Avalanche: An Environment for Design Space Exploration and Optimization of Low-Power Embedded Systems

Jörg Henkel, *Senior Member, IEEE* and Yanbing Li, *Member, IEEE*

***Abstract*—Power estimation and optimization has become a key issue in embedded system design, especially in the rapidly growing market of mobile handheld computing, communication, internet devices that are driven by battery power. It is of paramount importance to estimate and optimize power of those systems during an early design stage at a high level of abstraction in order to efficiently explore the design space and to take full advantage of the related high optimization potential.**

In this paper, we present *Avalanche*, a prototyping framework that addresses the issues of power estimation and optimization for mixed hardware and software embedded systems. *Avalanche* is based on a generic embedded system architecture consisting of embedded CPU, custom hardware, and a memory hierarchy. For system-level power estimation, given various system parameters like cache sizes, cache policies, and bus width, etc., Avalanche is able to rapidly evaluate/estimate power and performance and thus facilitate comprehensive design space explorations. For system-level power optimization, Avalanche offers different modes reflecting various design scenarios: if no hardware/software partitioning or only partial partitioning has been conducted, Avalanche guides the designer in finding power-aware hardware/software partitioning; when a system has already been partitioned, Avalanche can optimize system parameters such as cache and memory size; if system parameters and partitioning are given, Avalanche applies additional optimizations for power including source-to-source compiler transformations.

Avalanche has been deployed during the design phase of real-world applications including an MPEG II encoder in a set-top box design. Extensive design space explorations in terms of power and performance could be conducted within several hours and various optimization techniques led to power reductions of up to 94% without performance losses and only a slight increases in total chip size (i.e., transistor count).

***Index Terms*—Design space exploration, low-power design, power/performance tradeoff, system-level design.**

I. Introduction

As feature sizes within integrated circuits get smaller and smaller, reducing power consumption is becoming one of the most important design issues in deep submicron designs. There are several reasons for this tendency. First, heat-induced electromigration has a much higher impact on the correct functionality of deep submicron designs since a single electron carries an increasing relative amount of electrical energy (due to the smaller number of electrons needed to switch a transistor in a deep submicron design). Second, environmental issues force chip producers to reduce power since the billions of microprocessors operating today (desktop system plus embedded systems) together consume a significant amount of energy considering that just a single high-end microprocessor can consume close to 100 W of power. Finally, power consumption is key for the design of mobile handheld computing/communication/internet devices that are driven by battery power. This paper focuses on the latter case, i.e., reducing power consumption for battery-powered devices.

These mobile handheld computing/communication/internet devices (in the following we will denote them simply as *mobile devices*) are becoming increasingly complex since they may incorporate features within a single device that formerly could only be manufactured in two or more separate devices (for example, the combination of a cell phone and a PDA). As a result, these mobile devices require up to several hundreds of million transistors to implement the enhanced functionality. Since cost is a crucial factor, integrating system functionality into as few as possible chips is the ultimate aim. Ideally, those devices would require just one single chip that integrates the whole system [system-on-a-chip (SOC)]. In practice, however, mobile devices comprise more than just a single chip, typically, three to five chips (for a 3G cell phone, for example). Although today's silicon technology allows more than 400 000 000 transistors [2] to be integrated on an SOC, existing design methodologies and design tools can not effectively handle them. This is known as the *design productivity gap* [1]. The most promising solution to overcome this disadvantage is the so-called *core-based design methodology* where key parts of a large design (i.e., cores) are reused [1] for different designs. The EDA industry as well as many research groups are currently focusing on novel design methodologies to facilitate core-based design under various design constraints. Though there are already many commercial companies offering core libraries (as *soft, firm,* and *hard* cores[1]) seamless commercial design flows are hardly available. Especially, as of this date, there is no commercial system-level design tool available that addresses core-based design with respect to power estimation/optimization as it is needed for the abovementioned several hundred million SOCs for future mobile devices.

We have conducted research on a prototyping system that allows an SOC designer in an early design stage to explore, estimate and optimize the power consumption within the

Manuscript received November 9, 2000; revised September 20, 2001.
J. Henkel is with NEC Laboratories America, Princeton, NJ 08540 USA (e-mail: henkel@nec-lab.com).
Y. Li is with Synopsys, Mountainview, CA USA.
Digital Object Identifier 10.1109/TVLSI.2002.800524

[1]For a definition please refer to [3].

paradigm of core-based design methodology. Our research shows that designing a whole system for low power is not as simple as selecting low-power cores and integrating them into one low-power SOC. Rather, there are tight interdependencies between various system components (i.e., cores) in terms of power consumption. Hence, optimizing for overall system power requests a sophisticated approach that: 1) explores the power consumption of a single core within the context of other cores (i.e., system) that it communicate with; 2) enables power estimation assuming various sets of different system parameters like cache sizes, cache policies, main memory size, etc.; and 3) evaluates and performs power optimization at a high-level of abstraction. Our *Avalanche* low-power prototyping system-level tool is aiming to address these issues.

This paper is structured as follows. The next section gives an overview of related work. Section III describes the energy estimation models and the estimation flow in Avalanche. How to use Avalanche for design space exploration is explained in Section III-D. Sections V and VI describe our power optimization techniques used in the *Avalanche* framework: Section V describes power optimization through partitioning; Section VI describe our source code power optimization as well as power estimation through adapting/selecting appropriate system parameters. Finally, Section VII shows how all estimation and optimization parts are integrated into the whole simplified design flow of Avalanche.

II. Related Work

Power estimation and optimization has been studied at various levels of abstraction including gate-level, RTL-level, system-level. Our focus is on the system-level. Previous work has studied low power issues from both hardware and software point of views. Since our system specification consists of mixed hardware and software components, both aspects are relevant.

Starting with software power estimation, Tiwari and Malik [15] investigated the power consumption during the execution of programs running on different processor cores. Ong and Ynn [16] showed that the energy consumption may drastically vary depending on the algorithms running on a dedicated hardware. A power and performance simulation tool for a RISC design has been developed by Sato *et al.* [17]. Their tool can be used to conduct architecture-level optimizations. Kandemir *et al.* [10] present in their work a compiler optimizations for low power software. An analysis of the power consumption characteristics of a real-time operating system has been investigated by Dick *et al.* [11]. Instruction code compression to reduce the power consumption of embedded systems has been investigated by Benini *et al.* [12] and Lekatsas *et al.* [13].

In the area of hardware power estimation, Hsieh *et al.* [30] investigated the power consumption of high-performance microprocessors and derived specific software synthesis algorithms for low power. The work reported by Landman and Rabaey [31] deals with an architectural-oriented power minimization approach. Gonzales and Horowitz [18] explored the power consumption of different processor architectures (pipelined, unpipelined, super-scalar). Kamble and Ghose [19] analyzed cache energy consumption. Itoh *et al.* studied SRAM and DRAM energy consumption and low power RAM design techniques [27]. Panda *et al.* [20] presented a strategy for exploring on-chip memory architecture in embedded systems with respect to performance only. Optimizing energy consumption by means of high-level transformations has been addressed by Potkonjak *et al.* [21].

There has been some recent work addressing system-level power estimation/optimization: Dave *et al.* [4] introduce a task-level codesign methodology that optimizes for power consumption and performance. The influence of caches is not taken into consideration. The procedure for task allocation is based on estimations for an *average* power consumption of a processing element. The approach described by Hong *et al.* [29] uses a multiple-voltage power supply to minimize system-power consumption.

Another system-level power estimation approach that focuses on peripheral cores within SOCs is described by Givargis *et al.* [5]. They presented a hybrid approach that uses one-time obtained gate-level power data and propagates it to an executable specification in order to speed up power estimation. Simunic *et al.* [6] simulated the power consumption of an ARM processor plus a cache hierarchy and a main memory using a cycle-accurate approach. Lajolo *et al.* [7] have conducted research on a cycle-based cosimulation environment for power estimation. A generic power efficient scheduling method for fixed priorities is presented by Shin *et al.* [8]. Benini *et al.* [9] reduce power consumption of a system through adapted encoding of the data transmitted via the interfaces. The interface structure is also the target of the work by Fornaciari *et al.* [14]. They estimate power consumption of buses based on cache parameters.

Contribution and Focus of Our Work: The main contributions of our work are a comprehensive approach for system-level power estimation and a power optimization approach using hardware/software partitioning that uses the estimation part of the framework. We do take into consideration the mutual effects that certain system parameters (like cache size, main memory size, etc.) have on each other in terms of power and performance. The partitioning approach is the first of its kinds for low-power fine-grained hardware/software partitioning.

III. Power Estimation Models

This section describes the power estimation models used in Avalanche. We will first briefly introduce the underlying generic architecture and then describe the power models and finally show the estimation related design flow in Avalanche.

The generic[2] architecture used by Avalanche is shown in Fig. 1. It comprises a processor core, an instruction cache, a data cache, a main memory, and a custom hardware parts (ASIC).[3] The custom hardware and the program running on the CPU[4] are not necessarily fixed yet. In fact, as will be shown in Section V,

[2]We call the architecture *generic* because there are various system parameters (as introduced in the following) that actually instantiate the system based on this generic architecture.

[3]Please note that more than one ASIC block can exist. The figure is simplified in this sense.

[4]We currently use a SparcLite processor in our model. However, we can plug-in any other type of processors, provided a power-model enhanced instruction-set simulator for that type of processors.

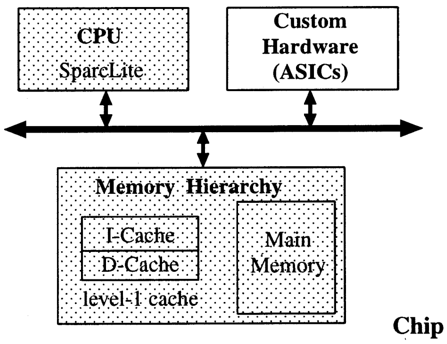


Fig. 1. Generic simplified architecture as a basis for energy/power exploration/estimation/optimization of our Avalanche system.

partitioning between hardware and software can be explored in our system to optimize for power consumption.

