Energy-Efficient Design of Battery-Powered Embedded Systems

Tajana Simunic† Luca Benini* Giovanni De Micheli†

† Stanford University
Computer Systems Laboratory
Stanford, CA 94305

* DEIS
Universitá di Bologna
Bologna, ITALY 40136

Abstract

Energy-efficient design of battery-powered embedded systems demands optimizations in both hardware and software. In this work we leverage cycle-accurate energy consumption models to explore compiler and source code optimizations aimed at reducing energy consumption. In addition, we extend cycle-accurate architectural power simulation with battery models that provide battery lifetime estmates.

The enhanced simulator and software optimizations are used to study and optimize the power dissipation of SmartBadge, a wearable system based on the ARM microprocessor developed by HP Laboratories. We found that standart compiler optimizations give less than 1% energy savings. Source code optimizations are capable of up to 90% energy savings. In addition, our analysis of battery lifetime for the MPEG decoder implemented on the SmartBadge shows that battery efficiency varies greatly with discharge currents on cycle-by-cycle basis and can cause up to 16% reduction in battery lifetime.

1 Introduction

Quality portable design demands high performance with low thermal dissipation and long battery life. Average energy consumption is directly related to battery life, hence it may be the critical factor that sets system weight and cost.

In our previous work we presented a cycle-accurate energy consumption simulator that is within 5% of measured energy consumption in hardware [3]. We used the simulator to study the power consumption implications of adding MPEG video decode feature to a SmartBadge [2], a wearable computer and communication systems developed at HP Laboratories. The best combination of hardware components for low energy consumption was selected using simulation results – StrongARM-1100 processor with FLASH instruction memory and burst SDRAM for data memory. The simulator also helped in selecting the most energy efficient MPEG stream configuration.

The major contributions of this paper are in three different areas. First we show that the compiler optimizations available with the ARM compiler are not sufficient for power reduction. Next, we analyze and compare different source code optimizations aimed at reducing power consumption and show that significant power savings can be obtained by implementing them. Finally, we extend the cycle-accurate simulator with a battery model that accounts for battery efficiency losses and thus can better aid in the design exploration.

Several techniques for compiler-based energy optimizations have been presented in the past. Tiwari et al. [9, 10] uses instruction-level energy model to develop compiler-driven instruction-level energy optimizations such as instruction reordering, reduction of memory operands, operand swapping in the Booth multiplier, efficient usage of memory banks, and series of processor specific optimizations. Energy efficient register labeling during the compile phase has been suggested as an approach to optimization [5]. Procedure inlining and loop unrolling [12] as well as instruction scheduling [11] have also been investigated. Our cycle-accurate energy consumption simulator presents an integrated framework that is fast enough and accurate enough to estimate the impact of software optimizations on any combination of processor, cache, and memory. In our work we show that the improvements that can be gained using ARM compiler optimizations are marginal compared to writing more energy efficient source code. The largest energy savings are observed at the inter-procedural level that compilers have not been able to exploit. We present a series of suggestions in source code writing style that can save from 1% to over 90% of energy.

Even though energy reduction is an important objective, the ultimate goal of energy optimization for portable systems is battery life optimization. Analytical estimates of the tradeoff between battery capacity and delay in digital CMOS systems are presented in [4]. Battery capacity is strongly dependent on the discharge current as can be seen from any battery data sheet [6]. Hence, it becomes important to accurately model discharge current as a function of time in an embedded system such as SmartBadge. We have extended cycle-accurate energy consumption simulation with a battery model that can predict cycle-by-cycle changes in battery efficiency for any embedded system, including systems that use discrete components where accurate capacitance estimates are not available. In this way accurate estimates of battery lifetime can be obtained.

The rest of this manuscript is organized as follows. An overview of cycle-accurate energy consumption simulator is presented in Section 2. Section 3 discusses the effect ARM compiler optimizations have on the energy consumption.

ISLPED99, San Diego, CA, USA

Source code optimizations aimed at reducing energy consumption are presented in Section 4. Finally, in section 5 we present a new battery model that gives cycle-accurate estimates of battery lifetime.

