured on real chip	IPC by SPSIM	Relative error
MPEG4-light	1.41	1.28	−9.21%
MPEG4-middle	1.43	1.29	−9.79%
MPEG4-heavy	1.47	1.31	−10.88%
G.729-light	1.10	1.04	−5.45%
G.729-middle	1.12	1.07	−4.46%
G.729-heavy	1.14	1.18	3.51%
EFR-light	1.10	1.07	−2.72%
EFR-middle	1.14	1.12	−1.75%
EFR-heavy	1.15	1.17	1.74%

6.3. Experiment Results. All simulations are performed on Linux (Ubuntu 9.04) server with an Intel 2.53 GHz dual-cores processor and 4 GB memory. The SoC models are constructed in SystemC 2.2.0 and TLM 2.0 and compiled with gcc4.1.3.

To provide a comprehensive coverage on the system behavior, we organize the workload of the 3 applications introduced in the previous sub-section into three levels: light, medium, and heavy. In the MPEG4, we change occurrence of DMA operations from one transfer per frame, to two transfers per frame, and until four transfers per frame. We hope that using this method will enlarge impaction from system noise. For the other two applications, we change the location of input and output data. The light, medium, and heavy workloads are realized by (1) storing input and output data in L1 cache, (2) storing only output data in an external memory, and (3) locating both input and output data in an external memory, respectively. All data movements to and from an external memory are realized as DMA transfers.

Table 2 lists the IPC values of the three applications with different levels of workloads. This paper reports IPC values collected from both simulation and real chip. The relative simulation errors for each case are listed in the 4th column of Table 2. For G.279 and EFR, the difference ratio between the simulated and the measured IPCs is within ∼5%. For the MPEG4 applications, the difference is slightly over 10%. The relatively higher level of mismatch is caused by the minor disparity between the simulated and real arbitration mechanisms. The overall accuracy is satisfying (Figure 16).

In Table 3, we compare the run-time results collected from simulation and real hardware. It should be noted that SPSIM includes both functional and event-driven, performance simulations. Since the functional simulation only needs to be executed once for a given application, the

Table 3: Speedup between real chip and SPSIM.

Application name	Simulation time	Target time	Slowdown
MPEG4-light	141	4.12	**34.22**
MPEG4-middle	139	4.14	**33.57**
MPEG4-heavy	138	4.23	**32.62**
G.729-light	1.23	3.35	**0.37**
G.729-middle	26.62	4.43	**7.76**
G.729-heavy	53.17	4.44	**15.46**
EFR-light	0.78	1.57	**0.50**
EFR-middle	5.16	1.57	**3.28**
EFR-heavy	9.13	1.56	**5.86**

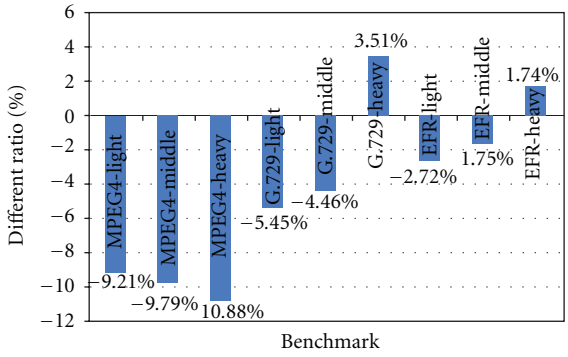

Figure 16: Relative simulation error with regard to measured data on real hardware.

simulation speed is determined by the performance simulation stage, especially when there are a large number of SoC configurations to be evaluated. Accordingly, we only report the simulation time of the SystemC-based performance simulation.

From Table 3, it can be seen that all simulations have a slowdown factor less than 35. Compared to state-of-the-art simulators, ASIM [8] has a slowdown factor of about 10, 000x, while FAST [7] achieves a level of 20∼30x through FPGA-based acceleration hardware.

One interesting observation from Table 3 is that the simulations of G.729 light and EFR light are even faster than their native execution on the target hardware. It is mainly due to a higher percentage of regular instructions in the input trace. As explained in Section 3.2, the processing of continual regular instructions in SPSIM is collapsing into one block to be simulated.