A. An Analytical Cache Power Model

We use a cache energy model based on transistor-level analysis. The building blocks of the model (see Fig. 2 are an input decoder, a tag array and a data array. Attached to the tag array are column multiplexers whereas data output drivers are attached to the data array. A SRAM cell in data and tag array comprises a standard CMOS transistors cell with six transistors. The switching capacitances in the equations derived below, are obtained by running the tool *cacti* [22].[5]

Only the energy portions in the bit lines for read and write ($E_{bit,rd}$ and $E_{bit,wr}$), in the word lines ($E_{word,rd/wr}$), in the decoder (E_{dec}) and in the output drivers (E_{od}) contribute essentially to the total energy. The according effective capacitances are as follows:

$$C_{bit,rd} = N_{bitl} \cdot N_{rows} \cdot (C_{SRAM,pr} + C_{SRAM,rd}) + N_{cols} \cdot C_{pr\_logic} \quad (1)$$

where $C_{SRAM,pr}$, $C_{SRAM,rd}$ and C_{pr_logic} are the capacitances of the SRAM cell affected by precharging and discharging and the capacitance of the precharge logic itself, respectively. N_{rows} is the number of rows (number of sets) in the cache. The number of bit lines is given by N_{bitl}:

$$N_{bitl} = (T \cdot m + St + 8 \cdot L \cdot m) \cdot 2$$
$$N_{cols} = m \cdot (8 \cdot L + T + St)$$

where m means an m-way set associative cache, L is the line size in bytes, T is the number of tag bits and St is the number of status bits in a block frame. $C_{bit,wr}$ is defined in a similar manner as $C_{bit,rd}$.

The effective wordline capacitance is given by

$$C_{word} = N_{cols} \cdot C_{word,gate} \quad (2)$$

where $C_{word,gate}$ is the sum of the two gate capacitances of the transmission gates in the six-transistor SRAM cell. For simplification, we do not include the equations for C_{dec} and C_{od} here. Apparently, the switched capacitance is directly related to the cache parameters (2). For the above equations we can see

[5]Note that *cacti* in its original version does not output capacitance. Rather, it is a tool for calculating delay times. We made slight modification to *cacti* and obtained various capacitances we needed for the cache model.

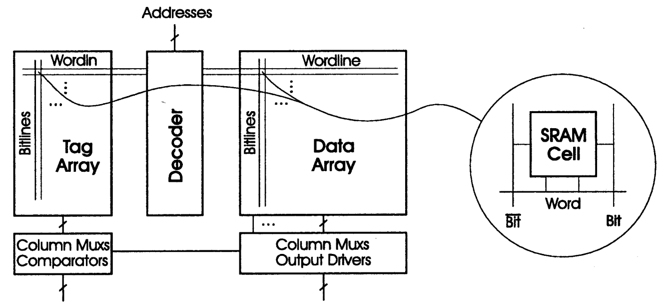


Fig. 2. Cache model.

that the switched capacitance during each cache access is directly related to the cache parameters such as size, line size, and associativity.

Finally, the total energy consumed within the cache (i-cache or d-cache) during the execution of a software program is related to the number of total cache accesses N_{acc}, as well as the number of hits and misses for cache reads and writes

$$E_c = \frac{1}{2} \cdot V_{DD}^2 \big(N_{acc} \cdot C_{bit,rd} + N_{acc} \cdot C_{word} + a \cdot C_{bit,wr} + b \cdot C_{dec} + c \cdot C_{od} \big) \quad (3)$$

where a, b and c are complex expressions that depend on read/write accesses and, in parts on statistical assumptions. $a \cdot C_{bit,write}$, $b \cdot C_{dec}$ and $c \cdot C_{od}$[6] are the effective capacitances to switch when writing one bit, during decoding of an access and during output, respectively.

The implemented cache model has a high accuracy (compared to the real hardware) since every switching transistor within the cache has been taken into consideration. All the capacitances are obtained by running *cacti* [22] and are derived for a 0.8 μm CMOS technology.[7] The calculation of the capacitances within *cacti* has been proofed against a *Spice* simulation.

B. Analytical Main Memory Power Model

For power/energy analysis of the main memory, we use a model of a DRAM as described by Itoh *et al.* [27]. The energy source for DRAM mainly includes: the RAM array, the column decoder, the row decoder, and peripherals.

$$I_a = m \cdot i_{act} + m(n-1) \cdot i_{hld} + m \cdot i_{dec} + n \cdot i_{dec} + I_{peri}. \quad (4)$$

Equation (4) shows the current drawn during each memory access. Note that during each access, m cells are selected. $m \cdot i_{act}$ is the active current of the m selected cells. $m(n-1) \cdot i_{hld}$ is the data retention current of the $m \cdot (n-1)$ cells that are not selected. $m \cdot i_{dec}$ and $n \cdot i_{dec}$ are the currents drawn on column and row decoder, respectively. I_{peri} represents the current on peripheral circuits. The equations show that the energy consumption of each memory access is directly related to the size of the

[6]The capacitances of the output drivers are derived for an on-chip cache implementation, i.e., we assume that all resources like processor, cache and main memory are implemented on a single chip.

[7]Note that is is a rather old technology. Results using our 0.18 μ and 0.10 μ cannot be disclosed for patent-related issues. However, the general oberservations and conclusions derived throughout this paper do not depend on the technology since it basically results in a certain scaling of the obtained power/energy numbers only.

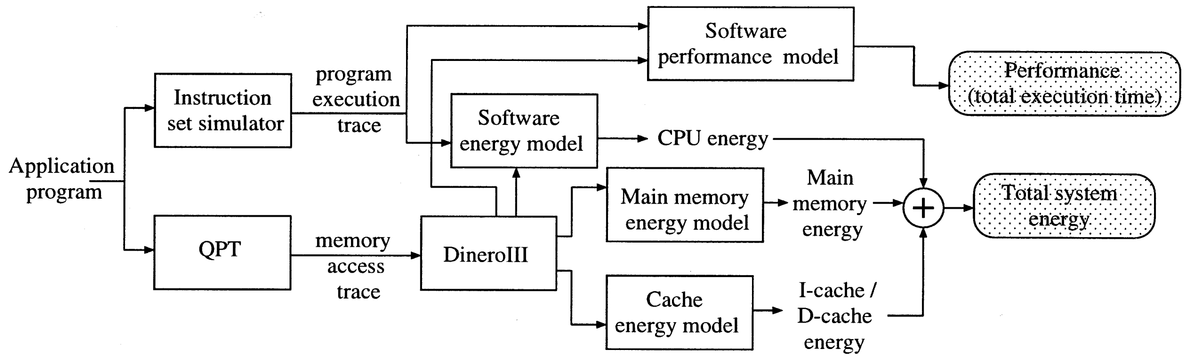


Fig. 3. Design flow of the estimation part of our *Avalanche* framework.

memory. For the total energy consumption, i_{active} is the dominating component. At high clock frequencies, i_{hld} is negligible [27].

C. Software Power and Performance Model

For software energy/power estimation we deploy an instruction set simulator (ISS) developed by Ye *et al.* [28] and enhanced it by values of the current drawn during the execution of an instruction. Those current values are obtained from [25].

The total SW program energy is

$$E_{prg} = T_{w\_c} \cdot V_{DD} \cdot \sum_{i=0}^{N-1} (I_{instr,i} \cdot N_{cyc,i}) + T_{cyc} \cdot V_{DD} \cdot (\underbrace{N_{miss,rd} \cdot N_{cyc,rd\_pen} \cdot I_{instr,nop}}_{\text{data read miss penalty}} + \underbrace{N_{miss,wr} \cdot N_{cyc,wr\_pen} \cdot I_{instr,nop}}_{\text{data write miss penalty}} + \underbrace{N_{miss,fetch} \cdot N_{cyc,fet\_pen} \cdot I_{instr,nop}}_{\text{instruction fetch miss penalty}}) \quad (5)$$

where V_{DD} is the voltage supply, I_{instr} is the current that is drawn during the execution of instruction i at the processor pins, $N_{cyc,i}$ is the number of cycles the instruction needs for execution, T_{cyc} is the cycle time, and N is the total number of instructions of the program. T_{w_c} is the execution time of the application assumed that there is a cache as specified.

The three additional portions within the brackets refer to the energy consumed during the penalty cycles due to a data cache write miss, a data read miss and an instruction fetch miss, respectively. We assume that the energy consumed within processor is negligible at times when all programs/processes are terminated. This can be accomplished through shutting down the processor during idle times. An additional assumption is that we do not account for any energy/power consumption that might arise even at times when no switching activity occurs (like power consumption related to leakage current).

Let T_{w/o_c} be the execution time of a program running on the processor core (simulated by an ISS) without cache, the corrected execution time (i.e., including cache behavior) is estimated by

$$T_{w\_c} = T_{w/o\_c} + T_{cyc} \cdot (N_{miss,rd} \cdot N_{cyc,rd\_pen} + N_{miss,wr} \cdot N_{cyc,wr\_pen} + N_{miss,fet} \cdot N_{cyc,fet\_pen}). \quad (6)$$

D. Power Estimation Flow in Avalanche

By using the above energy models and timing models, the estimation design flow (the power optimization parts are not shown) of our system is shown in Fig. 3. The input is an application program. It is fed into the instruction set simulator of the target processor that simulates the program and delivers a program trace to the *software energy model* and the *software performance model*. At the mean time, the input program is also fed into the memory trace profiler *QPT* [26], which generates the memory access trace to be used by *Dinero* [26]. *Dinero* provides the number of demand fetches and demand misses (for data and instructions). These numbers are then used: by the *software performance model* to obtain the total execution time with cache miss penalty considered (6), by the *software energy model* to adjust the software energy with the stalls caused by cache misses (5), and by the *cache* and *main memory energy models* [(3) and (4)] to calculate the energy consumption by the memory components based on the actual number of instruction/data cache accesses and main memory accesses.