2 Cycle-accurate system-level energy consumption estimation

The class of embedded systems considered in this paper can be modeled as shown in Figure 1. The system consists of a microprocessor with two levels of cache, off-chip memory, DC-DC converter and battery connected with the interconnect. Selection of the best hardware architecture and software organization given energy and performance constraints is done with help of an instruction-level simulator that has been extended with the energy models for all the system components. Cycle-by-cycle plots of energy consumption can be obtained for each system component. Models for energy consumption and performance estimation of each system component are described in detail in our previous work [3].

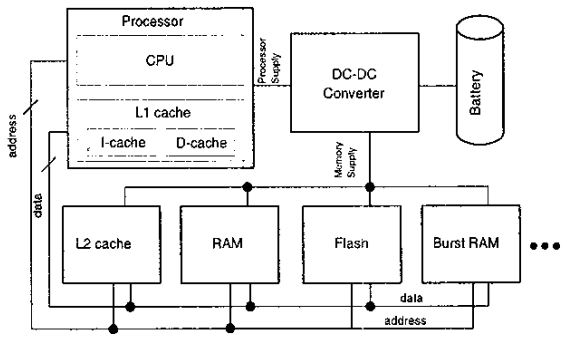

Figure 1: System Model

The total energy consumed during the execution of software on a given hardware architecture is the sum of the energies consumed during the each cycle. The total energy consumed by the system per cycle is the sum of energies consumed by the processor and L1 cache (E_{CPU}), interconnect and pins (E_{Line}), memory ($E_{Mem.}$), L2 cache (E_{L2}) and the DC-DC converter (E_{DC}):

$$E_{Cycle} = E_{CPU} + E_{Line} + E_{Mem.} + E_{DC} + E_{L2} \quad (1)$$

We validated the cycle-accurate power simulator by comparing the computed energy consumption with measurements on the SmartBadge prototype implementation. The SmartBadge prototype consists of the StrongARM-1100 processor, DC-DC Converter, FLASH and SRAM on a PCB board. Industry standard Dhrystone benchmark was used as a vehicle for methodology verification. Simulation results were within 5% of the hardware measurements for the same frequency of operation.

3 Compiler Optimizations

Compilers are typically written to optimize executable size and perfomance. Often it is unclear how such optimization will affect the energy consumption. In this section, we employ cycle-accurate energy consumption simulation to evaluate energy efficiency of various ARM Inc. compiler options on the MPEG decoder. There are two possible ways to optimize code with the ARM compiler - *general* and *specific* compiler optimizations. General command line switches can be used to optimize for code size, execution time or balance of the two. Table 1 shows that for the particular hardware architecture of the SmartBadge while running MPEG algorithm, the optimization for execution time gives lowest energy. Unfortunately, percentage improvements are very small.

Table 1: General Compiler Optimization Options

TYPE	SIZE	TIME	ENERGY
	% change	%change	%change
Balance	0.00	0.00	0.00
Code Size	-0.66	0.20	0.11
Time	0.78	-0.09	-0.10

Three different specific optimizations are possible as well: *cross-jump*, *multiple loads* and *common subexpression elimination*. Cross-jump optimization identifies same sections of code at the end of each case in a switch statement and groups them together. In this way the code size is reduced, but execution time can suffer due to introduction of multiple branches. This type of optimization could be energy efficient for code that contains many switch statements and that is limited by the access time and energy consumption of instruction memory. Multiple loads optimization is specific to the ARM instruction set. A set of sequential load instructions can be replaced by one *load multiple* instruction. Common subexpression elimination looks for patterns in the source code and stores the precomputed value in a register so that recalculation is not needed. This optimization tends to decrease both the execution time and the source code size, but it extends the register lifetime possibly causing more accesses to data memory. Table 2 shows the simulation results for the SmartBadge running MPEG decode algorithm that has been optimized with balance of code size and execution time option, with all special optimizations disabled and with each in turn enabled. Again the differences are very small - less than one percent. Common subexpression elimination did not help at all since it gives same energy consumption as when all three optimizations are disabled. Cross jump optimization causes energy consumption to be larger than when all special optimizations are disabled. Multiple load optimization conserved energy.

