# Source Code Optimization and Profiling of Energy Consumption in Embedded Systems

Tajana Šimunić, Luca Benini*, Giovanni De Micheli and Mat Hans†
Computer Systems Lab, Stanford University
*DEIS University of Bologna, Italy
†HP Labs, Palo Alto
tajana@polaris.stanford.edu

## Abstract

This paper presents a source code optimization methodology and a profiling tool that have been developed to help designers in optimizing software performance and energy in embedded systems. Code optimizations are applied at three levels of abstraction: algorithmic, data and instruction-level. The profiler exploits a cycle-accurate energy consumption simulator [3] to relate the embedded system energy consumption and performance to the source code. Thus, it can be used for analysis (i.e., to find energy-critical sections of the code), and for validation (i.e., to assess the impact of each code optimization).

Code optimizations and profiling tool are used to optimize and tune the implementation of an MPEG Layer III (MP3) audio decoder for the SmartBadge [2] portable embedded system. We show that using our methodology and tool we can quickly and easily redesign the MP3 audio decoder software to run in real time with low energy consumption. Performance increase of 92% and energy consumption decrease of 77% (over the original executable specification) has been achieved for MP3 audio decoding on the SmartBadge.

## 1 Introduction

Low cost with fast time to market is the top requirement in system-level design of embedded portable appliances. As a result, typical portable appliances are built of commodity components with a microprocessor-based architecture. The design process for such portable embedded systems starts with the selection of the commodity components that could meet the performance and the energy consumption criteria, based on the analysis of data sheets. Typically only a few processor families can be evaluated due to resource and time limitations. Whole system evaluation is often done on prototype boards resulting in long design times. FPGA hardware emulators are sometimes used for functional debugging. Emulation cannot give accurate estimates of energy consumption or performance since it uses FPGA technology that has different energy and performance characteristics. An alternative approach, called *virtual prototyping*, relies on system-level simulation to explore many different hardware and software architectures and get accurate performance and energy consumption estimates.

Virtual prototyping can substantially reduce design effort and time-to-market. Unfortunately, CAE support to virtual prototyping for embedded system design is still limited. Commercial tools target mainly functional verification and performance estimation [5, 6, 7, 8], but they provide no support for power-related cost metrics. A few research prototype tools [10, 11] have been proposed that separately estimate energy consumption of processor core, caches and main memory in SOC design. The final system energy is obtained by summing over the contribution of each component. The main limitation of these approaches is that the effect of the interaction between memory system (or I/O peripherals) and processor is not modeled. Processor energy consumption is generally estimated by *instruction-level power analysis*, first proposed by Tiwari et al. [12, 13]. This technique estimates the energy consumed by a program by summing the energy consumed by the execution of each instruction. Instruction-by-instruction energy costs are pre-characterized once for all for each target processors. The instruction-level power model can be augmented by considering the effect of first-level caches and inter-instruction effects. The shortcomings of previous approaches are addressed in [3, 4], where memory models and processor instruction-level simulator are tightly integrated into a cycle-accurate simulation engine. Estimation results were shown to be within 5% of measured energy consumption in hardware.

Several techniques for code optimization have been presented in the past. Tiwari et al. [12, 13] uses instruction-level energy models to develop compiler-driven 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. In addition, several other optimizations have been suggested, such as energy efficient register labeling during the compile phase [14], procedure inlining and loop unrolling [10] as well as instruction scheduling [15]. All of these techniques focus on automated instruction-level optimizations driven by the compiler. Even though these techniques may be very helpful once integrated into an industry-strength optimizing compiler, currently available commercial compilers have limited capabilities. In [4], it was shown that the improvements that can be gained using standard compiler optimizations are marginal compared to writing more energy efficient source code. The largest energy savings were observed at the inter-procedural level that compilers have not been able to exploit.

Accurate virtual prototypes have been exploited to drive the selection of the best hardware architecture given energy and performance requirements [10, 3]. In this work we explore another area of application, namely, *software optimization*. In an industrial environment, cost constraints, standardization requirements, backward compatibility issues and time-to-market pressure drastically reduce the hardware organization options that can be explored. On the other hand, a lot more freedom is available for software design. First, application-specific embedded software can be developed more rapidly than dedicated hardware. Second, many embedded applications are specified using a software programming language (e.g., C, Matlab). Mapping a software-only specification into an embedded system is a more straightforward process than performing hardware-software partitioning and co-design. Finally, it is much easier and less expensive to debug or update software than to modify incorrect or incomplete hardware.

