# Compiler-based Software Power Peak Elimination on Smart Card Systems

Matthias Grumer[1], Manuel Wendt[1], Christian Steger[1], Reinhold Weiß[1], Ulrich Neffe[2] and Andreas Mühlberger[2]

[1] Institute for Technical Informatics, Graz University of Technology
8010, Graz, Inffeldgasse 16, Austria
{grumer,wendt,steger,rweiss}@iti.tugraz.at
[2] NXP Semiconductors, Business Line Identification
8101, Gratkorn, Mikronweg 1, Austria
{ulrich.neffe,andreas.muehlberger}@nxp.com

**Abstract.** RF-powered smart cards are widely used in different application areas today. For smart cards not only performance is an important attribute, but also the power consumed by a given application. The power consumed is heavily depending on the software executed on the system. The power profile, especially the power peaks, of an executed application influence the system stability and security. Flattening the power profile can thus increase the stability and security of a system.
In this paper we present an optimization system that allows a reduction of power peaks based on a compiler optimization. The optimizations are done on different levels of the compiler. In the backend of the compiler we present new instruction scheduling algorithms. On the intermediate language level we propose the use of iterative compiling for reducing critical peaks.

**Keywords.** Software power optimization, compiler optimization, peak reduction.

## 1 Introduction

The complexity and functionality of smart cards is growing continuously. This results in a higher energy consumption of such devices. Smart cards are often supplied by a radio frequency (RF) field which provides a strictly limited amount of power. If the power consumed by such a device exceeds this limit a reset can be triggered by the power control unit or otherwise the chip may stay in an unpredictable state [1]. Furthermore the transmission from RF-system to a reader is often done via amplitude shift keying. Power peaks, which result in an unwanted modulation of the field, can potentially disturb the communication. Therefore the smart card has to be optimized for low power with the constraint to avoid peaks in power consumption. Mobile devices are often used to process and store confidential information. Simple power analysis (SPA) and differential power analysis (DPA) are attacks based on the analysis of the power consumption

Dagstuhl Seminar Proceedings 07041
Power-aware Computing Systems
http://drops.dagstuhl.de/opus/volltexte/2007/1103

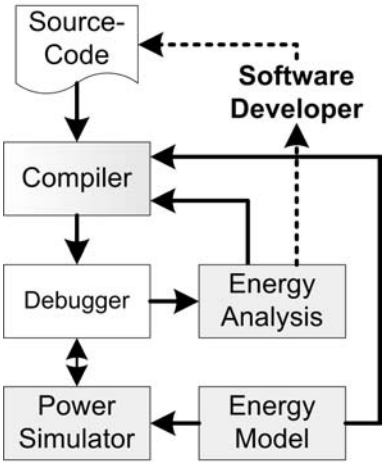


**Fig. 1.** Compiler loop for software power optimization.

profile of a smart card [2]. Reducing power peaks an thus flattening the power consumption profile can hinder these attacks.

To address these problems different solutions to reduce the power consumption at different system levels have been proposed. As power peaks are mainly caused by determined instruction sequences, in this work we focus on the software level. We present a new concept, where the optimization is done by the compiler. As depicted in Fig. 1, the source code is first compiled and then executed on an instruction set simulator. The simulator has an interface to a power model and can deliver a cycle accurate power profile of the executed code. The power model is derived from a previous characterization of the processor. The power profile is then processed by an energy analysis unit. The compiler uses the produced data to reduce and eliminate the power peaks. This cycle can be repeated in an iterative manner to find the optimal trade-off between performance and system stability and security.

The remainder of this paper is organized as follows. Section 2 surveys related work for software power optimization. In section 3 the optimization system is described. Section 4 depicts the optimization strategies on the different compiler levels. Results are presented in section 5. The conclusions are summarized in section 6.

## 2 Related Work

Tiwari et al. [3,4] outlined the importance of energy optimization at the software level in embedded systems already in the nineties. They presented different optimization techniques for reducing the software energy consumption. All these techniques are based on instruction level power analysis. The underlying energy

model defines base costs (BC) to characterize a single instruction. The circuit state overhead (CSO) describes circuit switching activity between two consecutive instructions.

Based on the energy model, different optimization strategies at the software level have been implemented. Tiwari et al. propose different compilation techniques for low energy software in [5]. They suggest the use of a code-generator-generator like IBURG [6], where the cost of each pattern is defined by the energy consumption. This technique, however, was not effective, as the authors observed that the energy-based and the cycle-based code generators produced very similar code.

In the same work the authors propose to reorder the instructions in such a way that the circuit switching activity is minimized. Therefore, the instructions are scheduled depending on the circuit state overhead. On a 486DX2 architecture this technique only led to a energy reduction of 2%.