Table 2: Specific Compiler Optimization Options

TYPE	SIZE	TIME	ENERGY
	% change	%change	%change
balance	0.00	0.00	0.00
disable all	0.86	-0.26	0.65
cross jump	0.07	-0.21	0.69
multiple loads	0.78	-0.25	0.63
cse	0.86	-0.26	0.65

4 Source Code Optimizations for Lower Energy Consumption

The previous section shows that the compiler optimizations are not sufficient to reduce energy consumption of embedded

software. Various approaches have been presented in [8] for either decreasing the code size or execution speed by changing the source code writing style. This section gives an overview of energy efficient optimizations on the source code level.

4.1 Integer division and modulo operation

The ARM compiler uses shift operation for modulo 2 division since it is much more efficient than the standard division operation. In modulo 2 division unsigned number should be used whenever possible as the unsigned implementation - `div16u` is 14.7% more efficient than the signed version. This is because signed version requires sign extension correction on the shift operation.

```
uint div16u (uint a)     int div16s (int a)
{   return a / 16; }     {  return a / 16;}
```

Whenever possible a condition should be used to replace modulo operation, as it is 51.39% more energy efficient. In example shown below `counter1` implements modulo arithmetic, where `counter2` uses an `if` operator.

```
uint counter1 (uint count)   uint counter2 (uint count)
{                            { if (++count >= 60)
  return (++count % 60);           count = 0;
}                              return (count);    }
```

4.2 Conditional Execution

All ARM instructions can be conditionalized. Conditionalizing is done in two steps. First a few compare instructions set the compare codes. Those instructions are then followed by the standard ARM instructions with their flag fields set so that their execution proceeds only if the preset condition is true. Grouped conditions should be used instead of separate `if` statements since they help the compiler conditionalize instructions. In this way 1.25% of energy can be saved. An example of a grouped condition is show below.

```
if (a > 0 && b > 0 && c < 0 && d < 0)
      return a + b + c + d;
```

4.3 Boolean Expressions

A more energy efficient way to check if a variable is within some range is to use the ability of the ARM compiler to conditionalize the arithmetic function. An example shown below is 10.6% more efficient than if comparison was done on each coordinate separately.

```
Conditionalized example      Original Code
return ((p.x - r->xmin)      return (p.x >= r->xmin &&
          < r->xmax &&               p.x < r->xmax  &&
        (p.y - r->ymin)              p.y >= r->ymin &&
          < r->ymax);                p.y < r->ymax);
```

4.4 Switch Statement vs. Table Lookup

Table lookup is 52.29% more energy efficient than the switch statement when the switch statement codes are more than half of the range of the possible labels. When dense switch statement is used, the table lookup is used to jump to the appropriate case statement. If the case statement contains the call to another function or if it sets a variable, then the table lookup of the address to jump to can be replaced by the code to be executed under the case statement. A good example is shown below where all opcodes are assigned values 0 through 3 thus making the table lookup possible.

```
return  "EQ\0NE\0CS\0CC\0" + 3 * cond;
```

4.5 Register Allocation

Usually a compiler cannot assign local variables to a register if their addresses are passed to other functions. If the copy of the variable is made and the address of the copy is used instead, then variable can be placed in the register thus saving memory access. As much as 9.54% energy savings are possible.

If global variables are used, it is beneficial to make a local copy so that they can be assigned to registers. In this way 6.42% of energy can be saved as compared to using a global copy. An example used is shown below.

```
int errs;
void globTest(void)
{   int localerrs = errs;

    localerrs += f2();
    localerrs += g2();
    errs = localerrs;    }
```

When pointer chains are used, it is energy efficient to store a first reference into a variable so multiple memory lookups are not needed. InitPos2 shown below saves 33.9% of energy over InitPos1.

```
void InitPos1(Object *p)   void InitPos2(Object *p)
{                          {   Point3 *pos = p->pos;
    p->pos->x = 0;             pos->x = 0;
    p->pos->y = 0;             pos->y = 0;
    p->pos->z = 0;   }         pos->z = 0;          }
```

4.6 Variable Types

The most energy efficient variable type for the ARM processor is integer, it saves 0.39% more energy than short and 18.32% more energy than char. Compiler by default uses 32 bits in each assignment, so when either short of char are used sign or zero extending is needed thus costing at most two extra instructions as compared to ints.