Embedded software optimization requires methodologies and tools for: (i) transforming a sub-optimal executable specification into an optimized embedded application; (ii) estimating the impact of program transformations on power consumption and performance. To address these issues, we first developed a code transformation methodology that has been helpful in driving the energy (and performance) optimization of embedded applications. We present three categories of source code optimizations: algorithmic changes, data representation changes and instruction-level optimizations.

The second contribution of the paper is a code profiling tool that provides detailed feedback on both performance and power of all components of a program, starting from procedures and functions down to the single instruction. Profiling is typically used to relate performance to the source code. Often performance profiling for embedded systems takes only processor and L1 cache performance into account, but does not include L2 cache, off-chip memory models and interconnect. For example, call-graph performance profiling in the ARM Ltd. instruction-level simulator [1] handles only processor and L1 cache models. Our profiler gives percentages of time and energy spent in each procedure for every system component (in addition to the total performance, energy consumption and cycle-by-cycle plots) accounting for the effect of L2 cache, main memory and system busses.

The remainder of this paper is organized as follows. Our code optimization methodology is described in Section 2, where code transformations are discussed in detail. The design tool support we have developed is presented in Section 3. A full software design example of MP3 audio decoder for the SmartBadge, with extensive experimental results, is given in Section 4.

## 2 Code Optimization

In our context, code optimization is the process of translating a high-level specification in an imperative language into optimized machine code for the target processor. Compilers are the tools of choice for code optimization. Extensive research on *optimizing compilers* has been carried out in last few years [28]). Prototype research compilers have shown impressive results [26]. Most optimizing compilers target high-performance and/or general-purpose computers, and relatively little effort has been dedicated to create powerful optimizing compilers for embedded processors. Even though several researchers are studying automatic code optimization techniques for embedded processors [29], currently, most embedded processors (or DSPs) are programmed directly in assembly by expert programmers and code optimization is mostly based on human intuition and skill.

Given the limited compiler support available, our approach to code optimization for embedded systems is still mostly based on manual code re-writing and optimization. The main advantage of our approach is that it enables designers to focus first on a very abstract view of the problem, find a good solution, then move down in abstraction, and perform optimizations that are narrower in scope. The complex problem of optimizing an executable specification is partitioned, and its parts are more manageable than the complete problem. Furthermore, different people can work on different optimization layers and parallelize the effort. In the next subsections, we will describe in detail the three optimization layers defined in out methodology, moving from high to low abstraction. We will illustrate our methodology on optimization of MP3 code [25] for the SmartBadge portable device [2].

### 2.1 Algorithmic optimization

The top layer in the optimization hierarchy targets algorithms. The original specification is first profiled to identify all computational kernels, i.e., the procedures where most time and power are spent. Alternative algorithms for implementing the same functionality are considered and compared with the original one using high-level estimators of algorithmic efficiency (such as number of basic operations). Most promising alternative algorithms are then analyzed in more detail and finally coded. This step is mostly based on human intuition and knowledge, and is unlikely to be automated.

Algorithmic optimizations have high potential, but they also have risks. First, developing and testing algorithms is a time-consuming and error-prone task. Since human resources are always scarce, it is unwise to dedicate too much effort to an activity where success is often based on intuition. Second, asymptotic analysis and operation counts are often misleading as estimators of algorithmic efficiency, hence marginal improvements should be regarded with suspicion when considering algorithmic changes.

For these reasons, our approach to algorithmic optimization in MP3 decoding has been conservative. First, we focused on just one computational kernel where a large fraction of run time and energy was spent, namely the *subband synthesis*. Second, we did not try to develop new original algorithms but we used previously published algorithmic enhancements [19, 20] that are still fully compliant to the MPEG standard. The new algorithm incorporates an integer implementation of the scaled Chen discrete cosine transform (DCT) instead of a generic DCT in the polyphase synthesis filterbank. The use of a scaled DCT reduces the DCT multiply count by 28%.

