Software Power Peak Reduction on Smart Card Systems based on Iterative Compiling

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

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

Abstract. RF-powered smart cards are widely used in different application areas today. The complexity and functionality of smart cards is growing continuously. This results in a higher power consumption. 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. If the power consumed by such a device exceeds the power provided by the RF-field a reset can be triggered by the power control unit or otherwise the chip may stay in an unpredictable state. Flattening the power profile can thus increase the stability of a system.
We present an optimization system which intends to eliminate critical peaks after the analysis of the power profile of an executed application. In an iterative compile process an optimal tradeoff between power and performance has to be found. This is achieved by selecting or deselecting different optimization passes on the intermediate language level of the compiler.

Key words: Iterative compiling - Software power optimization - Peak reduction - Smart card systems - Power profile analysis

1 Introduction

The complexity and functionality of smart cards is growing continuously. This results in a higher power 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. Smart cards 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 profile of a smart card [2]. Eliminating power peaks and thus flattening the power consumption profile can hinder these attacks.

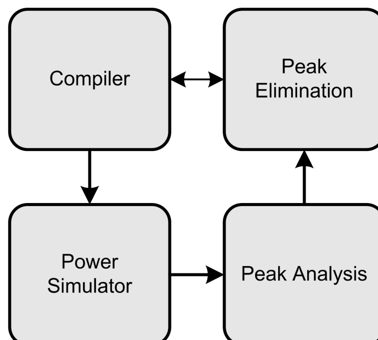

Fig. 1. Peak elimination framework - Overview.

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 in an iterative compile process. As depicted in Fig. 1, the source code is first compiled and then executed on a cycle accurate *power simulator*. The simulator delivers a cycle accurate power profile of the executed code. The *peak analysis unit* is able to identify critical peaks from the power profile and informs the *peak elimination unit* of the corresponding code segments. The *compiler* tries to eliminate these critical power peaks by selecting or deselecting the different compiler passes for these code segments. This whole cycle is repeated in an iterative manner to find the optimal trade-off between performance and system stability.

The remainder of this paper is organized as follows. Section 2 surveys related work for software power optimization and iterative compiling. In section 3 the classification of peaks is described. Section 4 depicts the peak elimination framework. 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, such as the use of a code-generator-generator and reordering the instructions . 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.

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. [5, 6]. 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 best solution in the extremely large search space. They propose to randomly evaluate a small percentage of the transformation space. Fursin et al. [7] demonstrated hill-climbing and random iterative search techniques to optimize large applications on a loop-level.

Cooper et al. [8–10] and later Kulkami et al. [11] demonstrated that finding optimal optimization order can also considerably improve code quality and performance.

Fursin and Cohen [12] presented an Interactive Compilation Interface (ICI). The main goals are to control only the decision process at global and local levels and to avoid revealing all intermediate compiler representations to allow further transparent compiler evolution and to treat current optimization heuristic as a black-box and progressively adapt it to a given program and given architecture. The interface supports different search strategies like exhaustive search, random search, hill-climbing search and machine learning. Although the interface should also support tuning programs for best power consumption, the authors have not shown this in any experiments.

While performance optimization is the main objective in research about iterative compiling, Gheorghita et al. use iterative compilation to reduce energy consumption [13].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 Peak Classification

Figure 2 (a) depicts a peak, which is critical for the system. The peak causes the voltage to drop under the threshold, under which the proper functionality of the processor can not anymore be guaranteed. In the depicted example this threshold is 1.6 V. Whether a peak is critical for the system depends on the shape of the peak himself and on the energy storage capacitor of the system.

Furthermore not only single peaks are critical for the system, but also sequences of peaks. Figure 2 (b) shows a sequence of two peaks. While neither the first peak nor the second one would be critical for the system if appearing alone, the sequence of the two is critical because the supply voltage has not enough time to recover.

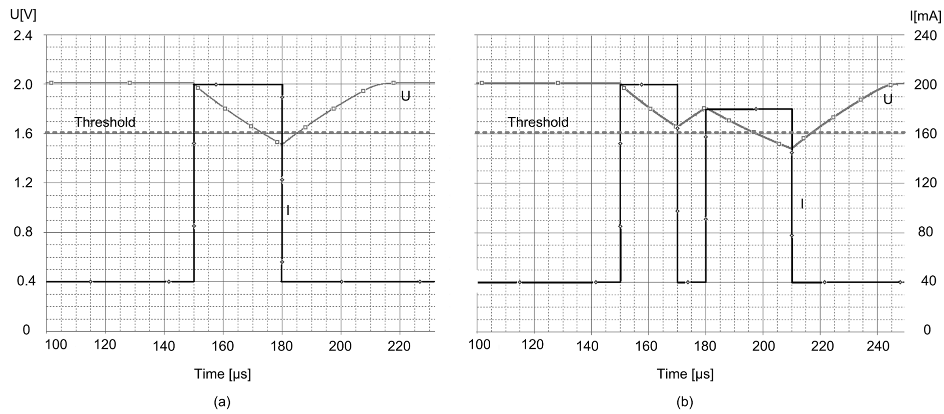

Fig. 2. Critical peaks for system stability: (a) single peak; (b) sequence of peaks.