4.7 Function Design

By far the largest savings are possible with good function design. Function call overhead on ARM is four cycles. Usually function arguments are passed on the stack, but when there are four or less arguments, they can be passed in registers. A simple example showed over 90% energy savings. Upon return from a function, structures up to four words can be passed through registers to the caller. In this way 72.3% energy can be saved.

When the return from one function calls another, the compiler can convert that call to branch to another function. Energy savings of 49.79% have been observed. An example of such function is shown below.

```
int func1 (int a, int b)
{   if (a > b)
        return (func2(a - b));
    else
        return (func2(b - a));   }
```

Functions that return result that depends only on the value of their arguments and do not have any side-effects can be declared pure. Such functions can then be optimized as common subexpressions by the compiler. Savings of 70% have been shown on a simple example using a square function. Similarly, a functions can be inlined and then no function call overhead is incurred and more optimizations are possible. When square function was inlined we observed

16.89% energy savings. The savings depend highly on the size of the function inlined.

Interprocedural optimization can be done by placing a function definition before its use. An example of that is shown below. Square function is defined before sumsquares, so sumsquares knows what registers square will not use and thus can use those registers for its needs resulting in 24% energy reduction.

```
int square(int x)
{  return x * x;  }

int sumsquares(int x, int y)
{  return square(x) + square(y); }
```

4.8 Complete Example

As a final test of the combined impact of source source code optimization, we have manually optimized example code provided by the ARM Inc. [8]. The original source code contained no energy efficient optimizations. Table 3 shows that both the general and specific compiler optimizations have a very small effect on the original source code in all categories - the maximum savings are only 0.6%. Once energy efficient source code optimizations are implemented, the savings are much larger – as much as 35% in execution time and 32.3% in energy. Clearly the compiler optimizations make almost no difference in this case as well.

Table 3: Complete Example

Energy Opt.	General Opt.	Spec. Opt.	SIZE %change	TIME %change	ENERGY %change
none	Balance	none	0.0	0.0	0.0
none	Time	none	0.0	- 0.2	- 0.2
none	Size	none	0.0	- 0.6	- 0.6
none	Balance	all	0.0	- 0.6	- 0.6
all	Balance	none	-5.8	-35.0	-32.2
all	Balance	all	-5.8	-35.0	-32.3

4.9 Recursion

Recursion has long been thought as very energy inefficient methodology due to high overhead of procedure calls. On the ARM processor the overhead is small - only four instructions, so depending on the type of the problem, recursion can prove to be more energy efficient than the iterative solution. Two examples are presented below - Towers of Hanoi in Figure 2 and Fibonacci series in Figure 3. For a low number of disks in Towers of Hanoi, the iterative solution is more energy efficient due to procedure call overhead. For a large number of disks, the recursive solution consistently outperforms the iterative solution. On the contrary, for the Fibonacci example, the iterative solution consistently outperforms the recursive algorithm. These examples show that recursion can be energy efficient depending on the type of hardware and software used. Cycle-accurate energy consumption simulation is needed to evaluate which approach is best for the given problem.

5 Battery Model

In the previous section, we have presented several techniques that improve the energy efficiency of software running on the CPU. This section presents a battery model that has been

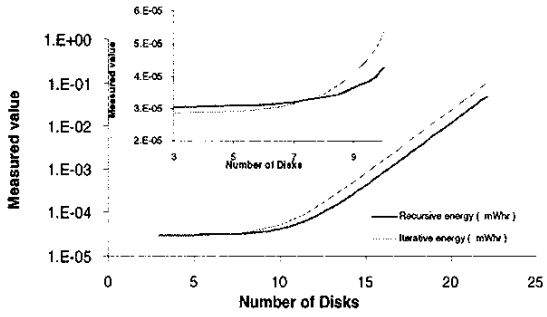

Figure 2: Energy Consumption of Recursive and Iterative Hanoi

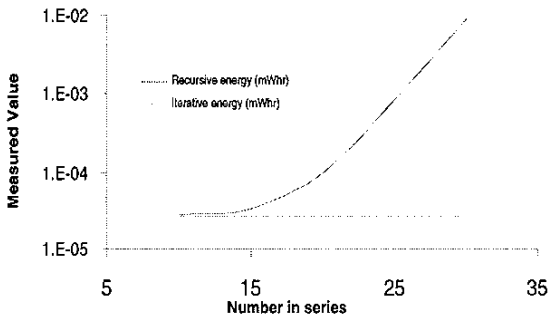

Figure 3: Energy Consumption of Recursive and Iterative Fibonacci

integrated with our simulation environment to provide cycle-accurate estimates of battery lifetime. The main battery characteristic is its rated capacity measured in mWhr. Since total available battery capacity varies with the discharge rate, manufacturers specify plots with discharge rate versus battery efficiency similar to the one shown below.