### 2.2 Data optimization

At a lower level of abstraction than the algorithmic level, we can optimize code by changing the representation of the data manipulated by the algorithms. The main objective is to match the characteristics of the target architecture with the processed data. Signal processing algorithms are often specified by assuming double-precision floating point data to avoid overflows and keep accuracy under control. Floating point computations are usually more complex and power-hungry than their integer counterparts. As no hardware floating point support is available in the ARM SA-1100 and the MPEG decoder specification performed most computations using doubles, we tried to emulated floating point using ARM's software library. The direct implementation of the decoding algorithm, even after algorithmic optimization, was unacceptably slow and power-consuming.

To overcome this problem, we developed a fixed-precision library and we implemented all computational kernels of the algorithm using fixed precision numbers. The number of decimal digits can be set at compile time. The ARM architecture is designed to support computation with 32-bits integers with maximum efficiency. Little can be gained by reducing data size below 32 bits. On the other hand, when multiplying two 32-bit numbers, the result is a 64-bit number and directly truncating the result of a multiplication to 32 digits frequently leads to incorrect results because of overflow. To increase robustness, 64-bit numbers have been used for fixed-point computation. This data type is supported by the ARM compiler through the definition of a `long long` integer type. Computing with `long long` integers is less efficient than using 32-bit integers, but results are accurate and the risk of overflow is minimized.

Data optimization produced significant energy savings and speedups for computational kernels of MP3 without any perceivable degradation in quality. The fixed-point library developed for this purpose contains macros for conversion from fixed-point to floating point, accuracy adjustment and elementary function computation. This optimization did not require extensive code rewriting, and it was implemented independently from algorithmic optimization.

### 2.3 Instruction flow optimization

The third layer of optimizations targets low-level instruction flow. After extensive profiling, the most critical loops are identified and carefully analyzed. Source code is then re-written to make computation more efficient. Well-known techniques such as loop merging, unrolling, software pipelining, loop invariant extraction, etc. [28, 27] have been applied. In the innermost loops, code can be written directly as inline assembly, to better exploit specialized instructions.

Instruction flow optimizations have been extensively applied in the MP3 decoder, obtaining significant speedup. We do not describe these optimizations in detail because they are common knowledge in the optimizing compilers literature [28, 27]. However, in our case most optimizations were performed manually due to lack of support by the ARM compiler.

A simple example of this class of transformation is the use of the multiply-accumulate instruction (`MLAL`) available in the ARM SA-1100 core. The inner loops of subband synthesis and inverse modified cosine transform (the two key computational kernels of MP3 decoder), contain matrix multiplications which can be implemented efficiently with multiply-accumulate. In this case, we forced the ARM compiler to use the `MLAL` instruction by inlining it in assembly.

Summarizing this section, we described three code optimization layers that have been useful to optimize MP3 decoding. We found that layering optimizations for decreasing levels of abstraction, and working on each level separately, was a very effective way to tackle the non-trivial task of speeding up and reducing the energy consumed in executing the original specification by more than an order of magnitude. In principle, stepwise optimization may reduce optimality. In practice, it often helps in finding better heuristic solutions in a shorter time. Many of the optimizations we applied manually could be automated, even though automation becomes more problematic as the level of abstraction raises. During code optimization, tool support was essential: code profiling was by far the most useful source of information to direct optimization, and assess its impact. In the next section we will describe the profiling tool that has been developed to support code optimization.

## 3 Profiler for Energy and Performance

The class of embedded systems considered in this paper consists of a microprocessor with two levels of cache, off-chip memory, DC-DC converter and battery. Previous work extended the ARMulator, a proprietary instruction-level performance simulator from ARM inc., with cycle-accurate energy models for all system components in [3]. The cycle-accurate energy consumption simulator can give cycle-by-cycle plots of energy consumption for each system component, thus enabling very detailed hardware and software architecture analysis. Simulation results with simulator were shown to be within 5% of the hardware measurements for the same frequency of operation when running the Dhrystone benchmark on the SmartBadge [2].

The main limitation of cycle-accurate energy simulator is that the impact of code optimizations is not easily evaluated. For example, in order to evaluate energy efficiency of two different implementations of a particular portion of software, the designer would need to obtain cycle-by-cycle plots and then manually relate cycles to the software portion of interest. The profiling methodology presented next addresses this limitation.