Similar to the last technique, in [7] a novel list-scheduling algorithm for low-energy program execution is presented. The algorithm is based on a basic block approach. First a dependency table is derived for each block, then registers are renamed for reducing output dependencies. From the ready set of instructions, the instruction which minimizes the inter-instruction cost is selected to be scheduled. The results of this algorithm have shown a 4.54% decrease in the total energy dissipation. Coupled with other optimization strategies such as operand replacements, total energy savings of over 9% are achieved.

On a higher level of compilation a promising technique for the reduction of power peaks is iterative compiling. Iterative Compiling was first presented by Knijnenburg et al. [8,9]. They propose to generate many variants of source programs and to select the best one by profiling these variants. The main problem is to find the the best solution in the extremely large search space. They propose to randomly evaluate a small percentage of the transformation space.

While performance optimization is the main objective in research about iterative compiling, Gheorghita et al. use iterative compilation to reduce energy consumption [10].The authors use iterative compilation in order to find the best compiled code for energy and energy-delay product. However the work only concentrates on the loop transformation passes.

In this work we propose to use iterative compiling for the power peak reduction on all compiler passes influencing the power behavior of an application.

## 3 Optimization system overview

The whole optimization system is depicted in Fig. 2. The source code of an application is compiled to target code. As compiler the GNU Compiler Collection (GCC) is used. The target architecture is a MIPS32 4KSc processor. The target code is then executed via a debugger on an cycle accurate instruction set simulator. The simulator is directly attached to the energy analysis unit which, based on the energy model, calculates the energy consumption per instruction.

As presented in [11] our simulator achieves an accuracy of more than 95% for all benchmarks.

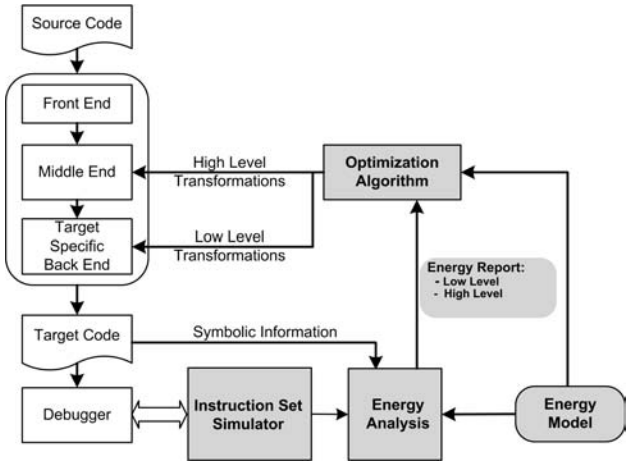


**Fig. 2.** Software energy optimization system overview.

Using symbolic information, the energy analysis unit abstracts these values and calculates the energy consumption at different levels. The resulting reports and the energy model are then used by the optimization unit to perform optimizations at the intermediate language levels.

### 3.1 Structure of GCC

The two intermediate languages under consideration in this work are the Register Transfer Language (RTL) and the GIMPLE language. Like most portable compilers, the compilation process of a GCC-based compiler can be conceptually split up in three phases:

- There is a separate front end for each supported language. A front end takes the source code, and does whatever is needed to translate that source code into a semantically equivalent, language independent abstract syntax tree (AST). The syntax and semantics of this AST are defined by the GIMPLE language, the highest level language independent intermediate representation GCC has.
- This AST is then run through a list of target independent code transformations that take care of such things as constructing a control flow graph, and optimizing the AST for optimizing compilations, lowering to non-strict RTL, and running RTL-based optimizations for optimizing compilations. The generation of the non-strict RTL-representation is performed based on the machine description of the target processor. The machine description

contains a pattern for each instruction that the target machine supports. These patterns are used to generate an non-strict RTL-list based on named instruction patterns. In this generation process, a pattern code and a unique id-number is assigned to each RTL-expression. The non-strict RTL is handed over to more low-level passes.
- The low-level passes are the passes that are part of the code generation process. The first job of these passes is to turn the non-strict RTL representation into strict RTL. Strict RTL-patterns fully match all operand constraints. Other jobs of the strict RTL-passes include scheduling, doing peephole optimizations, and emitting the assembly output. For further details on GCC see [12].

On the RTL level this work concentrates on the instruction scheduling pass. Originally, this pass looks for instructions whose output won't be available by the time it is used in subsequent instructions. It reorders instructions within a basic block to try to separate the definition and use of items that would otherwise cause pipeline stalls. This pass is used in this work to implement a power optimization scheme.

On the GIMPLE level the work is still ongoing. At the moment different passes on this level are analyzed to determine the influence on the power consumption of the system. In a next step optimizations on this level will be implemented.

## 4 Optimization strategies

In this section we present our optimizations implemented on the RTL-level and propose optimizations on the GIMPLE-level.

