A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems

Yanbing Li
Department of Electrical Engineering
Princeton University, Princeton, NJ 08544.
yanbing@ee.princeton.edu

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

Abstract Embedded system design is one of the most challenging tasks in VLSI CAD because of the vast amount of system parameters to fix and the great variety of constraints to meet. In this paper we focus on the constraint of low energy dissipation, an indispensable peculiarity of embedded mobile computing systems. We present the first comprehensive framework that *simultaneously* evaluates the tradeoffs of energy dissipations of software and hardware such as caches and main memory. Unlike previous work in low power research which focused only on software or hardware, our framework optimizes system parameters to minimize energy dissipation of the overall system. The trade-off between system performance and energy dissipation is also explored. Experimental results show that our *Avalanche* framework can drastically reduce system energy dissipation.

1 Introduction

The design of embedded systems is a challenging task for today's VLSI CAD environments. As opposed to a general purpose computing system, an embedded system performs just *one* particular application that is known a priori. Therefore, the system can be designed with respect to the particular application to have lower cost, higher performance, or be more energy-efficient. Energy efficiency is a hot topic in embedded system design. As mobile computing systems (e.g. cellular phones, laptop computers, video cams, etc.) become more popular, how to length the battery life of these systems becomes a critical issue.

From the design process point of view, many of the embedded systems can be integrated on just one chip (*systems on a chip*) using core based design techniques. Previous work in core-based system design has mainly focused on performance and cost constraints. Some recent work has been presented in co-synthesis for low power [1, 2]. However, the *trade-off* in energy dissipation among software [1], memory and hardware has not yet been explored. This is a challenging and indispensable task for the design of low power embedded systems. Consider for example, that the use of a bigger cache can reduce the number of cache misses and speed up the software execution, which may cause less energy dissipation on the processor. On the other hand, a larger cache size also causes bigger switching capacitance for cache accesses and therefore increases the cache energy dissipation per access.

In this paper we present our framework *Avalanche*, the first framework that explores the design space of hardware/software systems in terms of overall system energy dissipation. Since embedded system design usually has multiple constraints such as performance and power, our framework evaluates performance as well and optimizes for the best energy-performance trade-off.

[1] We use the term *software energy dissipation* for the energy that is dissipated within a processor core.

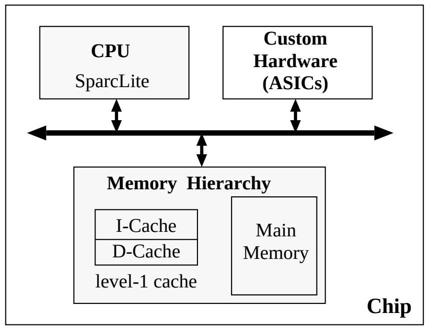


Figure 1: Target architecture of an embedded system

This paper is structured as follows: Sec.2 reviews some of the related work in energy estimation and optimization for embedded systems. Sec.3 describes our model for embedded system energy dissipation. In Sec.4 we present our approach for energy dissipation optimization under timing constraints, and energy and performance trade-off optimization. Experimental results are presented in Sec.5.

2 Related Research

Energy estimation and optimization has been studied for both software and hardware. Tiwari and Malik [3] investigated the energy dissipation during the execution of programs running on different processor cores. Ong and Ynn [4] showed that the energy dissipation 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. [5]. Their tool can be used to conduct architecture-level optimizations.

Further work deals with energy dissipation from a hardware point of view. Gonzales and Horowitz [6] explored the energy dissipation of different processor architectures (pipelined, un-pipelined, super-scalar). Kamble and Ghose [7] analyzed cache energy consumption. Itoh et al. studied SRAM and DRAM energy dissipation and low power RAM design techniques [14]. Panda et al [8] presented a strategy for exploring on-chip memory architecture in embedded systems with respect to performance only. Optimizing energy dissipation by means of high-level transformations has been addressed by Potkonjak et al. [9].

While estimating or optimizing power, these previous work only focuses on one component of the system at a time. A comprehensive approach that takes into consideration the mutual impacts of software *and* hardware in terms of energy dissipation — as it is actually the case in an embedded hardware/software system — has not been addressed so far.