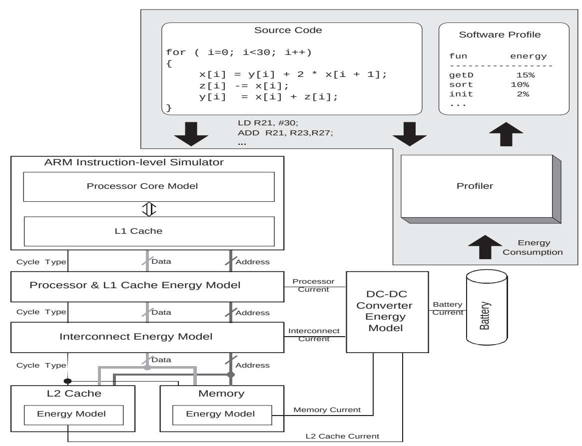


Figure 1: Profiler Architecture

The profiler is shown in Figure 1. Shaded portion represents the extension we made to the cycle-accurate energy simulator architecture to enable code profiling. Profiling for energy and performance enables designers to identify those portions of their source code that need to be further optimized in order to either decrease energy consumption, increase performance or both. Our profiler enables designers to explore multiple different hardware and software architectures, as well as to do statistical analysis based on the input samples. In this way the design can be optimized for both energy consumption and performance based on the expected input data set.

The profiler operates as follows. Source code is compiled using a compiler for a target processor. The output of the compiler is the executable that the cycle-accurate simulator executes (represented in this figure as assembly code that is input into the simulator) and a map of locations of each procedure in the executable that a profiler uses to gather statistics (the map is correspondence of assembly code blocks to procedures in 'C' source code). In order to increase the simulation speed, a user-defined profiling interval is set, so that the profiler gathers statistics only at predetermined time increments. Usually an interval of $1\mu s$ is sufficient.

During each cycle of operation, the cycle-accurate energy consumption simulator calculates the current total execution time and energy consumption of all system components as shown in Equation 1. 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 ($E_{L2}$), the DC-DC converter ($E_{DC}$) and the efficiency losses in the battery ($E_{Bat.}$) [3, 4].

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

The profiler works concurrently with the cycle-accurate simulator. It periodically samples the simulation results (using sample interval specified by the user) and maps the energy and performance to the function executed using information gathered at the compile time. Once the simulation is complete, the results of profiling can be printed out by the total energy or time spent in each function.

Table 1: Sample Energy Profiling

| Name | Cumulative (mWhr) | Self (mWhr) |
|---|---|---|
| main | 3.20E-01 | 2.52E-02 |
| ... | | |
| III_hybrid | | 6.71E-02 |
| SubBandSynthesis | | 3.72E-02 |
| III_stereo | | 2.75E-02 |
| III_reorder | | 2.02E-02 |
| III_antialias | | 1.45E-02 |
| III_dequantize_sample | | 1.40E-02 |
| III_hufman_decode | | 3.74E-03 |
| III_get_scale_factor | | 1.28E-04 |
| decode_info | | 3.20E-05 |
| ... | | |
| III_hybrid | 6.71E-02 | 6.36E-03 |
| inv_mdctL | | 6.07E-02 |
| SubBandSynthesis | 3.72E-02 | 1.95E-02 |
| chendct32_scaled | | 1.77E-02 |
| III_stereo | 2.75E-02 | 2.75E-02 |
| III_reorder | 2.02E-02 | 2.02E-02 |
| III_antialias | 1.45E-02 | 1.45E-02 |
| III_dequantize_sample | 1.40E-02 | 1.40E-02 |
| III_hufman_decode | 3.74E-03 | 1.53E-03 |
| huffman_decoder | | 2.17E-03 |
| initialize_huffman | | 1.03E-05 |
| hsstell | | 3.20E-05 |

The main advantage of the profiler is that it allows designers to obtain energy consumption breakdown by procedures in their source code after running only one simulation. This information is of critical importance when designing an embedded system, as it enables designers to quickly identify and address the areas in the source code that will provide largest overall energy savings. A good example of profiler usage is shown in Table 1. The table shows a portion of energy profile for MP3 audio decode. The first column gives the name of the top procedure, followed by its children. The next column gives the total energy spent for that procedure. For example, the total energy spent running the program (`main`) is $0.32mWhr$. The final column gives the amount of energy spent only in that particular procedure. For example, under `main` it is clear that `III_hybrid` and its descendants spend the most energy, $0.0671mWhr$. Looking at the entry for `III_hybrid`, it is easy to see that the largest portion of energy is consumed by its child, `inv_mdctL`. Therefore, the procedures to focus optimization on are `inv_mdctL` and `SubBandSynthesis`. Although in this example we showed source code profile of total battery energy consumption, the profiler can report energy consumption for any system component, such as SRAM or the interconnect.