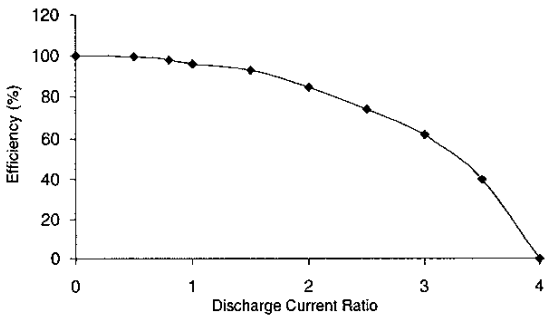

Figure 4: Battery Efficiency

The discharge rate (or discharge current ratio) is given by:

$$R_I = \frac{I_{ave}}{I_{rated}} \quad (2)$$

where I_{rated}, the rated discharge current, is derived from the battery specification and I_{ave} is the average current drawn

by the DC-DC converter. As battery cannot respond to instantaneous changes in current, a first order time constant τ is defined to determine the short-term average current drawn from the battery [7]. Given τ, and processor cycle time T_{cycle}, we can compute N_{bat}, the number of cycles over which average DC-DC current is calculated:

$$N_{bat} = \frac{\tau}{T_{cycle}} \tag{3}$$

then, I_{ave} is computed as:

$$I_{ave} = \frac{1}{N_{bat}} \sum_{cycle=1}^{N_{bat}} I_{system}(cycle) \tag{4}$$

where I_{system} is the instantaneous current drawn from the battery. Battery efficiency is the ratio of actual capacity of the battery to the rated capacity on per-cycle basis:

$$Efficiency = \frac{E_{Cycle}}{E_{Battery}} \tag{5}$$

When battery voltage is nearly constant, the battery efficiency can be defined as a ratio of total current drawn from it by the DC-DC converter to the rated discharge current.

Given the battery capacity model described above, battery estimation is performed as follows. First, the designer characterizes the battery with its rated capacity, the time constant and the table of points describing the discharge plot similar to the one shown in Figure 4. During each simulation cycle discharge current ratio is computed from the rated battery current and average DC-DC current calculated from the last N_{bat} cycles. Efficiency is calculated using linear interpolation between the points from the discharge plot. Total energy drawn from the battery during the cycle is obtained from Equation 5. Lower efficiency means that less battery energy remains and thus the battery lifetime is proportionally lower. For example, if battery efficiency is 60% and its rated capacity is $100mAhr$ at $1V$, then at computed average DC-DC current of $300mA$ battery would be drained in 12 minutes. With efficiency of 100% the battery would last 1 hour.

5.1 Battery Lifetime Analysis for an Embedded MPEG Decoder

The most energy efficient SmartBadge system consisting of the SA-1100 processor with L1 cache, burst SDRAM memory, FLASH and a DC-DC converter has been designed using our cycle-accurate energy consumption simulator. The ratio of SDRAM to processor speed is 1:3. Our simulations show that the group of picture decoded at 30 fr/s with eight I-frames and four P-frames is the most time and energy efficient for MPEG decode implemented on the redesigned SmartBadge. For best battery utilization, it is important to match the current consumption of the embedded system to the discharge characteristic of the battery. On the other hand, the more capacity battery has, the heavier and more expensive it will be. Figure 5 shows that the instantaneous battery efficiency varies greatly over time with MPEG decode running on the hardware described above.

Lower capacity batteries have larger efficiency losses. Figure 6 shows that the total decrease in battery lifetime when continually running MPEG algorithm on a battery with lower rated discharge current can be as high as 16%. The battery's time constant was set to $\tau = 1ms$.