3 System Model and Design Flow

In this section we present our *energy estimation model* of an embedded *system–on–a–chip*. It is based on an architecture template shown in Fig.1, which comprises a processor core, an instruction cache, a data cache, a main memory, and a custom hardware part (ASICs). We assume that hardware/software partitioning has already been performed and the custom hardware is fixed, therefore it adds a *constant* amount of energy to our model. During the design space exploration, we change the software and the cache/memory part, by performing high–level transformations on software and changing the cache and/or main memory parameters. When either of these components changes, the energy dissipation of other components is influenced and so is the overall system energy.

3.1 Analytical Cache Memory Model

We deploy a cache energy model based on transistor-level analysis. The model consists of 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 six CMOS transistors. The switching capacitances in the equations derived below, are obtained by the tool *cacti* [10].

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:

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

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}:

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

where m means a 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:

$$C_{word} = N_{cols} \cdot C_{word,gate} \tag{2}$$

where $C_{word,gate}$ is the sum of the two gate capacitances of the transmission gates in the 6–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 (Eq. 2).

Finally, the total energy dissipated 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:

$$\begin{aligned} E_c &= 0.5 \cdot V_{DD}^2 (N_{acc} \cdot C_{bit,rd} + N_{acc} \cdot C_{word} \\ &\quad + a \cdot C_{bit,wr} + b \cdot C_{dec} + c \cdot C_{od}) \end{aligned} \tag{3}$$

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 $c \cdot C_{od}$ [2] 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 very high accuracy (compared to the real hardware) since every switching transistor within the cache has been taken into consideration (even if this is not transparent through our equations because of the simplification). All the capacitances are obtained by running *cacti* [10] and are derived for a $0.8\mu m$ CMOS technology. The calculation of the capacitances within *cacti* has been proofed against a *Spice* simulation.

[2]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 just one chip.

3.2 Main Memory Energy Model

For energy analysis of the main memory, we use the model for DRAM described by Itoh, $et\ al.$ [14]. The energy source for DRAM mainly includes: the RAM array, the column decoder, the row decoder and peripherals.

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

Eq.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 energy dissipation of each memory access is directly related to the size of the memory. For the total energy dissipation, i_{active} is the dominating component. At high clock frequencies, i_{hld} is negligible [14].

3.3 Software Energy and Performance Model

For software energy estimation we deploy a behavioral simulator ([16]) that we enhanced by values of the current drawn during the execution of an instruction. Those current values are obtained from [12]. The total SW program energy is:

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

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 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 dissipated in the penalty cycles when occurs a data cache write miss, a data read miss and an instruction fetch miss, respectively. We assume that the energy dissipated within processor is negligible after the program has been executed (through gated clock).

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

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

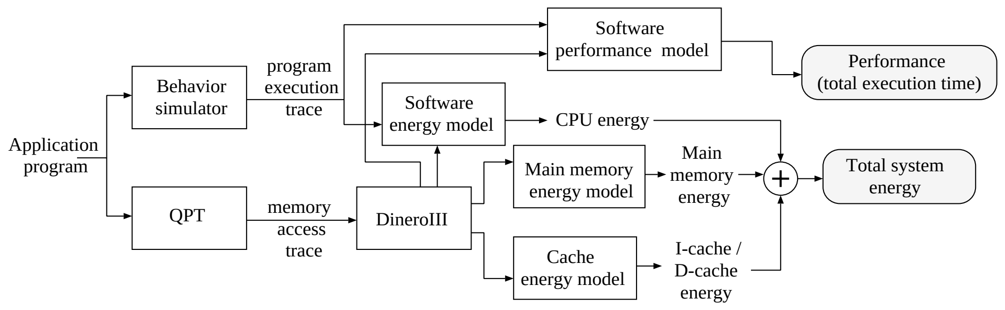


Figure 2: Design flow of the estimation part of our *Avalanche* framework

3.4 Design Flow of Our Framework

Using the above energy models and timing models, the estimation design flow (the energy optimization part is not shown) of our framework is shown in Fig.2. The input is an application program. It is fed into a behavioral model 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* [13] which generates the memory access trace to be used by *Dinero*[13]. *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 get the total execution time with cache miss penalty considered (Eq.6); by the *software energy model* to adjust the software energy with the stalls caused by cache misses (Eq.5); and by the *cache* and *main memory energy models* (Eq.3 and Eq.4) to calculate the energy dissipation by the memory components based on the actual number of instruction/data cache accesses and main memory accesses.