Another notable observation is that the run time of MPEG4 does not change much with different workloads, while the execution times of the other two applications vary dramatically on different workloads. The underlying reason is the good cache behavior of the MPEG4 applications. In fact, most required memory data are already cached and will not incur activities on the system interconnect and other peripherals.

We also use our power estimation tool to evaluate the energy consumption on the system interconnect fabric. Figure 17 lists the results with varying number of DMA operations for each frame of data in different applications.

In fact, from the above discussion, it can be seen that the SPSIM simulator can greatly help designers expose internal details of a SoC design and thus identify performance bottlenecks. Next subsection will show how we use SPSIM to explore a real scenario.

6.4. Microscope for SoC. In this sub-section, we provide a case study to demonstrate the power of SPSIM to identify SoC design details. With SPSIM, we are able to boot a complete Android operating system [21] and then run a browser, that is, WebKit [33]. As a result, SPSIM can simulate a typical user experience of surfing web pages on a mobile communication platform. Under such a context, SPSIM serves as a microscope to identify the memory, network, and bus behaviors during a complete Internet session.

Figure 18 is a detailed profiling of the memory access in system memory and network traffic from booting Android OS to launching a browser until loading a complete page. In the top drawing, the browser is launched to load one page automatically, while in the bottom drawing the browser is idle. After analysis, we can get the following conclusions.

A few important observations can be made in Figure 18. First, there are three bursts of network access traffic in the top drawing. Compared to the two drawings, it is easy to find that the first two bursts are triggered by booting operations such as configuring the TCP/IP stack. And the last group of traffic is generated by loading page. Reader can find that the memory access drops down sharply in the same time frame when there is no network access in the bottom drawing.

Besides analyzing the overall trend, we can also enlarge the last peak for a localized examination in Figure 19. First, by checking the number of Ethernet instructions (labeled as "Ethernet_inst" in Figure 19, which control all network activities and the size of memory accesses, the developer can easily identify that every Ethernet instruction will bring a 4-byte data transfer in the network interface. It is also very interesting that the sizes of memory read and write are basically identical. After careful analysis on both software and hardware, it can be identified that the write data consists of two parts, configuring information to be written into control register of the network interface and the user data to be sent out to the remote server for better user experience. For IP-based SoC designs, such information is essential to help designers understand the behavior of target applications and perform optimizations accordingly.

7. Conclusion and Future Work

In this work, we developed an efficient system-level simulation framework. By integrating leading-edge transaction level modeling techniques, our simulation framework can attain an accuracy level that in the past can only be achieved through cycle-accurate simulations. The high

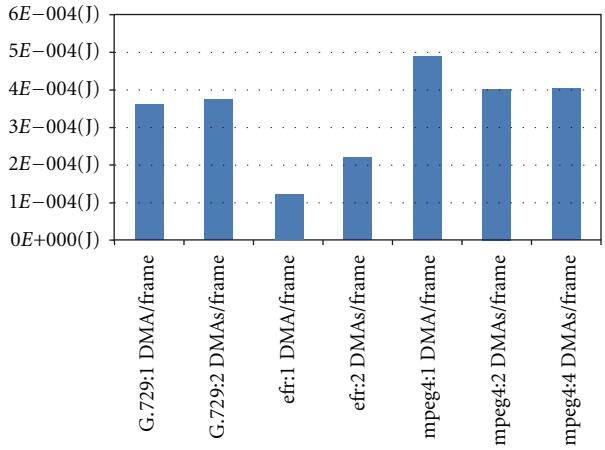

Figure 17: Bus energy consumption of different applications, different architecture.