The profiler allows for fast and accurate evaluation of software and hardware architectures. Most importantly, it gives good guidance to the designer during the design process without requiring manual intervention needed in the simulator without the profiler. In addition, the profiler accounts for all embedded system components, not just the processor and the L1 cache as most general-purpose profilers do. In the next section we present a real design example that uses the profiler to guide the implementation of the source code optimizations described earlier for the MP3 audio decoder running on the SmartBadge.

## 4 Optimizing MP3 audio decoder

We optimized the implementation of the MP3 audio decoder for the SmartBadge portable device [2]. The SmartBadge is an embedded system consisting of the StrongARM-1100 processor, FLASH, SRAM, sensors, and modem/audio analog front-end on a PCB board powered by the batteries through a DC-DC converter. The hardware prototype of the SmartBadge uses a standard PCB with line delay of $71ps/cm$ and stripline and microstrip capacitances of 1.6 and $1.1pF/cm$ respectively. The characteristics of CPU and memory chips are given in Table 2.

Table 2: SmartBadge CPU and Memory Configuration

| Component Units | Cycle T. (ns) | Active P (mW) | Idle P (mW) | Pin Cap. (pF) | Line L. (cm) |
|---|---|---|---|---|---|
| SA-1100 | 5-20 | 400 | 170 | 5 | N/A |
| FLASH (1MB) | 80 | 74 | 0.5 | 10 | 2 |
| SRAM (1MB) | 90 | 55 | 0.01 | 8 | 3 |

We obtained the original MP3 audio decoder software from the International Organization for Standardization [18]. Our design goal was to obtain real-time performance with low energy consumption while keeping in full compliance with the MPEG standard. The block diagram of the MP3 decoding algorithm is shown in Figure 2. It consists of three blocks: frame unpacking, reconstruction, and inverse mapping. The first step in decoding is synchronizing the incoming bitstream and the decoder. Huffman decoding of the subband coefficients is performed before requantization. Stereo processing, if applicable, occurs before the inverse mapping which consists an inverse modified cosine transform (IMDCT) followed by a polyphase synthesis filterbank.


Figure 2: MP3 Audio Decoder Architecture

### 4.1 Experimental results of software optimization

We first profiled the original source code to highlight areas where improvement is needed. Without the profiler, we could have obtained the total energy consumption for running whole code and cycle-by-cycle plots. In order to find out where most energy consumption occurs, we would have needed to run a series of cycle-by-cycle plots, each time focusing on a different function. With the profiler, we only need to run the simulation once to obtain the breakdown of energy spent per each function. In addition, the profiler enabled us to identify the key issues in code optimization and allowed us to proceed with the optimizations in parallel.

Table 3: Profiling for MP3 Implementations

| MP3 Code Rev. | 1st | 2nd | 3rd |
|---|---|---|---|
| Original code | Floating Pt. 80.31% | SubBandSynthesis 10.31% | III_stereo 1.43% |
| Algorithmic Opts. | Floating Pt. 62.73% | III_stereo 6.12% | III_reorder 5.62% |
| Data & Instruction | SubBandSynthesis 34.32% | inv_mdctL 18.22% | III_stereo 7.32% |
| Combined Opts. | inv_mdctL 18.98% | III_stereo 8.61% | main 7.87% |

Table 3 shows the top three functions in energy consumption for each code revision we worked on. The original code has a very large overhead due to floating point emulation - about 80% of energy consumption. The next largest issue is the redesign of SubBandSynthesis function that implements the polyphase synthesis filterbank. The details of each optimization type, namely algorithmic, data and instruction-level optimizations, have been presented in Section 2.

We will use the SubBandSynthesis function redesign as a vehicle to illustrate the use of our profiler. In the initial stage, we transferred all critical operations to fixed-point from floating point. The transfer resolved the issue with floating-point operations, but at the same time increased SubBandSynthesis fraction of total energy six times. Next we introduced a series of instruction-level optimizations that resulted in 30% decrease of SubBandSynthesis fraction of total energy, to 34.32% as shown in the Table 3. In parallel we had decided to try the algorithmic changes on the current code.