4 System-level Energy Optimization

To optimize the system energy, we explore the design space in the dimensions of software and cache/memory. As mentioned in Sec.3, our framework assumes that the hardware (ASIC) is fixed. It changes the software by performing various high-level transformations. It changes the cache/main memory by modifying their parameters such as size, associativity, etc. When one component (software, cache or memory) changes, it not only affects the energy consumption of itself, but also that of other components in the system; it not only affects the power, but also the performance. The interesting aspect is that the change of overall system energy and performance can not be easily predicted unless comprehensive system analysis is performed. We now discuss some scenarios of software and cache/memory changes and their possible impacts on energy and performance:

- **Software transformation:** suppose a transformation can be performed on the software to lower the *software energy*. However, this transformation may 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, software transformations may increase the code size so that a larger main memory is required to accommodate the new program; therefore, the energy of each memory access increases.
- **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 dissipated by the caches increases due to its increased size, and the system energy change is ambiguous.

```
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 */
  }
  ... ... }
```

Figure 3: Program example for software transformations.

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

4.1 Software Transformation and Energy

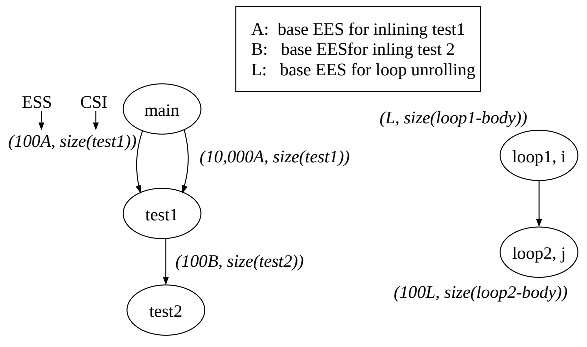
Many source-level transformations have been proposed for the purpose of improving performance. However, they may have some side-effects other than performance improvement, such as a bigger main memory requirement due to increased code size. This will lead to larger energy dissipation due to larger capacitances to switch for each access. Here we have a brief look at some of the 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 8 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. [15], whose impacts on both the software and cache/memory accesses may make it hard to judge the change of the overall system energy dissipation.

4.2 Software Transformation Selection 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



(a) Procedure call graph (b) Loop nesting graph

Figure 4: Procedure calling graph and loop graph of the program example.

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 [11]. 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.

In the first step of the algorithm, to evaluate the impacts of the transformations, we developed some 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. Fig.3 shows a code segment that contains two loops and three procedure calls. The EES for inlining the procedure *test1* at location *A* is 100 times the base EES of inlining *test1*. At location *B*, EES is 10,000 times the base EES. Similar considerations apply to loop unrolling.

To identify the calling relationships, a *procedure calling graph* is constructed (Fig.4) for each program. In the 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 inlining 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 one loop is nested in 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 possible transformations are not independent of each other. For the example in Fig.3: 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; $2)$ a 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 (Sec.4.3).

```
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.
```

Figure 5: System-level energy optimization algorithm.

4.3 System-Level Energy Optimization Algorithm

We now formally define the problem of our system-level energy optimization algorithm. We assume that:
1. Hardware/software partitioning has already been done and application specific hardware is synthesized and therefore fixed.
2. A processor has been chosen.
3. We are given an initial version of the software.
4. The user specifies one *optimization goal*. The algorithm is designed for minimizing energy. However, as power is usually not the sole concern in the design process, we allow three different optimization goals:

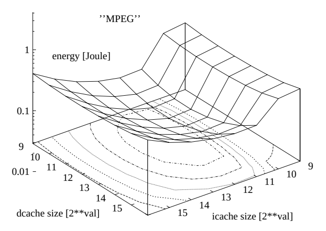
- *Goal I*: minimized power.
- *Goal II*: minimized power under performance constraints.
- *Goal III*: multiple objective optimization

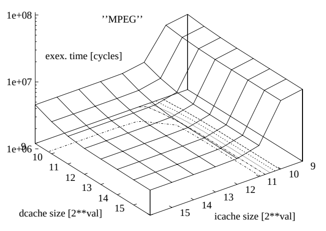
Goal III is to find a set of solutions within performance *and* energy constraints. This will provide important tradeoff information to the designer. The designer can review different design options and choose the most suitable one. For example, there are two designs A and B, with design A being 20% faster than design B, but Design B consuming just 1% less energy. They both meet the performance constraints. *Goal I and II* will discard A, although it might be a better choice for the designer.