IV. POWER/PERFORMANCE DESIGN SPACE EXPLORATION

Using the power estimation model described in the previous section, Avalanche provides a system's designer with the capability to explore and visualize the design space. This section focuses on how to use Avalanche in design space exploration. Unlike in the Sections V and VI, here no optimization is applied. The designer simply specifies which design parameters (see Section III) are fixed and which are variables, plus the range of the nonfixed variables. Our system can visualize the design space based on these specifications. We only provide a brief overview of the design space exploration capabilities since the main focus of this paper is on optimizations (see Sections V and VI).

Fig. 5 shows the design space exploration results for three applications. For each application, there are two figures: the figure on the left shows the total energy consumption of the whole

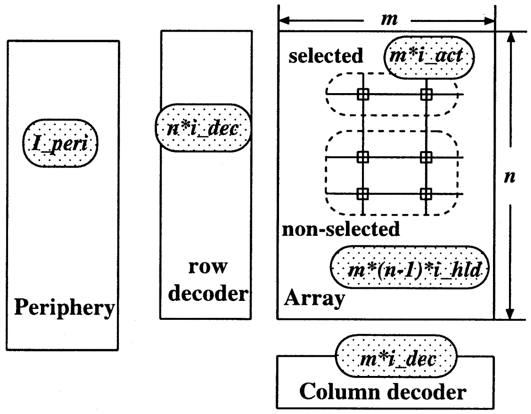


Fig. 4. Source of power consumption in main memory (DRAM).

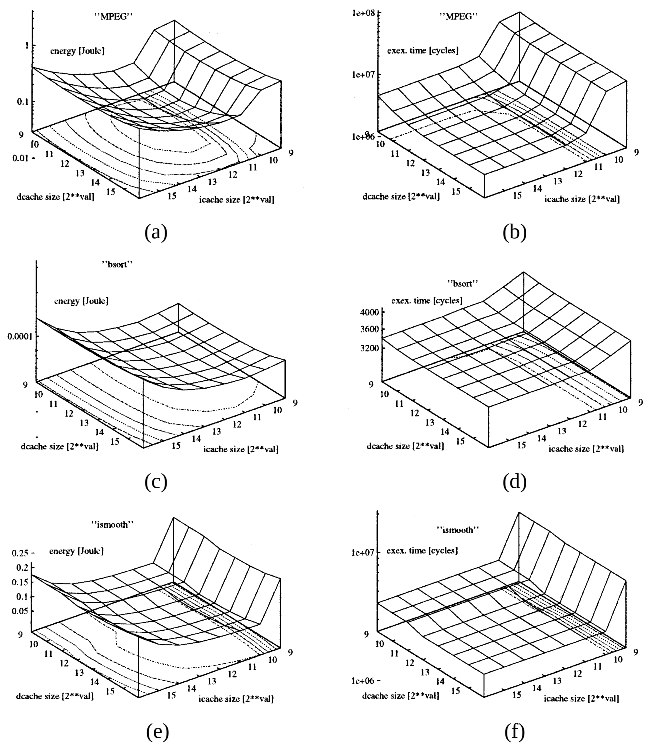


Fig. 5. Energy and execution time versus instruction and data cache size, for the applications *MPEG*, *bsort*, and *ismooth*.

system; the one on the right shows the application execution time on the target architecture in clock cycles. The varying parameters in each figure are the data cache size (left axis) and the instruction cache size (right axis). All other parameters are fixed for these examples. A number of 2 on the data or instruction cache axis refers to a cache size of 4K ($= 2^{12}$) byte.

Results of the MPEG encoder are shown in Fig. 5(a) and (b). Here, both large and small cache sizes lead to a high system energy consumption. In case of large caches, the caches' energy dominates the system energy; while in case of smaller cache sizes, the software energy is dominating—from (5), large number of cache misses (due to small caches) result in poor performance and therefore large energy consumed in the miss penalty cycles. The right figure shows in fact that the program execution time raises drastically for small instruction cache sizes ($<$ 1024 byte). Remarkably, the least energy consuming

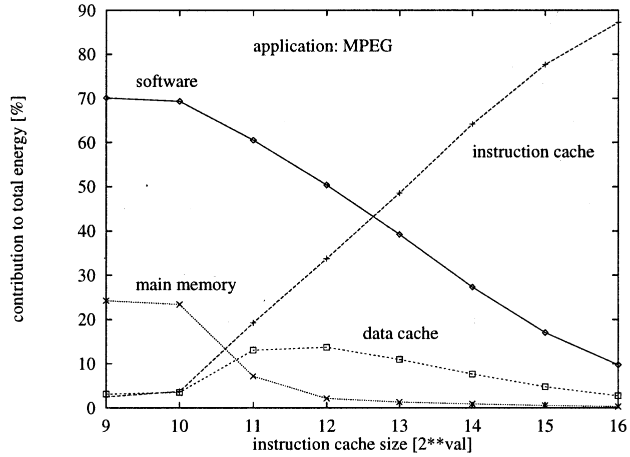


Fig. 6. Contribution of software, d-cache, i-cache, and main memory to total energy consumption at a fixed data cache size of 4k.

configuration (data cache size and instruction cache size 4k each) is also one of those with the highest performance (i.e., small number of clock cycles). This is a behavior that would possibly not be expected (and is not the case for the other applications). In addition, Fig. 6 reveals the contribution (in percentage) of each component to the whole system energy consumption (i.e., software program, caches, main memory). In that figure, the data cache size has been fixed whereas the instruction cache size varies.

The experiments conducted with the *bsort* [Fig. 6(c) and (d)] application (i.e., bubble sort) show mainly that there is almost no dependency on data cache size in terms of system energy consumption and system performance. This is due to the small data size used in this application. More dependencies can be observed by changing the instruction cache size. Obviously, a large instruction cache size leads to a large system energy consumption also. But as opposed to the *MPEG* encoder, a small instruction cache size does not lead to a larger system energy consumption as a consequence of a larger program execution time. Rather than that, the performance decreases (more cycles due to the mid-right figure in Fig. 5. The last example is *ismooth*, [Fig. 5 (e) and (f)], an image smoothing application, which shows yet another different type of behavior.

One important observation of the power/performance exploration is that it is difficult to predict system energy consumption and performance when system parameter change. Powerful tools are needed for both power analysis and optimizations. For example, whether a larger or smaller cache size leads to a smaller power consumption cannot be easily be determined, unlike the performance, where larger cache size always leads to higher or at least equal performance. Similar scenarios can be discussed using other parameters. Avalanche can either be used for simple design space exploration as a pure design aid to a designer or can be used in conjunction with optimization strategies as described in the upcoming sections.

V. Power Optimization Through Hardware/Software Partitioning

In some design cases, partitioning between hardware and software is given *a priori*. One reason, for example, is that a designer has very tight design time constraints that do not allow any further optimization. Rather, the designer *re-uses* as many as possible existing IPs (software code or cores) that stem from previous projects or provided by an IP vender. However, an SOC product that is entirely designed through the composition of (commodity) IPs most likely will not have performance or power advantage compared to products with custom components. If there are strict performance and power constraints for a particular design, the key cores of the design usually need to be re-designed. One powerful method in reducing power consumption is a sophisticated hardware/software partitioning approach that considers power as a design metric.

In this section, we introduce our hardware/software partitioning approach used in the Avalanche system. It is the first fine-grained (instruction/operation-level) approach to address low-power optimizations (please note that some work has been conducted on task-level partitioning for low power; see Section II).

We apply our method to a target architecture as described in Section III. Please note that, in general, we can have more than just one application specific core (Fig. 1 shows only one for simplification).

Our goal is to partition a system (in the following we will simply talk of an *application*) application between one or more μP cores and the application specific core(s) in order to minimize the total power consumption (note, though the partitioning algorithm is general, only one μP can be handled by the other parts of the framework).

A. The Basic Idea of Our Low-Power Partitioning Approach

During the execution of a program on a μP core different hardware resources within this core are invoked according to the instruction executed at a specific point in time. Assume, for example, an *add* instruction is executed that invokes the resource *ALU* and *Register*. A *multiply* instruction uses the resources *Multiplier* and *Register*. A *move* instruction might only use the resource *Register*, etc. Conversely, we can argue: during the execution of the *add* instruction the multiplier is not used; during execution of the *move* instruction neither the *ALU* nor the *Multiplier* is used, etc.[8] In case the processor does not feature the technique of gated clocks to shut down *all* nonused resources clock cycle per clock cycle,[9] those nonactively used resources will still consume energy since the according circuits continue to switch. We denote to this situation as "*the circuits are not actively used*." Accordingly, "*the circuits are actively used*" when the *are* invoked at that time by an instruction. For each resource rs_i of all resources RS within a core, we define a utilization rate

$$u_{rs} = \frac{N^{rs}_{act\_used}}{N_{total}} \tag{7}$$

where $N^{rs}_{act\_used}$ is the number of cycles resource rs is actively used and N_{total} is the number of all cycles it takes to execute the whole application. We define the "wasted energy" within a core i.e., the energy that is consumed by resources during times frames where those resources are not actively used as

$$E^{core}_{non\_act\_used} = \sum_{rs_i \in RS} (1 - u_{rs_i}) \cdot P^{rs_i}_{av} \cdot T_{app} \tag{8}$$

where $P^{rs_i}_{av}$ is the average power that is consumed by the particular resource and T_{app} is the execution time of the whole application when executed entirely by this core. Minimizing the total energy consumption can be achieved by minimizing $E^{core}_{non\_act\_used}$. Our solution is to deploy an additional core for that purpose, i.e., to partition the functionality that was formerly solely performed by the original core, to a new (to be specified) application specific core and in parts to run it on the initial core such that

$$\sum_{i=1}^{N_{core}} \left(E^{core^i}_{non\_act\_used} + E^{core^i}_{act\_used} \right) \leq E^{initial\_core}. \tag{9}$$

Whenever one of the cores $i, \ldots, N_{core}$ is performing, all the other cores are shut down (as far as they are not used, of course), thus consuming *no* energy. Equation (9) is most likely fulfilled when the individual resource utilization rate

$$U^{core}_R = \frac{1}{N_R} \cdot \sum_{rs \in RS} u_{rs_i} \tag{10}$$

of each core is as high as possible (*note*: in the ideal case it would be 1). There, N_R gives the number of all resources that are part of that core. We use the values U^{core}_R of all participating cores (i.e., those that are subject to partitioning) to determine whether a partition of an application is advantageous in terms of power consumption or not.