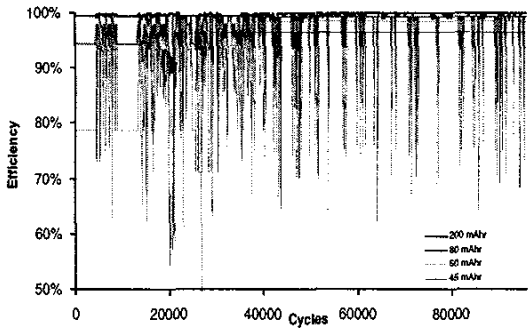

Figure 5: Battery Efficiency for MPEG Decoder

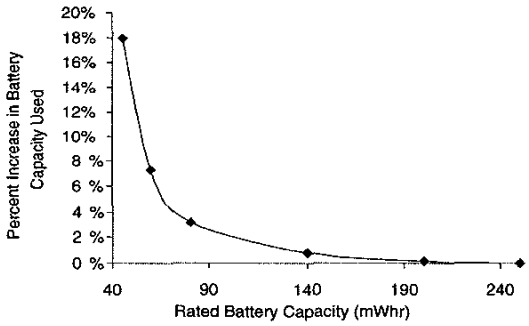

Figure 6: Percent Decrease in Battery Lifetime for MPEG Decode

6 Conclusions

Cycle-accurate battery model and energy efficient source code optimizations have been presented in this paper. ARM compiler optimizations were shown to be largely ineffective in reducing energy consumption of a larger MPEG design. On the other hand, a series of source code optimizations aimed at reducing energy consumption were shown to be very promising, in some cases offering up to 90% of reduction in energy consumption. Traditional methods that first estimate average current consumption in the system and then employ average current to estimate battery capacity do not model accurately battery efficiency and can thus give erroneous estimates of the battery capacity needed. A cycle-accurate battery model better predicts battery lifetime, as battery energy efficiency depends strongly on cycle-by-cycle system current consumption.

7 Acknowledgments

The authors would like to thank Mark Smith for his help. This work was sponsored by the Hewlett-Packard Laboratory and ARPA/MARCO GSRC.

References

[1] Advanced RISC Machines Ltd (ARM), *ARM Software Development Toolkit Version 2.11*, 1996.

[2] G. Q. Maguire, M. Smith, H. W. Peter Beadle, "SmartBadges: a wearable computer and

communication system," *Invited talk slides url: www.it.kth.se/maguire/Talks/CODES-980313.pdf*, 6th International Workshop on Hardware/Software Codesign, 1998.

[3] T. Simunic, L. Benini, G. De Micheli, "Cycle-Accurate Simulation of Energy Consumption in Embedded Systems," *to appear in Proceedings of DAC*, 1999.

[4] M. Pedram, Q. Wu, "Battery-Powered Digital CMOS Design," *Proceedings of DATE*, 1999.

[5] H. Mehta, R.M. Owens, M.J. Irvin, R. Chen, D. Ghosh, "Techniques for Low Energy Software," *Proceedings of ISLPED*, pp. 72–75, 1997.

[6] "Commercial NiMH Technology Evaluation," *Proceedings of the 12th Battery Conference*, p.9–15,1997.

[7] "A PSPICE Macromodel for Lithium-Ion Battery Systems," *Proceedings of the 12th Battery Conference*, p.215–222,1997.

[8] "Writing Efficient C for ARM," *Application Note 34*, ARM Inc., January 1998.

[9] V. Tiwari, S. Malik, A. Wolfe, M. Lee, "Instruction Level Power Analysis and Optimization of Software," *Journal of VLSI Signal Processing Systems*, vol 13, no.2-3, pp.223–2383, 1996.

[10] V. Tiwari, S. Malik, A. Wolfe, "Power Analysis of Embedded Software: A First Step Towards Software Power Minimization," *IEEE Transactions on VLSI Systems*, vol. 2, no.4, pp.437–445, December 1994.

[11] H. Tomyiama, H., T. Ishihara, A. Inoue, H. Yasuura, "Instruction scheduling for power reduction in processor-based system design," *Proceedings of Design, Automation and Test in Europe*, pp. 23-26, February 1998.

[12] Y. Li and J. Henkel, "A Framework for Estimating and Minimizing Energy Dissipation of Embedded HW/SW Systems," *Proceedings of DAC 1998*, pp.188–193, 1998.