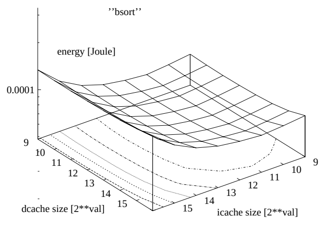
The algorithm returns the optimized new system configuration of the target system architecture (Fig.1): the transformed program, the data cache and instruction cache sizes and other parameters, and the main memory size. The energy dissipation of each component and the performance is also delivered. For *Goal-III*, a set of designs is returned, with percentage data indicating energy and performance difference between two designs adjacent in terms of energy dissipation.

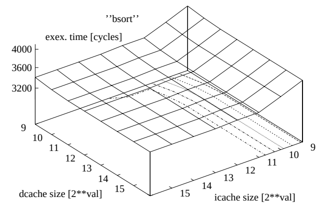
Fig.5 shows the pseudo-code for the optimization algorithm. It consists of two main steps:

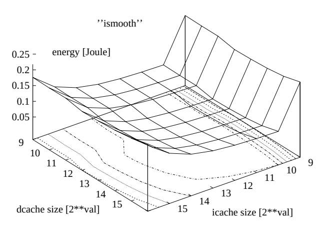
1. **Static analysis of the application program** (lines 1-4), includes
   - generating the procedure calling graph and loop graph, as described in Sec.4.2, and
   - generating the set of feasible cache and memory sizes and configurations based on the current version of the program.


(a) MPEG: energy


(b) MPEG: exec time


(c) bsort: energy


(d) bsort: exec time


(e) i-smooth: energy

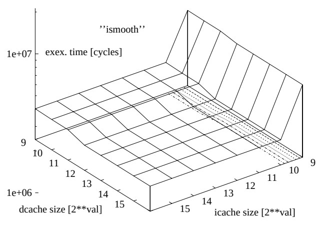

(f) i-smooth: exec time

Figure 6: Energy and execution time vs. instruction/data cache size, for applications *MPEG*, *bsort* and *ismooth*.

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 out-weighs 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 (*I, II or III*) 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 Sec. 4.2 (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. The evaluation depends on the design

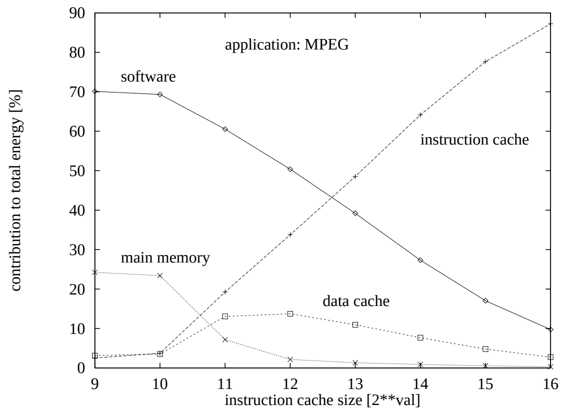


Figure 7: Contribution of software, d-cache, i-cache and main memory to total energy dissipation, at fixed data cache size (4k).

goal: for *minimizing energy* (*Goal I*), energy is the sole standard; for minimized energy under a performance constraints (*Goal II*), the performance constraint is to be met first then energy is considered; for *Goal III*, 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 Sec.3.

5 Experiments and Results

As will be shown in this section, our framework can be deployed to support a system designer through a design space exploration or for automated optimization according to the designers goals (energy, performance). We used data-dominated applications (including an MPEG encoder consisting of about 200kB of C source code), in which caches and memory play a key role in energy and performance.

5.1 Design space explorations

Since the variety of system parameters to choose is very large, we fixed all parameters except for the sizes of data cache and instruction cache. Each application in Fig.6 has been dedicated two figures showing the total energy dissipation of the whole system (left) and the number of clock cycles to perform that application on our target architecture (right). The varied parameters in each figure are the data cache size (left axis) and the instruction cache size right axis). A number of 12 for example means a cache size of 4K ($= 2^{12}$) byte. Figures a) and b) show the results of the MPEG encoder. Here, both large and small cache sizes lead to a high system energy dissipation. 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 Eq.5, large number of cache misses (due to small caches) result in poor performance and 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 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.7 reveals the contribution (in percent) of each component to whole system energy dissipation (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* (c) and d)) application (bubble sort) show mainly that there is almost no dependency on data cache size in terms of system energy dissipation and system performance. More dependencies can be observed by changing the instruction cache size. Obviously, a large instruction cache size leads to a large system energy