At this point one could argue that we better shut down the individual resources *within* each core rather than deploying additional cores to minimize energy. This is because we suppose that a state-of-the-art core based design techniques are used as described in the introduction. This implies that the designer's task is to compose a system of cores they can buy from a vendor rather than modifying a complex core like a μP core.

Hence, our methodology allows the use of core-based design techniques *and* minimizing energy consumption *without* mod-

The Low Power Partition Process

Build a graph $G = \{V, E\}$
$C = decompose\_into\_cluster(\{V, E\})$
For All cluster $c_i \in C$
calculate: $N^{c_i}_{Trans,\mu Pcore \leftrightarrow ASICcore}$
$C = pre - select(C, N^{c_i}_{max})$
For All $cluster\ c_i \in C$
For All $rs_i \in RS$
$do\_list\_schedule(c_i, rs_i)$
If $(U^{core}_R > U^{core}_{\mu P})$
Then
$E^{core}_R = U^{core}_R \cdot \sum_{rs_i \in RS}(P^{rs_i}_{av} \cdot N^{rs_i}_{cyc} \cdot T^{rs_i}_{cyc})$
$E^{core}_{\mu P} = E^{core}_{\mu P,initial} - E^{core}_{\mu P,c_i}$
$OF = F \cdot \frac{E^{core}_R + E^{core}_{\mu P} + E_{rest}}{E_0} + \ldots$
Synthesize a core with a min. OF value
Estimate energy (gate-level) and comp. energy savings

Fig. 7. Pseudocode of our partitioning algorithm.

[8] These are examples for demonstration purposes only. However, this might not apply in this simple form to a particular processor.

[9] This is actually the case for most today's processors deployed in embedded systems. An example is the LSI SPARCLite processor core.

ifying complex standard cores. Whereas the above described basic idea was formulated more general, the following implementation of our core/core partitioning algorithms[10] is based on hardware/software partitioning between one μP core and an application specific core (ASIC core).

B. The Low-Power Partitioning Process

This section gives an overview of our low power partitioning approach in coarse steps. It is based on the idea that an application specific hardware (we call it in the following ASIC core) can, under specific circumstances, achieve a higher utilization rate U_R^{core} than a standard (programmable) processor core (in the following we refer to it as μP core).

The input to the partitioning process is a behavioral description of an application that is subject to a core/core partition between the ASIC core and the μP core. The following descriptions refer to the pseudocode in Fig. 7.[11] Step 1 derives a graph $G = \{V, E\}$ from that description. There, V is the set of all nodes (representing operations) and E is the set of all edges connecting them. For more details of the graph representation, please see [32].

Using this graph representation, Step 2 performs a decomposition of G in so-called *cluster*. A cluster in our definition is a set of operations which represents code segments like nested loops, if–then–else constructs, functions, etc. Note, that a cluster can comprise one or more control structures that are nested or sequntial. For more detailed information, please refer to [32]. The decomposition algorithm is not described here because it is not key to our approach. Decomposition is done by structural information of the initial behavioral description solely. Whether the implementation of a cluster on an ASIC is leading to an actual net energy reduction also depends on whether an additional (high) communication traffic is implied by that or not. Though we do not explicitly take related power/energy consumption into consideration (only the trafic in terms of number of transfers), it is up to the system designer to decide whether clusters that imply high communication traffic should be considered for an ASIC implementation or not. The calculation is done in lines 3 and 4. Due to the importance, the separate Section V-C is dedicated to that issue. Line 5 performs a pre-selection of clusters, i.e., it preserves only those clusters for a possible partitioning that are expected to yield high-energy savings based on the bus traffic calculation. Here, the designer has a possibility of interaction by specifying different constraints like, for example, the total number of clusters $N_{\max}^c$ to be pre-selected. Please note that it is necessary to reduce the number of all clusters since the following Steps 6 to 12 are performed for *all* remaining clusters.

In line 7, a loop is started for all *sets of resources* where the set of different resource sets RS is specified by the designer. The designer tells the partitioning algorithm how much hardware (#ALUs, #multipliers, #shifters, ...) they are willing to spend for the implementation of an ASIC core. The different sets specified are based on reference designs, i.e., similar designs from past projects. Due to our design praxis, three to five sets are given, depending on the complexity of an application. Afterwards, in line 8, a simple list schedule is performed on the current cluster in order to prepare the following step. That step is one major part of the work presented here: the computation of U_R^{core} (line 9). There it is tested whether a candidate cluster can yield a better utilization rate on an ASIC core or on a μP core. Due to the complexity of calculating U_R^{core}, a detailed description is given in the separate Section V-D. In case a better utilization rate is possible, a rough estimation on expected energy savings is performed (lines 11 and 12). Note that the energy estimate of the ASIC core is based on the utilization rate. For each resource rs_i of the whole sets of resources RS (as discussed above), an average power consumption $P_{av}^{rs_i}$ is assumed.[12] $N_{cyc}^{rs_i}$ is the number of cycles resource rs_i is actively used whereas $T_{cyc}^{rs_i}$ gives the minimum cycle time the resource can run at. The energy consumed by the μP core is obtained by using our instruction set energy simulation tool (it will be explained in some more detail in Section VII). The objective function OF of the partitioning process is defined as a superposition of the normalized total energy consumption and additional hardware effort we have to spend. Please note that E_{rest} gives the energy consumption of all other cores (instruction cache, data cache, main memory, bus). E_0 is provided for the purpose of normalization only. Finally, F is a factor given by the designer to balance the objective function between energy consumption and possible other design constraints. F is heavily dependent on the design constraints as well as on the application itself. For the partition that yields the best value of the objective function, the steps in lines 14 and 15 are executed: the synthesis and the following gate-level energy estimation. These two steps are described during introduction of the whole design flow in Section VII.

As already mentioned, the following two sections Sections V-C and V-D are dedicated to a closer description of the pre-selection criteria for a cluster and U_R^{core}, respectively.

C. Determining the Pre-Selection Criteria of a Cluster

The pre-selection algorithm of clusters is based on an estimation for communication traffic increase for a cluster implemented as an ASIC. When a hardware/software partition of an application between a μP core and an ASIC core is deployed, the following additional bus traffic—based on the architecture shown in Fig. 8(a) where two cores communicate via a shared memory—is implied.

a) When the μP core arrives at a point, where it "calls" the ASIC core, then it is depositing data or references to that data in the memory such that it can be accessed by the ASIC core for subsequent use.

b) Once the ASIC core starts its operation it will access, i.e., download the data or references to it from the memory.

[10]Please note that we sometimes use the term *core/core partitioning* and sometimes the term *hardware/software partitioning*. Through our definition, both terms have the same meaning. But according to the specific context the one or the other term is actually used.

[11]Please note that we do not use "{"and"}" to indicate the scope of validity of an *If* statement or a loop. Rather than that we indicate it by aligning to columns accordingly.

[12]The according data is derived by means of the CMOS6 library that is used later on for gate-level energy calculation as well.

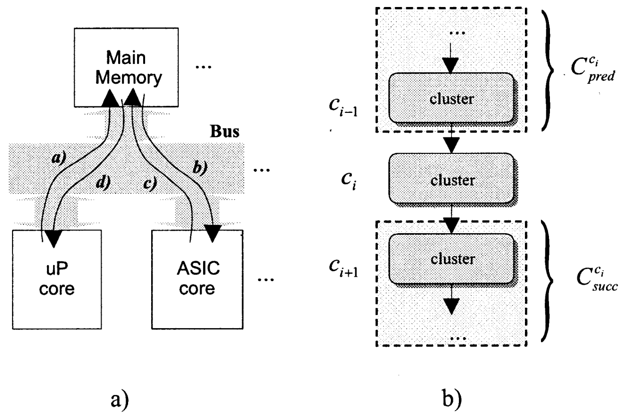


Fig. 8. (a) Bus transfer scheme. (b) Nomenclature of algorithm to estimate those transfers. A cluster can either be mapped to "μP *Core*" or to "*ASIC Core*".

c) After the ASIC core has finished its job, some data might be used by the μP core to continue execution. Therefore, the ASIC core is depositing the according data or references to it in the main memory.

d) Finally, the μP core reads data back from the memory.

The amount of transfers described in b) and c) occur in any case, no matter whether there is a μP core/ASIC core partitioning or not. Hence, we do not account for those in the following algorithm that is supposed to be the calculation of an *additional* (i.e., due to partitioning only) energy effort that would have to be spent.

The algorithm is based on the conventions shown in Fig. 8(b). There, each node represents a cluster. The arcs are indicating the direction of the control flow. The current cluster is denoted as c_i, whereas the previous one is drawn as c_{i-1} and the succeeding one is given as c_{i+1}. Furthermore, we define $C^{c_i}_{pred}$ to represent *all* clusters preceding c_i Similarly, $C^{c_i}_{succ}$ combines *all* clusters succeeding c_i. Step 1 computes the number of all transfers from the μP core to the memory. Apparently, only data has to be transferred that is *generated*[13] in all clusters preceding the current one *and* that is *used* in the current one (i.e., that one that is supposed to be implemented on the ASIC core). Step 2 tests whether the preceding cluster might probably be already part of the ASIC core such that the estimation can take that into account accordingly. The estimation of communication effort for the ASIC core (Steps 3 and 4) follows the same principle as described Steps 1 and 2.

D. Determining the Utilization Rate

Now, since a scheduling has been performed, we can compute the resource utilization rate U^{core}_R of a core. The following definitions hold: CS is the set of all control steps (result of the list schedule) and cs_i is the denotation of one individual control step within CS. Furthermore, O_c is the set of all operations within a cluster c whereas $o_{i,c}$ is an operation within O_c that is scheduled into control step i. An operation can be be mapped to one of the D resource types in $RS = \{rs_1, \ldots, rs_D\}$.[14] Please note that each type π of a resource rs—or short, rs_π—can have several instances. With these definitions we can discuss the algorithm in Fig. 10 that is given in pseudocode.