### 4.1 Optimizations on the RTL-Level

The system performs two optimizations at the RTL-level. Both optimizations are implemented in the instruction schedule pass of the compiler and are described in the following two sections.

**Instruction packing** As there are no CSO costs when two equal instructions are executed successively, this optimization simply tries to group equal instructions. For this purpose, the algorithm searches for expressions in the ready list, having the same pattern code as the last expression scheduled. If there is no expression with the same pattern code, normal scheduling is continued.

**Instruction Scheduling by CSO-Costs** A new greedy algorithm has been implemented, which schedules the RTL-expressions depending on their CSO-costs. This algorithm substitutes the original selection algorithm in the Haifa scheduler.

The algorithm requires double compilation for an application. In the first compilation cycle, the algorithm matches the unique id generated for each RTL-expression against the CSO-cost. This is necessary because the mapping of RTL-expressions to instructions is ambiguous. In a second compilation cycle, for every RTL-expression scheduled, the algorithm searches the RTL-expression with the lowest CSO-costs in the ready list. The algorithm schedules this RTL-statement, updates the ready list and identifies the next RTL-statement to be scheduled.

This algorithm doubles the compilation time. However application for embedded system tend to be relatively small and thus compile time is acceptable.

### 4.2 Optimizations on the GIMPLE-level

Work on this level is still in progress. We first analyzed the impact on the power consumption of the different optimization passes of this level. Table 1 summarizes the results of the benchmark "bubblesort".

| | Gain [%] | | | |
|---|---|---|---|---|
| Pass | Cycles | Energy | Std-dev | Mean Power |
| cse-skip-blocks | -77,59 | -76,81 | -23,65 | 3,46 |
| delayed-branch | -4,11 | -3,80 | 1,78 | 0,32 |
| gcse | -79,58 | -78,93 | -23,35 | 3,21 |
| no-delayed-branch | -75,51 | -75,13 | -15,20 | 1,57 |
| O1 | -79,54 | -78,90 | -23,32 | 3,15 |
| O1 no-loop-optimize | 0,37 | 0,33 | 0,15 | -0,04 |
| O1 no-tree-copy-rename | 19,11 | 17,87 | 7,69 | -1,04 |
| O1 no-tree-ch | 19,50 | 19,41 | 4,56 | -0,07 |
| O1 no-tree-dominator-opts | 9,56 | 8,18 | -0,74 | -1,26 |
| O1 schedule-insn | -0,19 | -0,09 | -0,22 | 0,09 |
| O1 schedule-insn2 | -0,19 | -0,09 | -0,52 | 0,10 |
| O2 | -79,58 | -78,14 | -22,80 | 7,06 |
| O2 no-gcse | 0,00 | 0,00 | -0,02 | 0,00 |
| O2 no-cse-skip-blocks | -0,01 | -3,57 | -1,17 | -3,56 |
| O2 no-schedule-insns2 | 0,00 | 0,00 | 0,31 | 0,00 |
| Os | -75,59 | -74,00 | -15,60 | 6,53 |

**Table 1.** Energyanalyse Bubblesort.

The results show clearly, that the total energy consumed is heavily depending on the execution time. Thus optimizations of the performance usually also influence the total energy consumption in a positive way. While the total energy consumption decreases, the mean power mostly increases. It can be deduced, that the power level is higher and thus resulting peaks are more critical for the system. The lower standard deviation is possibly caused by the higher mean power and not from peak reduction. Results of other benchmarks show, that a

certain pass can influence the power and energy consumption in different manners, depending on the application.

Based on these results two different optimization strategies can be proposed.

1. In an iterative compile process a optimal pass selection is found. As cost function different metrics such as standard deviation, mean power, total energy or the number of peaks can be used.
2. A peak detection system identifies critical parts of the code. The compiler then tries to modify these parts of the code. This can be achieved by selecting or deselecting different compiler passes on a basic block or function level in an iterative process. The resulting code is a trade-off between performance and system stability.

At the moment we concentrate on the second strategy proposed.

## 5 Experimental Results

For a first evaluation of the optimizations on RTL level, six evaluation programs, consisting mainly of algebraic and array functions were compiled. Table 2 shows the changes in mean value, total energy consumption and standard deviation by applying the instruction packing algorithm.

**Table 2.** Changes of the mean value, total energy consumption and standard deviation when applying instruction packing algorithm.

| Program | Mean value Gain [%] | Total energy Gain [%] | Standard deviation Gain [%] |
|---|---|---|---|
| C1 | 0.96 | 0.96 | 4.7 |
| C2 | 0.83 | 0.83 | 6.4 |
| C3 | 0.17 | 0.17 | 5.9 |
| C4 | 0.63 | -2.57 | 7.2 |
| C5 | 0.32 | 0.32 | 5.6 |
| C6 | 0.92 | 0.92 | 5.1 |

While the mean value and the total energy consumption is only reduced by up to 1 %, the standard deviation was reduced by up to 7%. It can be deduced, that the instruction packing does not reduce the energy consumption, but it produces a significantly smoother power profile. While the reduction of the mean value corresponds in all programs to the reduction of the total energy consumption, this is not the case in program “C4”. The reason is, that by reordering the instruction, the execution time rose, probaly due to additional pipline stalls, which also increase the total energy consumption.