Profiling results in Table 3 show that the algorithmic optimizations considerably reduced the energy consumption of SubBandSynthesis function - it does not appear in the top three functions, and in fact it is only 3.2% of the total energy consumption. The final step is to combine the algorithmic changes with the data and instruction-level changes, resulting in decrease of SubBandSynthesis fraction of energy consumption to 6% of total.

Table 4: Energy for MP3 Implementations

| MP3 Code Revision | Battery (mWhr) | CPU (mWhr) | Flash (mWhr) | RAM (mWhr) | DC-DC (mWhr) | Lines (mWhr) |
|---|---|---|---|---|---|---|
| Original code | 0.446 0% | 0.089 0% | 0.005 0% | 0.178 0% | 0.045 0% | 0.129 0% |
| Algorithmic Opts. | 0.107 76% | 0.020 77% | 0.007 -44% | 0.040 77% | 0.011 76% | 0.029 77% |
| Data & Instruction | 0.130 71% | 0.025 71% | 0.004 27% | 0.051 71% | 0.013 71% | 0.037 71% |
| Combined Opts. | 0.105 77% | 0.019 78% | 0.007 -41% | 0.040 78% | 0.010 77% | 0.028 78% |

System and component energy consumptions are shown in Table 4 for different revisions of source code optimization. Positive percentage of energy decrease with respect to the original code is shown as well. Table 5 shows the same results, but for performance measurements. The positive percentages show performance increase. Although the energy savings of algorithmic versus data and instruction-level optimizations as compared to original code are comparable, the performance improvement of data and instruction-level optimizations is significant. Note that the increase in energy consumption and the decrease in performance of Flash is due to the increase in code size with the algorithmic change in SubBandSynthesis procedure. The total improvement in system performance and energy consumption more than makes up for the degradation of Flash performance and energy consumption. Combined optimizations give real-time performance for MP3 audio decode which is a primary constraint for this project.

Table 5: Performance for MP3 Implementations

| MP3 Code Revision | System (s) | Flash (s) | RAM (s) |
|---|---|---|---|
| Original code | 68.490 0% | 0.396 0% | 6.309 0% |
| Algorithmic Opts. | 34.562 50% | 0.746 -88% | 2.776 56% |
| Data & Instruction | 9.185 87% | 0.381 4% | 4.186 34% |
| Combined Opts. | 5.193 92% | 0.718 -81% | 2.093 67% |

The final MP3 audio decoder compliance to the MPEG standard has been tested as a function of precision for fixed-point computation. We used the compliance test provided by the MPEG standard [22, 24]. The range of RMS error between the samples defines the compliance level. Table 6 shows that results. Clearly, the larger number of precision bits results in better compliance. In our final MP3 audio decoder we used 27 bits precision.

Table 6: Fixed-point Precision and Compliance

| Precision # bits | Compliance |
|---|---|
| 15 | None |
| 20 | Partial |
| 27 | Full |

### 4.2 Profiling for different hardware configurations

The design tools described in Section 3 can be used to evaluate energy consumption and performance for the different hardware configurations in addition to different source code revisions. Table 7 shows comparison of energy consumption and performance for each change in hardware with respect to the original SmartBadge configuration while keeping the source code the same. Positive percentage indicates an increase in energy or decrease in performance. Change of CPU to ARM710a causes a large increase in energy consumption and a decrease in performance. Burst SDRAM increases performance by 26% at the expense in energy consumption increase of 147%.

## 5 Conclusions

We have presented in this paper a methodology for source code optimizations and a tool for profiling energy consump-

Table 7: MP3 Results for Different Hardware Configurations

| Hardware Change | Energy (mWhr) | Performance (s) |
|---|---|---|
| Final Config. | 0.105<br>0% | 5.193<br>0% |
| CPU ARM710a | 1.709<br>1534% | 19.78<br>281% |
| SDRAM 15ns Burst | 0.258<br>147% | 3.850<br>-26% |

tion and performance of software in embedded systems. Our profiler is based on the cycle-accurate energy consumption simulator that has been shown to give simulation results that are within 5% of hardware measurements [3]. Three major categories of software optimizations have been presented: algorithmic, data and instruction-level.