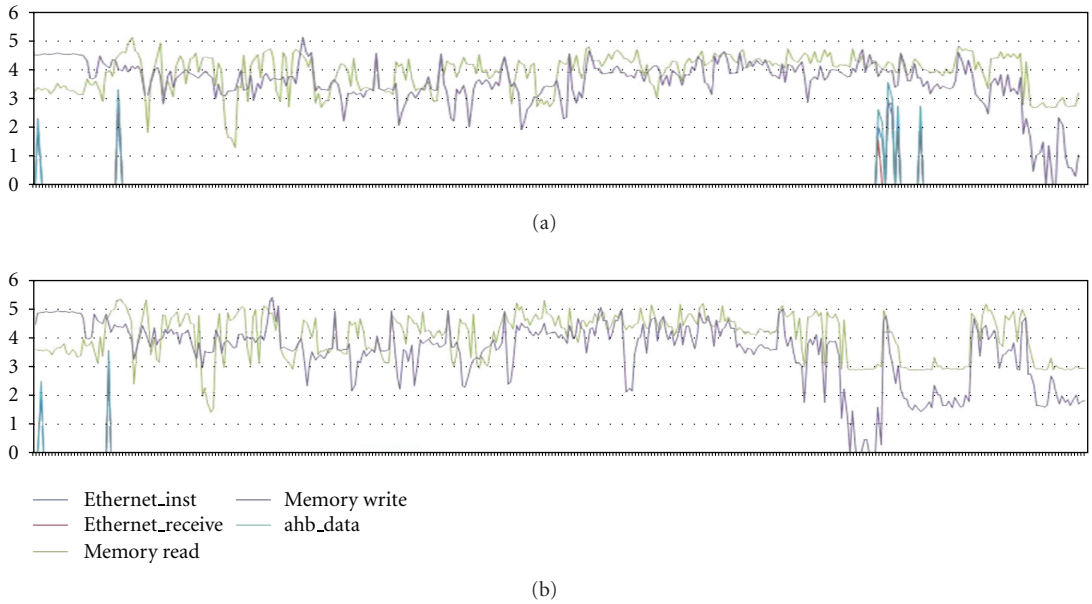

Figure 18: Memory and network actives when opening one page after booting whole operation system.

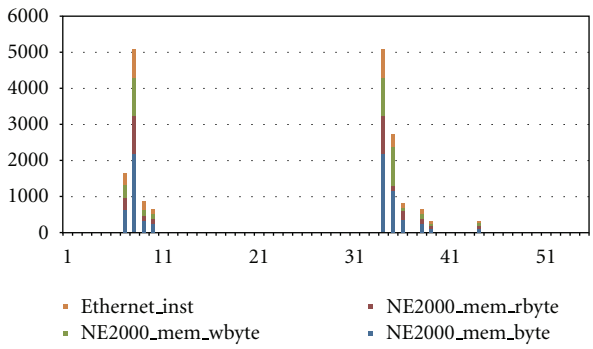

Figure 19: Detail of memory access of loading one page.

simulation throughput, that is, within 35 times of native execution, enables SoC architects to quickly go through the large solution space and determine an optimized architecture. Furthermore, the simulation framework also provides generic templates to help developer build models for new peripheral/accelerators in a more efficient manner. A power estimation model and an effective methodology are also developed to evaluate energy efficiency of a given SoC. The SPSIM simulation framework can be used for SoC architecture design and early software development.

In the future, we are going to extend SPSIM in several directions. One direction is to extend the simulator to capture the complex behaviors of heterogeneous multicore platforms. Especially, we will cover the operations of graphic processing units in our simulator so that a complete cell phone system can be simulated. Meanwhile, we are also exploring parallel simulation techniques (e.g., [34]) to further improve the simulation speed.

References

[1] A. Jerraya and W. Wolf, *Multiprocessor Systems-on-Chips*, Systems on Silicon Series, Morgan Kaufmann, Waltham, Mass, USA, 1st edition, 2004.

[2] S. Pasricha and N. Dutt, *On-Chip Communication Architectures: System on Chip Interconnect*, Systems on Silicon Series, Morgan Kaufmann, Waltham, Mass, USA, 2008.

[3] K. Bennett and P. Layzell, "Service-based software: the future for flexible software," in *Proceedings of the Asia-Pacific Software Engineering Conference*, pp. 214–221, December 2000.

[4] Open SystemC Initiative (OSCI), "OSCI TLM-2.0 user manual".

[5] T. Austin, E. Larson, and D. Ernest, "SimpleScalar: an infrastructure for computer system modeling," *Computer*, vol. 35, no. 2, pp. 59–67, 2002.

[6] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt, "The M5 simulator: modeling networked systems," *IEEE Micro Special Issue on Architecture Simulation and Modeling*, vol. 26, no. 4, pp. 52–60, 2006.

[7] D. Chiou, D. Sunwoo, J. Kim et al., "FPGA-accelerated simulation technologies (FAST): fast, full-system, cycle-accurate simulators," in *Proceedings of the 40th IEEE/ACM International Symposium on Microarchitecture, (MICRO '07)*, pp. 249–261, December 2007.