[13]We use gen $[\cdots]$ and use $[\cdots]$ as it is defined in [23].

[14]Examples for a resource type are an *ALU*, a *shifter*, a *multiplier*, etc.

Computing the energy of additional bus transfers

1) Number of bus transfers between μP core and memory
$N^{c_i}_{Trans,\mu Pcore \to mem} = |gen\left[C^{c_i}_{pred}\right] \cap use\left[c_i\right]|$

2) Take into consideration synergetic effects:
If $(implemented\_in\_ASIC\_core(c_{i-1})))$
Then

$$N^{c_i}_{Trans,\mu Pcore \to mem} = T^{c_i}_{Trans,\mu Pcore \to mem} - |gen\left[c_{i-1}\right] \cap use\left[c_i\right]|$$

3) Number of bus transfers betw. ASIC core and memory
$N^{c_i}_{Trans,ASICcore \to mem} = |gen\left[c_i\right] \cap use\left[C^{c_i}_{succ}\right]|$

4) Take into consideration synergetic effects:
If $(implemented\_in\_ASIC\_core(c_{i+1})$
Then

$$N_{Trans,ASICcore \to mem} = N_{Trans,ASICcore \to mem} - |gen\left[c_i\right] \cap use\left[c_{i+1}\right]|$$

5) Total energy:

$$E^{c_i}_{Trans,\mu Pcore \leftrightarrow ASICcore} = (N^{c_i}_{Trans,\mu Pcore \to mem} + N^{c_i}_{Trans,ASICcore \to mem}) \times E_{bus\ read/write}$$

Fig. 9. Pseudocode of the algorithm to calculate energy of bus transfer in order to determine the pre-selection of cluster.

Computing U^{core}_R and GEQ_{RS}

1) Initialize *Glob_RS_List*[][][]
2) **For All** $cs_i \in CS$
3) Initialize *Loc_RS_List*[][]
4) **For All** $o_{i,c} \in O_c$
5) Build up *Sorted_RS_List*[]
6) $rs_\pi :=$ *Sorted_RS*_List[0]
7) **For All** *elements* $\in$ *Sorted_RS_List*[]
8) $rs_\pi :=$ current resource type of *Sorted_RS_List*[]
9) **If** $\#(rs_\pi)_{rs_\pi \in \text{Glob\_RS\_List}[cs_i][][]} >$
10) $\#(rs_\pi)_{rs_\pi \in \text{Loc\_RS\_List}[][]}$
11) **Then**
12) Update $\#(rs_\pi)$ in *Loc_RS_List*$[rs_\pi]$[]
13) continue with 4)
14) Update $\#(rs_\pi)$ in *Loc_RS_List*$[rs_\pi]$[]
15) Update *Glob_RS_List*$[cs_i]$[][] with *Loc_RS_List*[][]
16) Initialize GEQ_{RS}
17) **For All** $rs_\pi \in$ *Glob_RS_List*[][][]
18) $GEQ_{RS} := GEQ_{RS} + \#(rs_\pi) \times GEQ(rs_\pi)$
19) **For All** $cs_i \in CS$
20) **For All** $rs_i \in RS$
21) **For All** instances is
22) **If** $(Glob\_RS\_List[cs_i][rs_i][is] == 1)$
23) **Then** $(util[rs_i][is] = util[rs_i][is] + \#ex\_cycs)$
24) $U^{core}_R = \frac{1}{N^c_{cyc}} \cdot \sum_{rs_i \in RS}(\frac{1}{N_{is}} \cdot \sum_{is}) u_r[rs_i][is]$

Fig. 10. Pseudocode of our algorithm to compute the utilization rate U^{core}_R and the hardware effort GEQ_{RS} of a cluster.

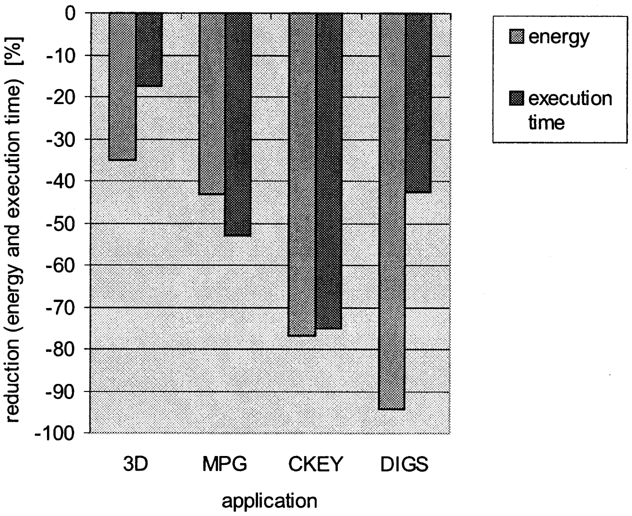


Fig. 11. Energy and execution time improvements (i.e., reductions) as a result of hardware/software partitioning.

```
test1(...)                        test2()
{ ... ...                         { ... ... }
test2(); /*call 3*/
 ... ... }
main()
{ int i, j;
 ... ...
 for (i=0; i<100; i++) { /* loop 1 */
 test1(...); /* A */
 for(j=0; j<100; j++) /* loop 2 */
 test1(...); /* B */
 }
 ... ... }
```

Fig. 12. Program excerpt example for software transformations.

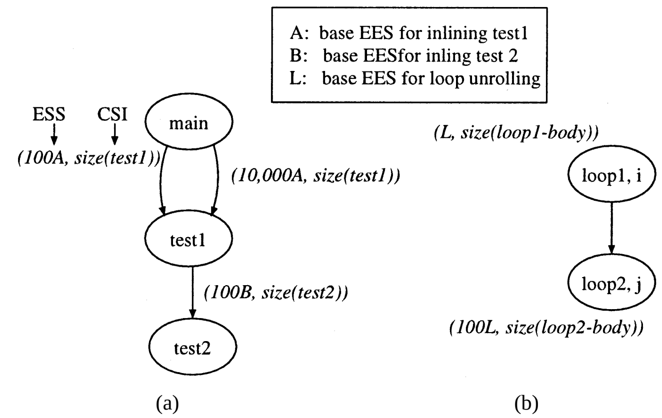


Fig. 13. (a) Procedure calling graph and (b) loop graph of the program example of Fig. 12.

```
Inputs: source_software, design_goal;
Variable: solution_pool, current_sw, new_sw,
current_design, new_design, tmp_design;
1. Static analysis of program:
2.   identify all possible transformations;
3.   construct procedure graphs / loop graphs;
4.   generate possible cache / memory sizes;
5. Energy optimization:
6.   for each memory size m_size {
7.     current_sw = source_sw;
8.     current_design = best design with current_sw
from solution_pool;
9.     do{
10.      new_sw = transformation_select(current_sw,
m_size);
11.      new_design = (new_sw,0,0,m_size);
12.      A=set of i_cache/d_cache sizes for new_sw;
13.      for each (dcache, icache) in A {
14.         tmp_design = (new_sw,d_cache,
i_cache,m_size);
15.         new_design = choose one by
design_goal(tmp_design, new_design);
16.      }
17.      if new_design better than current_design{
18.         save new_design in solution_pool;
19.         current_sw = new_sw; current_design=
new_design; }
20.      else continue;
21.    } while( !stop_condition)
22.  }
23.  output: designs from solution_pool that
satisfies design_goal.
```

Fig. 14. System-level energy optimization algorithm.

At the beginning a *global resource list* Glob_RS_List [][][] is defined. The first index indicates the control step cs_i, the second stands for the resource type rs_π, while the third is reserved for a specific instance of that resource type. An entry can be either a "1" or a "0". For example, Glob_RS_List $[34][5][2] = 1$ means that during control step 34 instance "2" of resource type "5" is used. Accordingly, "0" means that is not used. The encoding of the existence of a module type is accomplished by providing or not providing an entry in Glob_RS_List [][][].[15] Line 2 starts a loop for all control steps cs_i and in line 3 a local resource list Loc_RS_List [][] is initialized. It has the same structure as the global resource list except that it is used within one control step only. Line 4 starts a loop for all operators within a control step. A sorted resource list is defined in line 5. It contains all resources that could execute operator $o_{i,c}$. It is sorted according to the increasing size of a resource.[16] An initial resource is selected in line 6. In the following lines 9 to 13, all possible resource types are tested whether they are instantiated in a previous control step. If this is true, that resource type is assigned to the current operator, an according entry is made in the local resource list and a new operator is chosen. In the other case, the searching process through the local resource list continues until an already instantiated instance is found that is not used during the current control step. In case the search did not succeed, the first[17] resource is assigned to the current operator and an according entry is made (line14). When all operators within a control step have been taken care of, the global resource list is enhanced by that many instances of a resource as indicated by the local resource list (line 15).

As a result, the global resource list contains the assignment of all operators to resources for all control steps. We can use this information to compute the according hardware effort GEQ_R^{core} in lines 16 to 18 where $\#(rs_\pi)$ gives the number of resources of type π and $GEQ(rs_\pi)$ is the hardware effort (i.e., *gate equivalents*) of an according resource type.

[15]This is possible since the implementation of Glob_RS_List [][][] is a chained list.

[16]This is for the computation of the hardware effort of the final core only.

[17]Note that the list is sorted. So, the first resource means the largest in terms of utilization rate and therefore the most energy efficient one.