appl.	objective	orig. architecture		optim arch. Goal I
		w/o cache	fixed cache	
bsort	energy [J]	0.33	0.30	23.21E-3
	energy improv. [%]	n/a	-9.09	-93.21
	time [# cyc$\times 10^6$]	19.4	17.6	1.4
	exec. improv. [%]	n/a	-9.52	-92.98
eg2	energy [J]	0.31	0.28	19.41E-3
	energy improv. [%]	n/a	-9.68	-93.73
	time [# cyc$\times 10^6$]	17.9	16.5	1,186.8
	exec. improv. [%]	n/a	-8.34	-93.39
ismooth	energy [J]	1.03	0.96	67.411E-3
	energy improv. [%]	n/a	-6.80	-93.45
	time [# cyc$\times 10^6$]	60.1	56.2	3.9
	exec. improv. [%]	n/a	-6.60	-93.46
itimp	energy [J]	2.97	2.78	186.01E-3
	energy improv. [%]	n/a	-6.40	-93.73
	time [# cyc$\times 10^6$]	173.8	162.5	8.8
	exec. improv. [%]	n/a	-6.50	-94.91

Table 1: Optimization of architecture through *GOAL I* compared to original architecture (no cache) and an achitecture with non–optimized cache size

dissipation also. But as opposed to the *MPEG* encoder, a small instruction cache size does not lead to a larger system energy dissipation as a consequence of a larger program execution time. Rather than that, the performance decreases (more cycles due to the mid-right figure on page 6).

An additional different behavior is shown by the application *ismooth*, (e) f)) an image smoothing application. As could be observed, the behavior of a system in terms of energy and performance is hard to predict and therefore needs powerful tools for analyzing and optimizing.

5.2 Optimizing system-level energy dissipation

In Sec.4.3 we have presented our algorithms for three different design goals. For all following experiments we have chosen the clock cycle time to 30ns. The behavior simulation tool for evaluating software performance is a SPARC simulator [16]. The same conventions were used for the experiments in Sec.5.1. All other system parameters are subject to change through the optimization process.

Table.1 shows the results for *Goal I* compared to two original architectures called *w/o cache* (no cache) and the architecture called *fixed cache* (a small standard cache; same size for i-cache and d-cache). For all applications, system energy dissipation (Joule) and execution time (number of clock cycles) is given. Additionally, the relative improvement $(value - value_{ref})/value_{ref} * 100$ for both objectives is given. Apparently, a negative percentage number means an improvement.

As shown in Table.1, *Goal I* yields remarkable improvements in energy and performance. The "fixed cache" architecture is somewhere between "w/o cache" and the optimized architecture. As investigations have shown, in parts, the drastic improvements in performance are due to the architecture: we assume a cache miss penalty of 20 clock cycles, which is a quite typical value. The contribution of software transformation techniques solely (as described in section 4.1) to energy and performance improvements varies in most of the shown cases between 5% and 10%.

Table.2 shows the results yielded with our algorithms for improving energy dissipation under performance constraints (*Goal II* and a multiple objective optimization *Goal III*). In the latter case, the designer is provided with a set of different solutions where he can choose from since design constraints are not completely defined every time. The algorithm for *Goal II* is searching the design space around a given performance constraints i.e. it searches for design configurations

appl.	objective	w/o cache	optimized architecture			
			GOAL II	GOAL III		
bsort	energy [J]	0.33E-3	22.4E-3	0.27	0.24	0.21
	e-improv. [%]	n/a	-93.2	-17.6	-26.8	-36.1
	time [# cyc$\times 10^6$]	19.4	1.4	15.9	14.1	12.3
	t-improv. [%]	n/a	-93.0	-18.4	-27.5	-36.8
eg2	energy [J]	0.31	19.4E-3	0.25	0.23	-
	e-improv. [%]	n/a	-93.7	-18.2	-24.8	-
	time [# cyc$\times 10^6$]	18.0	1.2	14.8	13.6	-
	t-improv. [%]	n/a	-93.4	-17.6	-24.3	-
ismooth	energy [J]	1.03	67.4E-3	0.10	0.72	0.48
	e-improv. [%]	n/a	-93.4	-90.1	-29.8	-53.5
	time [# cyc$\times 10^6$]	60.1	3.9	3.6	42.1	28.0
	t-improv. [%]	n/a	-93.5	-94.0	-30.0	-53.5
itimp	energy [J]	2.97	0.19	2.4	2.2	-
	e-improv. [%]	n/a	-93.7	-19.9	-25.3	-
	time [# cyc$\times 10^6$]	173.8	8.8	139.1	129.5	-
	t-improv. [%]	n/a	-94.9	-20.0	-25.5	-