[8] J. Emer, P. Ahufa, E. Borch et al., "Asim: a performance model framework," *Computer*, vol. 35, no. 2, pp. 13–76, 2002.

[9] M. Caldari, M. Conti, M. Curaba, L. Pieralisi, and C. Turchetti, "Transaction-level models for AMBA bus architecture using systemC 2.0," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, (DATE '03)*, pp. 26–31, 2003.

[10] W. Klingauf, "Systematic transaction level modeling of embedded systems with systemC," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, (DATE '05)*, pp. 566–567, March 2005.

[11] W. Klingauf, R. Günzel, O. Bringmann, P. Parfuntseu, and M. Burton, "GreenBus: a generic interconnect fabric for transaction level modelling," in *Proceedings of the Design Automation Conference, (DAC '06)*, pp. 905–910, 2006.

[12] A. Bona, V. Zaccaria, and R. Zafalon, "System level power modeling and simulation of high-end industrial network-on-chip," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, (DATE '04)*, pp. 318–323, February 2004.

[13] QEMU, "Open source processor emulator," http://www.qemu.org.

[14] Skyeye, "Open source ARM processor simulator," http://www.skyeye.org.

[15] ARM Limited, *ARM Developer Suite Debug Target Guide Version 1.2*.

[16] Uclinux, "A simple Linux kernel for embedded," http://www.uclinux.org.

[17] S. Nussbaum and J. E. Smith, "Modeling superscalar processors via statistical simulation," in *Proceedings of the Internatinal Conference on Parallel Architectures and Compilation Techniques, (PACT '01)*, pp. 15–24, September 2001.

[18] L. Eeckhout,, K. D. Bosschere, and H. Neefs, "Performance analysis through synthetic trace generation," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, pp. 1–6, August 2000.

[19] "Intel® Atom™ Processor," http://www.intel.com/technology/atom/.

[20] Android OS, "An light-weighted OS for mobile device developed by Google Corp," http://www.android.com/.

[21] S. J. Eggers, J. S. Emer, H. M. Levy, J. L. Lo, R. L. Stamm, and D. M. Tullsen, "Simultaneous multithreading: a platform for next-generation processors," *IEEE Micro*, vol. 17, no. 5, pp. 12–19, 1997.

[22] ARM, "AMBA AXI Protocol version: 2.0," http://infocenter.arm.com/help/index.jsp.

[23] Synopsys, *DesignWare Databook Version 1.12a*, 2008.

[24] A. Varma, E. Debes, I. Kozintsev, P. Klein, and B. Jacob, "Accurate and fast system-level power modeling: an XScale-based case study," *Transactions on Embedded Computing Systems*, vol. 7, no. 3, article 25, 2008.

[25] M. Caldari, M. Conti, M. Curaba, L. Pieralisi, and C. Turchetti, "System-level power analysis methodology applied to the AMBA AHB bus," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, (DATE '03)*, pp. 32–37, 2003.

[26] PrimeTime® PX User Guide, Version C-2009.06.

[27] S. N. Adya and I. L. Markov, "Consistent placement of macro-blocks using floorplanning and standard-cell placement," in *Proceedings of the International Symposium on Physical Design, (ISPD '02)*, pp. 12–17, April 2002.

[28] C. Chu, "FLUTE: fast lookup table based wirelength estimation technique," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design, Digest of Technical Papers, (ICCAD '04)*, pp. 696–701, November 2004.

[29] R. Ho, K. W. Mai, and M. A. Horowitz, "The future of wires," in *Proceedings of the IEEE*, pp. 490–504, August 2002.

[30] Wikipedia, "NE2000," http://en.wikipedia.org/wiki/NE2000.

[31] "Slirp, the PPP/SLIP-on-terminal emulator," http://slirp.sourceforge.net/.

[32] A. S. Tanenbaum, *Computer Networks*, 4th edition, 2003.

[33] WebKit, "An open source web browser engine," http://webkit.org/.

[34] B. Wang, Y. Zhu, and Y. Deng, "Distributed time, conservative parallel logic simulation on GPUs," in *Proceedings of the 47th Design Automation Conference, (DAC '10)*, pp. 761–766, June 2010.