TABLE I
ENERGY CONSUMPTION AND EXECUTION TIME FOR BOTH, INITIAL (I) AND PARTITIONED (P) DESIGN

App.		Energy							Exec. Time [cycles]			
		i-cache	d-cache	mem	μP core	ASIC core	total	Sav%	μP core	ASIC core	total	Chg%
3d	I	116.93uJ	14.26uJ	29.71uJ	566.78uJ	n/a	727.68μJ	-35.21	39,712	n/a	39,712	-17.29
	P	0.2627μJ	0.1227μJ	0.2626μJ	0.4705mJ	0.3078μJ	471.46μJ		32,689	154	32,843	
MPG	I	44.79mJ	17.98mJ	2.305mJ	74.32mJ	n/a	140.92mJ	-43.20	5,167,958	n/a	5,167,958	-52.90
	P	35.36mJ	13,14mJ	2.941mJ	27.14mJ	1.462mJ	80.04mJ		1,696,771	737,154	2,433,925	
ckey	I	0.0	0.0	0.0	329.99mJ	n/a	329.99mJ	-76.81	169,511,665	n/a	169,511,665	-74.98
	P	0.0	0.0	0.0	53.042mJ	23.468mJ	76.51mJ		30,258,256	12,144,420	42,402,676	
digs	I	11.69mJ	5.123mJ	0.493mJ	52.70mJ	n/a	70.00mJ	-94.12	3,706,291	n/a	3,706,291	-42.64
	P	14.27μJ	2.039μJ	20.43μJ	46,78μJ	4.03mJ	4.11mJ		6,347	2,119,750	2,126,097	

The final computation of the *utilization rate* is performed in line 24. Before, in lines 19 to 23 a list is created that gives information about how often each instance of each resource is used within all control steps. Note that $\#ex\_cycs \cdot \#ex\_times$ is the number of cycles it takes to execute an operation on that resource multiplied by the number of times the according control step is actually invoked.[18] Finally, we can compute U_R^{core} in line 24. Please note that N_{cyc}^c is the number of cycles it takes to execute the whole cluster.

As a summary, in this section we have computed U_R^{core} that gives the average utilization rate of all resources deployed within a candidate core. As we have seen in Section V-B, U_R^{core} is actually used to determine whether this might lead to an advantageous implementation of a core in terms of energy consumption or not.

Also note that all resources contribute to U_R^{core} in the same way, no matter whether they are large or small (i.e., though they may actually consume more or less energy). This is because our experiments have shown that an according distinction does not result in better partitions though the individual values of U_R^{core} *are* different. Reason is that the *relative* values of U_R^{core} of different clusters are actually responsible for deciding on an energy efficient core/core partition.

E. Partitioning Results

Energy and data values are obtained using the estimation-related design flow of Avalanche as seen in Fig. 3. A 0.8 μ CMOS process is assumed for all system parts. We investigated the following DSP-oriented applications: an algorithm for computing 3-D vectors of a motion picture ("*3-D*"), an MPEGII encoder ("*MPG*"), a complex chroma-key algorithm ("*ckey*") and a smoothing algorithm for digital images ("*digs*"). The size of the applications range from about 5 to 230 kB of C code. Two rows are dedicated to each application: the initial (nonpartitioned) "I" implementation and the partitioned "P" implementation. In each case, the contribution of each involved core in terms of energy consumption is given. It is an important feature of our approach that *all* system components are taken into consideration to estimate energy savings. This is because a differently partitioned system might have different access patterns to caches and main memory, thus resulting in different energy consumptions of those cores (compare according rows of columns "*i-cache*," "*d-cache*," and "*mem*"). The sole energy estimation of the μP core and the ASIC core would not be sufficient since the energy consumption of the other cores in some cases drops dramatically as well. In one case ("*ckey*") which was in fact the less memory-intensive one, the contribution to total energy consumption could be neglected.

The rightmost four columns give the execution time before and after the partitioning. This is of paramount importance: we achieved high energy savings but *not* at the cost of performance (except for one case). Instead, energy savings are achieved at additional hardware costs for the ASIC core through our selective algorithms described in Section V. The largest (but still small) additional hardware effort accounted for slightly less than 16k cells. But in that case ("*digs*") a large energy saving of about 94% could be achieved. Due to todays design constraints in embedded high-performance applications, a loss in performance through energy savings is in the majority of cases not accepted by designers. On the other side a (low) additional hardware effort of 16k cells is not a real constraint since state-of-the-art systems on a chip have about 10 Mio transistors.[19]

We achieved high energy savings between about 35% and 94% while the decrease in execution time (i.e., faster) ranges between about 17% and 75%. It shows that our approach is especially tailored for energy minimization and improvement of execution time is only a side effect. Fig. 11 visually summarizes the results as a percentage improvement for energy and execution time.

VI. POWER OPTIMIZATION THROUGH SYSTEM-LEVEL PARAMETER OPTIMIZATION AND SOURCE-TO-SOURCE TRANSFORMATIONS

This section describes the usage of Avalanche in a design scenario where hardware/software partition has already been determined either by the method described in the previous section or, alternatively, partitioning has been determined through other design constraints.

Source-to-source transformations change the software by various high-level techniques as discussed below. System parameter optimization changes various cache and main memory parameters (see Section III for parameters) and finds a power efficient solution. Please note, in case one component (software, cache or memory) or one component's parameters are changed, it not only affects the energy consumption of that particular system component, but also that of other components in the

[18]We obtain $\#ex\_times$ through profiling and $\#ex\_cycs$ through the CMOS6 technology library.

[19]Please note that due to current state-of-the-art technology of 0.18 μ an even higher transistor count would be possible. But due to the current "*design gap*" (therefore see also [1], a maximum is currently about 10 Mio. transistors on a chip (not including main memory).

system; it not only affects the power, but also the performance. The interesting aspect is that the change of overall system energy and performance cannot be easily predicted unless comprehensive system analysis is performed. We can summarize that system parameters are interdependent. We continue to discuss some scenarios of software and cache/memory changes and their possible impacts on energy and performance.

a) **Source-to-Source Transformations:** In case a transformation can be performed on the software to lower the *software energy*,[20] this transformation may also change the cache/main memory access pattern and result in ambiguous changes of the caches or main memory energy and the performance. In some cases, source-to-source transformations may increase the code size so that a larger main memory is required to accommodate the new code; therefore, the energy consumption of each memory access increases.

b) **I-Cache and D-Cache:** When a larger instruction and/or data cache is used, in general, there are less cache misses and the *system performance* is improved. The *software energy* decreases because less cache misses imply less main memory access penalties. The energy of the main memory is decreasing because of less accesses. However, the energy consumed by the caches increases due to its increased size and the system energy change is ambiguous.

c) **Main Memory:** When a larger main memory is used, the energy consumption of the main memory increases because of its larger size (4), but the energy of other parts is usually not affected.

A. Source-to-Source Transformation and Their Effect on Energy Consumption

Many source-level transformations have been proposed for the purpose of improving performance. However, they may have some side effects other than performance improvement, such as a bigger main memory requirement due to an increased code size. This will lead to larger energy consumption due to larger capacitances to switch for each access. Here we give a brief look at some commonly used transformation techniques and analyze their impacts on energy and performance.

Procedure calls are costly in most architectures. *Procedure in-lining* can help improve performance and save software energy by eliminating the overhead associated with calls and returns. For example, suppose we have a SPARC architecture that features up to eight register windows. For each new procedure call a new window is required and released after the return from the procedure. However, if the depth of procedure calls (i.e., a consecutive number of calls without returns) exceeds the available number of register windows, an interrupt is released for the operating system to process the spilling of register contents to the main memory. This is time consuming. A side effect of in-lining is the increased code size, especially when the procedure is called from different points within the program.

Loop unrolling is another transformation technique. It can help to increase the instruction level parallelism and eliminate control overhead. Similar to procedure in-lining, it also results in code size increase. Another possible impact is that an unrolled loop may no longer fit in the instruction cache so that it possibly will be slowed down. Other techniques include *software pipelining, recursion elimination, loop optimization, etc.* [10], whose impacts on both the software and cache/memory accesses may make it hard to judge the change of the overall system energy consumption.

[20]For example, through methods like proposed in [25] or [10]

TABLE II
OPTIMIZATION RESULTS COMPARED TO ORIGINAL ARCHITECTURE

App.	Objective	ref. arch.	set of pareto opt. solutions		
bsort	energy [J]	0.33E-3	0.27	0.24	0.21
	e-improv. [%]	n/a	-17.6	-26.8	-36.1
	time [# cyc$\times 10^6$]	19.4	15.9	14.1	12.3
	t-improv. [%]	n/a	-18.4	-27.5	-36.8
eg2	energy [J]	0.31	0.25	0.23	-
	e-improv. [%]	n/a	-18.2	-24.8	-
	time [# cyc$\times 10^6$]	18.0	14.8	13.6	-
	t-improv. [%]	n/a	-17.6	-24.3	-
smooth	energy [J]	1.03	0.10	0.72	0.48
	e-improv. [%]	n/a	-90.1	-29.8	-53.5
	time [# cyc$\times 10^6$]	60.1	3.6	42.1	28.0
	t-improv. [%]	n/a	-94.0	-30.0	-53.5
itimp	energy [J]	2.97	2.4	2.2	-
	e-improv. [%]	n/a	-19.9	-25.3	-
	time [# cyc$\times 10^6$]	173.8	139.1	129.5	-
	t-improv. [%]	n/a	-20.0	-25.5	-

B. Our Low-Power Source-to-Source Transformation Algorithm

When a designer is concerned about both performance and power, a sophisticated approach is mandatory to choose which transformations to perform and in what order. In order to find the combination and sequences of transformations that yield the most energy savings under memory size constraints, we designed a *transformation-selection* algorithm. Given a set of available transformations techniques, the algorithm needs to

1) identify *which* transformations can be applied and *where* and evaluate these choices of transformations;
2) choose the combination and the order of the transformations that obtain the best energy improvement without violating a memory size limit.

Currently, we have implemented procedure in-lining and loop unrolling using SUIF [24]. However, our *transformation-selection* algorithm is applicable to general types of transformations as long as their particular characteristics are defined (see later). The algorithm is independent of the transformations themselves.

As a first step, we developed heuristic measures to characterize the estimated-energy-saving (EES) and code size increase (CSI) incurred by these transformations. EES is the estimated energy improvement while performing a certain transformation. It can either be a constant, or a function of some parameters depending on the type of the actual transformation. Fig. 12 shows a code segment that contains two loops and three procedure calls. The EES for in-lining the procedure *test1* at location *A* is 100 times the base EES of in-lining *test1*. At location *B*, EES is 10 000 times the base EES. Similar considerations apply to loop unrolling.