Table 2: Optimization of architecture through *GOAL II* and *GOAL III* compared to original architecture that has no cache

with minimum energy dissipation while not exceeding the budget of clock cycles to execute. Here also, large improvements could be yielded as shown in Table.2.

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 hours on an Ultra Sparc.

6 Conclusions

We have presented our *Avalanche* framework for estimating and optimizing the energy dissipation of embedded systems. It is the first approach that trades off the energy dissipation of software against the energy dissipation of system resources like caches and main memory. Through various experiments we have shown that it is not straightforward to judge the change of the total system energy when various system parameters are varied and software transformation are performed. Our *Avalanche* framework provides a powerful tool for low power design at system level. Experimental results have shown significant improvements (up to $\approx$ 95% energy cut) in energy dissipation.

References

[1] D. Kirovski, M. Potkonjak, *System-Level Synthesis of Low-Power Hard Real-Time Systems*, Proc. DAC'97, pp.697-702, 1997.

[2] B.P. Dave, G. Lakshminarayana, N.K. Jha, *COSYN: Hardware-Software Co-Synthesis of Embedded Systems*' Proc. DAC'97, pp.703-708, 1997.

[3] V. Tiwari, S. Malik, A. Wolfe, *Instruction Level Power Analysis and Optimization of Software*, Kluwer Academic Publishers, Journal of VLSI Signal Processing, pp. 1–18, 1996.

[4] P.-W. Ong, R.-H. Ynn, *Power-Conscious Software Design – a framework for modeling software on hardware*, IEEE Proc. of Symp. on Low Power Electronics, pp. 36–37, 1994.

[5] T. Sato, M. Nagamatsu, H. Tago, *Power and Performance Simulator: ESP and its Application for 100 MIPS/W Class RISC Design*, IEEE Proc. of Symp. on Low Power Electronics, pp. 46–47, 1994.

[6] R. Gonzales, M. Horowitz, *Energy Dissipation in General Purpose Processors*, IEEE Proc. of Symp on Low Power Electronics, pp. 12–13, 1995.

[7] M.B. Kamble, K. Ghose, *Analytical Energy Dissipation Models For Low Power Caches*, IEEE Proc. of Symposium on Low Power Electronics and Design, pp. 143–148, 1997.

[8] P.R. Panda, N. D. Dutt, A. Nicolau, *Architectural Exploration and Optimization of Local Memory in Embedded Systems*, Proc. of IEEE International Symposium on System Synthesis, pp. 90–97, 1997.

[9] I. Hong, D. Kirovski, M. Potkonjak, *Potential-Driven Statistical Ordering of Transformations*, Proc. DAC'97, pp.347-352, 1997.

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

[11] Stanford Compiler Group, *The SUIF Library: A set of core routines for manipulating SUIF data structures*, Stanford University, 1994.

[12] V. Tiwari, *Logic and system design for low power consumption*, PhD thesis, Princeton University, Nov. 1996.

[13] M. D. Hill, J. R. Laurus, A. R. Lebeck et al., *WARTS: Wisconsin Architectural Research Tool Set*, Computer Science Department University of Wisconsin.

[14] K. Itoh, K. Sasaki and Y. Nakagome, *Trends in Low-Power RAM Circuit Technologies*, Pro. of the IEEE, VOL. 83, No. 4, 1995.

[15] H. Mehta, R. Owens, M.J. Irwin, R. Chen and D. Ghosh, *Techniques for Low Power Software*, IEEE Proc. of Symposium on Low Power Electronics and Design, pp. 72–75, 1997.

[16] W. Ye, R. Ernst, Th. Benner, J. Henkel, *Fast Timing Analysis for Hardware-Software Co-Synthesis*, Proc. ICCD, pp.452–457, 1993.