We gave an example of application of our methodology and the profiling tool to the optimization of MP3 audio decoding for the SmartBadge [2] portable embedded system. Profiling results enabled us to quickly and easily target the redesign the MP3 audio decoder software. In addition, we showed the results of evaluating different hardware configurations using our design tools.

Our final MP3 audio decoder is fully compliant with the MPEG standard and runs in real time with low energy consumption. Using our design tools and the methodology for source code optimization we have been able to increase performance by 92% while decreasing energy consumption by 77%.

## 6 Acknowledgments

The authors would like to thank John Dias and Mark Smith for their help. This work was supported by the Hewlett-Packard Laboratory and NSF grant CCR-9901190.

## 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," *Proceedings of DAC*, 1999.

[4] T. Simunic, L. Benini, G. De Micheli, "Energy-Efficient Design of Battery-Powered Embedded Systems," *Proceedings of ISLPED*, 1999.

[5] CoWare, *CoWareN2c url:www.coware.com/n2c.html* .

[6] Mentor Graphics, *www.mentor.com/codesign*.

[7] Synopsys, *www.synopsys.com/products/hwsw*.

[8] Cadence, *www.cadence.com/alta/products*.

[9] P. Landman, J. Rabaey, "Activity-Sensitive Architectural Power Analysis," *IEEE Transactions on CAD*, pp.571–587, June 1996.

[10] 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.

[11] B. Kapoor, "Low Power Memory Architecutres for Video Applications," *Proceedings of the 8th Great lakes symposium on VLSI*, pp. 2–7, 1998.

[12] V. Tiwari, S. Malik, A. Wolfe, M. Lee, "Instruction Level Power Analysis," *Journal of VLSI Signal Processing Systems*, no.1, pp.223–2383, 1996.

[13] 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.

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

[15] 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.

[16] M. Wan, Y. Ichikawa, D. Lidsky, J. Rabaey, "An Energy Conscious Methodology for Early Design Exploration of Heterogeneous DSPs," *Proceedings of the Custom Intergrated Circuit Conference*, 1998.

[17] "Fixed Point Arithmetic on the ARM," *Application Note 33*, ARM Inc., September 1996.

[18] "Coded representation of audio, picture, multimedia and hypermedia information," *ISO/IEC JTC/SC 29/WG 11*, Part 3., May 1993.

[19] M. Hans and V. Bhaskaran, "A Compliant MPEG-1 Layer II Audio Decoder with 16-bit Arithmetic Operations," *IEEE Signal Processing Letters*, vol. 4, no. 5, May 1997.

[20] M. Hans, "An MPEG Audio Decoder Based on 16-bit Integer Arithmetic and SIMD Usage," *Proceedings of the First IEEE Workshop on Multimedia Signal Processing*, Princeton, N.J., June 1997.

[21] ISO/IEC JTC 1/SC 29/WG 11 11172-3, "Information Technology — Coding of moving pictures and associated audio for digital storage media up to 1.5 Mbit/s — Part 3: Audio," *International Organization for Standardization*, May 1993.

[22] ISO/IEC JTC 1/SC 29/WG 11 11172-4, "Information Technology — Coding of moving pictures and associated audio for digital storage media up to 1.5 Mbit/s — Part 4: Compliance Testing," *International Organization for Standardization*, 1995.

[23] ISO/IEC JTC 1/SC 29/WG 11 13818-3, "Information Technology — Generic Coding of Moving Pictures and Associated Audio: Audio," *International Organization for Standardization*, November 1994.

[24] ISO/IEC JTC 1/SC 29/WG 11 13818-4, "Information Technology — Generic Coding of Moving Pictures and Associated Audio: Conformance," *International Organization for Standardization*, 1996.

[25] P. Noll, "MPEG Digital Audio Coding," *IEEE Signal Processing Magazine*, pp. 59–81, September, 1997.

[26] M. Hall, J. Anderson, S. Amarasinghe, B. Murphy, S. Liao, E. Bugnion, M. Lam, "Maximizing multiprocessor performance with the SUIF compiler," *IEEE Computer* vol. 29, no. 12, pp. 84–89, Dec. 1996.

[27] D. Bacon, S. Graham and O. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, no. 4, pp. 345–420, Dec. 1994.

[28] S. Muchnick, *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997.

[29] *Workshop on Code generation for Embedded Processors* in *Design Automation for Embedded Systems*, vol. 4, no. 2-3, March 1999.