In order to identify the calling relationships, a *procedure calling graph* is constructed (Fig. 13) for each program. In the

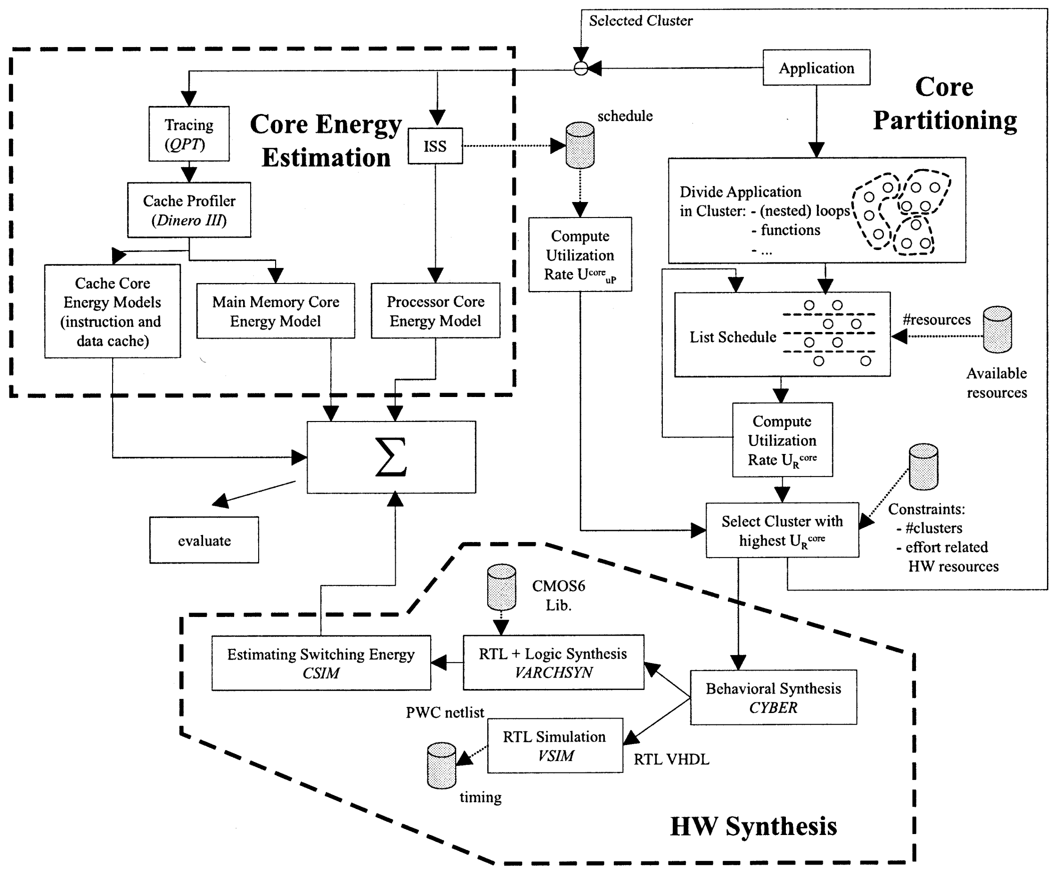


Fig. 15. Design flow of our low-power hardware/software partitioning methodology.

calling graph, a node represents a procedure and a directed edge represents a procedure call. Multiple edges between nodes may exist, reflecting that a procedure can be called from different locations. The edges have been assigned the attributes *EES* and *CSI*. Since our algorithm does not support recursion, the procedure calling graph is *acyclic*. After in-lining has been applied, the edge corresponding to the call is removed. For loop unrolling, a similar graph is created in which a node represents a loop, an edge indicates that one loop is nested within another. However, unlike the procedure calling graph, the nodes are labeled instead of the edges because the nodes are where the transformations are applied to.

Note that the various transformations are not independent of each other. Let us consider the example in Fig. 12: if *loop 1* is unrolled, 100 new instances of *test1* calls will be generated and new calling edges in the procedure calling graph need to be added. It is important for the algorithm to not only choose the best combination of the transformations, but also the right order.

In the second step, the algorithm

1) prioritizes all possible transformations according to a heuristic measure—the *EES/CSI* ratio;
2) probability is assigned to each transformation according to its priority value;
3) in each transformation step, randomly select a transformation based on the probabilities, perform the transformation, and update the procedure and loop graphs;
4) repeat 3) until the memory limit is reached.

This algorithm is called repeatedly by the system-level energy optimization algorithm (Section VI-C).

C. System-Level Energy Optimization Algorithm

Let us now define the problem of the optimization algorithm. At this point we assume that

- hardware/software partitioning has already been applied and application specific hardware is synthesized and, therefore, fixed;
- processor has been chosen;
- we are given an initial version of the software.

The algorithm is designed for minimizing energy. However, as power is usually not the sole concern in the design process, a multiple objective function is the choice.

The goal is to find a set of solutions within performance *and* energy constraints. This will provide important tradeoff information to the designer. The designer can review different design options and choose the most suitable one.

The algorithm returns the optimized new system configuration of the target system architecture: the transformed program, the data cache and instruction cache sizes, etc., and the main memory size. For our goal, a set of designs is returned, with percentage data indicating energy and performance difference between two designs adjacent in terms of energy consumption.

Fig. 14 shows the pseudocode for the optimization algorithm. It consists of two main steps.

1) **Static analysis of the application program** (lines 1–4), includes
   a) generating the procedure calling graph and loop graph, as described in Section VI-B;
   b) generating the set of feasible cache and memory sizes and configurations based on the current version of the program.
2) **Optimization step** (lines 6–23): choose the design, i.e., set of transformations and the cache and memory parameters to meet the constraints and optimization goal.

In the algorithm, we limit the maximum memory size to four times of the original memory size (of the original, not transformed software program) because as shown by our experiments, the energy overhead of a very large memory usually outweighs the energy saving provided by software transformations. We generate a set of designs for each possible memory size (lines 6–22) and select design(s) that meet the design goal in line 23. A design is represented as a quadruple of software, instruction cache, data cache, and main memory.

To construct designs for a certain memory size, we perform software transformations using the algorithm described in Section VI-B (line 10) and then decide the subset of feasible instruction/data caches for the transformed software (lines 11–12). The best instruction/data caches are chosen based on the designer's goal (line 13–16). The transformed software, the best suited cache sizes and parameters and the new memory size makes up a new design. If the new design has a better quality than the previous one, then it is saved in a solution pool (line 17–19) and will be used in the next iteration. Otherwise, the transformation is discarded (line 20) and a new transformation is performed on the previous version of the software. The process is repeated until a stop criteria is met (line 21): there is no improvement in a given number of consecutive iterations, or the total number of iterations reaches a preset limit.

An important issue in the algorithm is evaluating the quality of two designs. For our design goal for designs falling within energy and performance constraints, we use the *Pareto optimality* measure to discard solutions that are both higher in energy and slower in performance. Note, the energy and performance data are obtained using the models described in Section III.

D. Power and Performance Results

Table II shows the results yielded with our algorithm for multiple objective optimization. Shown are the application programs *bsort*, *eg2*, *ismooth* and *itimp* that are 2 KB, 12 KB, 2 KB, and 16 KB in size, respectively. The designer is provided with a set of different solutions from which he can choose.

The computation time for determining *one* design point (fixed system parameters) is in the range of 3–5 minutes. A whole optimization run is between 2 and 10 h on an UltraSparc 2.

Table II shows the results for our design goal compared to a reference architecture called "*ref. arch*" (a small standard cache; same size for i-cache and d-cache and not adapted to the system). For all applications, system energy consumption (Joule) and execution time (number of clock cycles) is given for the reference case. For the objective the relative improvement

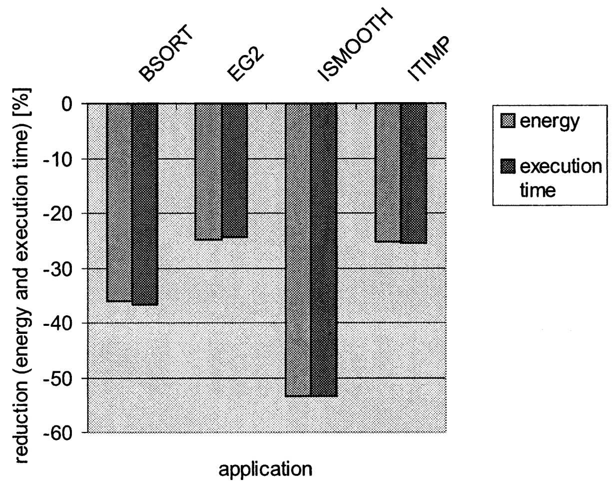


Fig. 16. Energy and execution time improvements (i.e., reductions) as a result of system parameter optimization and source code transformations.

$(value - value_{ref})/value_{ref}{}^*100$ is given. Apparently, a negative percentage number means an improvement.

The results comprise both improvement through system parameter adaptation as well as source-to-source transformations. But please note that in most cases of energy and performance improvement, the contribution of the source-to-source transformations is about 2% to 10%. The largest improvement has always been achieved by optimally adapting system parameters to each other. Fig. 16 visualizes the results in terms of improvements in energy consumption and execution time reduction.

VII. Summary of the Avalanche Design Flow and Conclusion

The whole design flow of our Avalanche is shown in simplified form in Fig. 15. The dashed upper left part is the estimation part as introduced in more detail in Section III. The dashed lower part is our standard in-house synthesis flow. All other parts refer to optimization and integration.

The partitioning design flow starts with the box "Application" where an application in a behavioral description is given. This might be a self-coded application or an IP core purchased from a vendor. Then the application is divided into clusters as described Section V-B after an internal graph representation has been build up. Preferred clusters are pre-selected by the criteria that is described in Section V-C. The next step is a list schedule that is performed for each remaining cluster.[21] such that the utilization rate U_{core}^{R} using the algorithm in Section V-D can be computed. Those cluster(s) that yield a higher utilization rate compared to the implementation of a μP core *and* that yield the highest core of the objective function, are provided to the hardware synthesis flow. This block starts with a behavioral compilation tool, followed by an RTL simulator[22] to retrieve the number of cycles it needs to execute the cluster, an RTL logic synthesis

[21]Please note that the flow in Fig. 15 is simplified, i.e., it does not feature all arcs representing the loops in the according algorithms.