Applying the instruction scheduling by CSO costs to the same programs delivers almost the same results. When taking a closer look at the produced

assembler files, it can be seen that both optimization algorithm produce quite the same code. Obviously, if there is the possibility to schedule the same instruction as the last scheduled, the scheduling by CSO behaves like the instruction packing, because there are no CSO cost between equal instructions.

All six programs are quite small and each of them fits into the cache. Thus no cache miss can occur. When concatenating the six programs and compiling them to one executable cache misses will occur. The results of this evaluation show only a reduction of the standard deviation of 3.4 % for the instruction packing and 1.9 % for instruction scheduling by CSO costs. The high power consumption of a memory access could be an explanation for this.

## 6 Conclusion

The minimization of power peaks in the power profile of mobile devices represents an important aspect for the system stability and system security. In this paper we presented a new approach to flatten the power profile of an application executed on a embedded processor. The results at the RTL-level have shown that optimizations at the software level can produce a flattened power profile. More promising is the optimization on the GIMPLE level of the GCC framework. We will use iterative compiling to reduce the power peaks of an application.

## 7 Acknowledgments

This work was funded by the Austrian Federal Ministry for Transport, Innovation, and Technology under the FFG contract FFG 810124

## References

1. Haid, J., Kargl, W., Leutgeb, T., Scheiblhofer, D.: Power management for rf-powered vs. battery-powered devices. In: Proceedings of Workshop on Wearable and Pervasive Computing. (2005)
2. Rothbart, K., Neffe, U., Steger, C., Weiss, R., Rieger, E., Muehlberger, A.: Power consumption profile analysis for security attack simulation in smart cards at high abstraction level. In: Embedded Software (EMSOFT 2005), 5th ACM International Conference on. (2005)
3. Tiwari, V., Malik, S., Wolfe, A.: Power analysis of embedded software: a first step towards software power minimization. In: ICCAD '94: Proceedings of the 1994 IEEE/ACM international conference on Computer-aided design, Los Alamitos, CA, USA, IEEE Computer Society Press (1994) 384–390
4. Tiwari, V., Malik, S., Wolfe, A., Lee, M.T.C.: Instruction level power analysis and optimization of software. J. VLSI Signal Process. Syst. **13** (1996) 223–238
5. Tiwari, V., Malik, S., Wolfe, A.: Compilation techniques for low energy: an overview. In: IEEE Symposium on Low Power Electronics. (1994) 38–39
6. Fraser, C.W., Hanson, D.R., Proebsting, T.A.: Engineering a simple, efficient code-generator generator. ACM Lett. Program. Lang. Syst. **1** (1992) 213–226

7. Sinevriotis, G., Stouraitis, T.: A novel list-scheduling algorithm for the low energy program execution. IEEE International Symposium on Circuits and Systems (2002) 97–100
8. Knijnenburg, P.M.W., Kisuki, T., O'Boyle, M.F.P.: Iterative compilation. (2002) 171–187
9. Kisuki, T., Knijnenburg, P.M.W., O'Boyle, M.F.P.: Combined selection of tile sizes and unroll factors using iterative compilation. In: PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques, Washington, DC, USA, IEEE Computer Society (2000) 237
10. Gheorghita, S., Corporaal, H., Basten, T.: Using iterative compilation to reduce energy consumption. In: ASCI 2004: Proceedings of the 10th Annual Conference of the Advanced School for Computing and Imaging, Delft, the Netherlands (2004) 197–202
11. Neffe, U., Rothbart, K., Steger, C., Weiss, R., Rieger, E., Muehlberger, A.: A flexible and accurate model of an instruction-set simulator for secure smart card software design. Lecture Notes in Computer Science **3254/2004** (2004) 491–500
12. Stallman, R.M.: GNU Compiler Collection Internals (GCC). GCC Developer Community, http://gcc.gnu.org. (2005)
13. Sami, M., Sciuto, D., Silvano, C., Zaccaria, V.: An instruction-level energy model for embedded vliw architectures. Computer-Aided Design of Integrated Circuits and Systems **21** (2002) 998–1010
14. Steinke, S., Knauer, M., Wehmeyer, L., Marwedel, P.: An accurate and fine grain instruction-level energy model supporting software optimizations (2001) In International Workshop on Power And Timing Modeling, Optimization and Simulation (PATMOS), September 2001.