[22]In order to keep the Fig. 15 of the design flow as clear as possible we did not draw the inputs of input stimuli pattern at various points in the design flow.

tool using a CMOS6 library and finally the gate-level simulation tool with attached switching energy calculation. Note that these steps, especially the last one, are the most time-consuming ones (all given times refer do not include the syntheses and gate-level/RTL-level simulation times). Hence, our partitioning algorithm has to reduce the number of clusters to those that are most likely to gain an energy saving.

The other application parts that are intended to run on the μP are fed into the "Core Energy Estimation" block. An instruction set simulator tool (ISS) is used in the next step. Attached to the ISS is the facility to calculate the energy consumption depending on the instruction executed at a point in time (the same methodology as in [15] is used). Analytical models for main memory energy consumption and caches are fed with the output of a cache profiler that itself is preceded by a trace tool (both [26]).

Finally, the total energy consumption is calculated and it is tested whether the total system energy consumption could be reduced or not. If "not" then the whole procedure can be repeated and the designer will make use of his/her interaction possibilities to provide the partitioning algorithms with different parameters. Please note that the designer does have manifold possibilities of interaction like defining several sets of resources, defining constraints like the total number of clusters to be selected or to modify the objective function according to the characteristics of an application. The major limitation of aour approach that the estimation is trace-based. In case that large traces are necessary, this can lead to high computation times. However, in cases like the MPEG encoder we have limited the traces to about six frames and achieved acceptable computation times.

In this paper, we have presented the Avalanche prototyping system for low-power embedded system design. We have seen that Avalanche can be used for various design scenarious like power/performance design space exploration, power/performance estimation, as well as for power optimization purposes. Those steps can either be conducted independently or they can all be applied to one design if that is in compliance with the designer's goal (as mentioned earlier, partitioning, for example, might have already been fixed due to other design constraints or can still be subject to optimization strategies). As for the results, we could achieve the highest energy savings when applying hardware/software partitioning. This is also the most costly optimization methods since it requires additional hardware. On the other side, adapting system parameters like cache policies, for example, or applying source-to-source transformations requires less effort in both design time and additional resource requirements. As a consequence, it turned out that the savings in energy and execution time where less dramatical.

Avalanche has been used to evaluate some real world designs among those a settop box design.

As one topic of continuing work on Avalanche we are currently studying the power consumption of the interconnects (e.g., buses) that connect the cores of an SOC.

References

[1] M. Keaton and P. Bricaud, *Reuse Methodology Manual for System-on-a-Chip Designs*. Norwell, MA: Kluwer, 1998.

[2] (1998) TI's 0.07 Micron CMOS Technology Ushers in Era of Gigahertz DSP and Analog Performance. Texas Instruments. [Online]. Available: http://www.ti.com/sc/docs/news/1998/98 079.htm

[3] R. K. Gupta and Y. Zorian, "Introducing core-based system design," *IEEE Design Test Comput. Mag.*, vol. 13, no. 4, pp. 15–25, 1997.

[4] B. P. Dave, G. Lakshminarayana, and N. K. Jha, "COSYN: Hardware-software co-synthesis of embedded systems'," in *Proc. DAC'97*, 1997, pp. 703–708.

[5] T. Givargis, F. Vahid, and J. Henkel, "A hybrid approach for core-based system-level power modeling," in *Proc. ASP-DAC'99*, 1999, pp. 141–145.

[6] T. Simunic, L. Benini, and G. De Micheli, "Cycle-accurate simulation of energy consumption in embedded systems," in *Proc. DAC'99*, 1999, pp. 867–872.

[7] M. Lajolo, A. Raghunathan, S. Dey, and L. Lavagno, "Efficient power co-estimation techniques for system-on-chip design," in *Proc. DATE'00*, 2000, pp. 27–34.

[8] Y. Shin and K. Choi, "Power conscious fixed priority scheduling for hard real-time systems," in *Proc. DAC'99*, 1999, pp. 134–139.

[9] L. Benini, A. Macci, E. Macii, M. Poncino, and R. Scarsi, "Synthesis of low-overhead interfaces for power-efficient communication over wide buses," in *Proc. DAC'99*, 1999, pp. 128–133.

[10] M. Kandemir, N. Vijaykrishnan, M. J. Irwin, and W. Ye, "Influence of compiler optimizations on system power," in *Proc. DAC'99*, 2000, pp. 304–307.

[11] R. Dick, G. Lakshminarayana, A. Raghunathan, and N. Jha, "Power analysis of embedded operating systems," in *Proc. DAC'99*, 2000, pp. 312–315.

[12] L. Benini, A. Macci, E. Macii, and M. Poncino, "Selective instruction compression for memory energy reduction in embedded systems," in *Proc. ISLPED'99*, 1999, pp. 206–211.

[13] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design," in *Proc. DAC'00*, 2000, pp. 294–299.

[14] W. Fornaciari, D. Sciuto, and C. Silvano, "Power estimation for architectural exploration of HW/SW communication on system-level buses," in *Proc. Codes'99*, pp. 152–161.

[15] V. Tiwari, S. Malik, and A. Wolfe, "Instruction level power analysis and optimization of software," *J. VLSI Signal Processing*, pp. 1–18, 1996.

[16] P.-W. Ong and R.-H. Ynn, "Power-conscious software design—A framework for modeling software on hardware," in *IEEE Proc. Symp. Low Power Electronics*, 1994, pp. 36–37.

[17] T. Sato, M. Nagamatsu, and H. Tago, "Power and performance simulator: ESP and its application for 100 MIPS/W class RISC design," in *IEEE Proc. Symp. Low Power Electronics*, 1994, pp. 46–47.

[18] R. Gonzales and M. Horowitz, "Energy dissipation in general purpose processors," in *IEEE Proc. Symp. Low Power Electronics*, 1995, pp. 12–13.

[19] M. B. Kamble and K. Ghose, "Analytical energy dissipation models for low power caches," in *IEEE Proc. Symp. Low Power Electronics and Design*, 1997, pp. 143–148.

[20] P. R. Panda, N. D. Dutt, and A. Nicolau, "Architectural exploration and optimization of local memory in embedded systems," in *Proc. IEEE Int. Symp. System Synthesis*, 1997, pp. 90–97.

[21] I. Hong, D. Kirovski, and M. Potkonjak, "Potential-driven statistical ordering of transformations," in *Proc. DAC'97*, 1997, pp. 347–352.

[22] S. J. E. Wilton and N. P. Jouppi, "An Enhanced Access and Cycle Time Model for On-Chip Caches," DEC, WRL Res. Rep. 93/5, 1994.

[23] A. W. Aho, R. Sethi, and J. D. Ullmann, *COMPILERS Principles, Techniques, and Tools*: Bell Telephone Lab., 1987.

[24] "The SUIF Library: A Set of Core Routines for Manipulating SUIF Data Structures," Stanford Compiler Group, Stanford Univ., Stanford, CA, 1994.

[25] V. Tiwari, "Logic and System Design for Low Power Consumption," Ph.D., Princeton Univ., Princeton, NJ, 1996.

[26] M. D. Hill, J. R. Laurus, and A. R. Lebeck *et al.*, *WARTS: Wisconsin Architectural Research Tool Set*, Wisconsin: Computer Sci. Dept. Univ. Wisconsin.

[27] K. Itoh, K. Sasaki, and Y. Nakagome, "Trends in low-power RAM circuit technologies," *Proc. IEEE*, vol. 83, pp. 524–543, Apr. 1995.

[28] W. Ye, R. Ernst, Th. Benner, and J. Henkel, "Fast timing analysis for hardware-software co-synthesis," in *Proc. ICCD*, 1993, pp. 452–457.

[29] I. Hong and D. Kirovski *et al.*, "Power optimization of variable voltage core-based systems," in *IEEE Proc. 35th Design Automation Conf. (DAC98)*, 1998, pp. 176–181.

[30] Ch. Ta Hsieh, M. Pedram, G. Mehta, and F. Rastgar, "Profile-driven program synthesis for evaluation of system power dissipation," in *IEEE Proc. 34th Design Automation Conf. (DAC97)*, 1997, pp. 576–581.

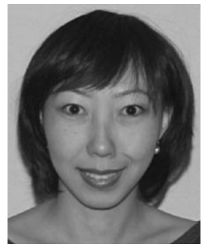
[31] P. Landman and J. Rabaey, “Architectural power analysis: The dual bit type method,” *IEEE Trans. VLSI Syst.*, vol. 3, pp. 173–187, June 1995.

[32] J. Henkel and R. Ernst, “An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques,” *IEEE Trans. VLSI Syst.*, vol. 9, pp. 271–289, Apr. 2001.

Jörg Henkel (M’95–SM’01) received the M.Sc. and Ph.D. degrees in electrical engineering both from the Technical University of Braunschweig, Braunschweig, Germany, in 1991 and 1996, respectively.

Since 1997, he has been with the Computer and Communication Research Laboratories, NEC, Princeton, NJ, where he is a Senior Research Staff Member. In Spring 2000, he was a Visiting Professor at the University of Notre Dame, IN. Besides hardware/software codesign, his interests include embedded system design methodologies and embedded architectures.

Dr. Henkel is currently a General Co-Chair of the IEEE/ACM International Symposium of Hardware/Software Co-Design CODES’02.



Yanbing Li (S’95–M’98) received the the B.S. degree from Tsinghua University, China, in 1992, the M.S. degree from Cornell University, Ithaca, NY, in 1995, and the Ph.D. degree from Princeton University, Princeton, NJ, in 1998, all in electrical engineering.

She joined Synopsys Inc., Mountainview, CA, in 1998 and has worked on various Synopsys products and research projects. She has published about 20 journal and conference papers in her areas of interest, which include system-level design, hardware-software codesign, compilers, low-power design, and reconfigurable computing.