Peaks can also be classified according to their origin. Peaks can be produced from both, hardware and software. Software-peaks arise from the execution of power intensive code segments. Hardware-peaks result for example from the usage of peripherals. Furthermore critical peaks can also arise from a combination of hardware and software-peaks.

	Single peak	Sequence of peaks
Software-peak	- Compiler optimization - Insertion of nonfunctional code	- Compiler optimization - Insertion of nonfunctional code
Hardware-peak	- No elimination possible	- Scheduling of peripheral activity
Software/Hardware-peak		- Scheduling of peripheral activity - Compiler optimization - Insertion of nonfunctional code

Table 1. Classes of peaks and counter measurements.

Table 1 summarizes the different classes of peaks and depicts possible counter measurements. Single or sequences of software-peaks may be eliminated by the insertion of nonfunctional code. Nonfunctional Code should always be selected in such a way, that circuit state overhead costs are minimized, e.g. an *ADD A,0* should be selected if the last instruction executed was an *ADD*. In the case of single peaks, this lowers the power level over the time and hinders the production of a critical peaks. In the case of sequences of peaks, the nonfunctional code in between two software peaks enables the recovery of the energy storage capacitor. Single hardware-peaks can not be eliminated from the software engineer's point of view. Sequences of hardware-peaks can be avoided by scheduling the periph-

eral activizy in such a manner that there is enough time for the energy storage capacitor to recover. The hardware/software-peaks allow a combination of the presented counter measurements. On software level the compiler optimization is also a good strategy to prevent critical peaks. This work concentrates on this optimization and presents a corresponding framework in the next section. In a later step the framework will also support the other methods presented for peak elimination.

4 Peak Elimination Framework

The whole framework is depicted in Fig. 3. The source code of an application is compiled to target code. As *compile*r 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 power simulator*, which delivers the power profile of the executed code. A *peak analysis unit* detects all critical peaks and generates a peak report. The *peak elimination unit* decides based on the peak report the level of optimization for each function of the application.

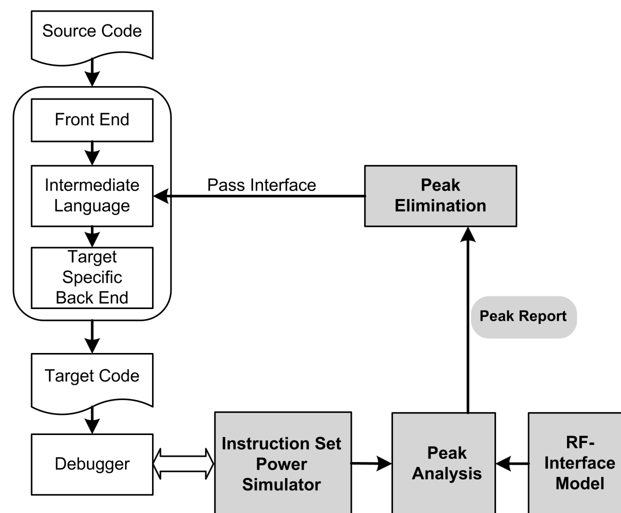

Fig. 3. Peak elimination framework.

The following sections describe first the power analysis of different compiler passes and then the peak elimination framework in more detail.

4.1 Power Analysis

First the impact on the power consumption of the different optimization passes of the compiler has been analyzed. For this purpose different benchmarks with

different optimization levels have been compiled by deselecting passes with the corresponding compiler flags. The resulting executables have been simulated on the power simulator.

Table 2 depicts the results for some passes of the benchmark *bubblesort*. The bold values are always referenced to the not optimized code (O0). The other values are referenced to the corresponding O-level, e.g. level O1 without the flag *-tree-ch* executes 19.5% slower then level O1 with *-tree-ch*.

	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 2. Analysis of *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 typically increases. It can be deduced, that the power level is higher and thus resulting peaks are more critical for the system. This fact is also depicted in Fig. 4. It clearly shows, that the power level increases with the level of optimization.

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. That is why it is not possible to make an optimal pass selection a priori.

Therefore we propose the following strategy: 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

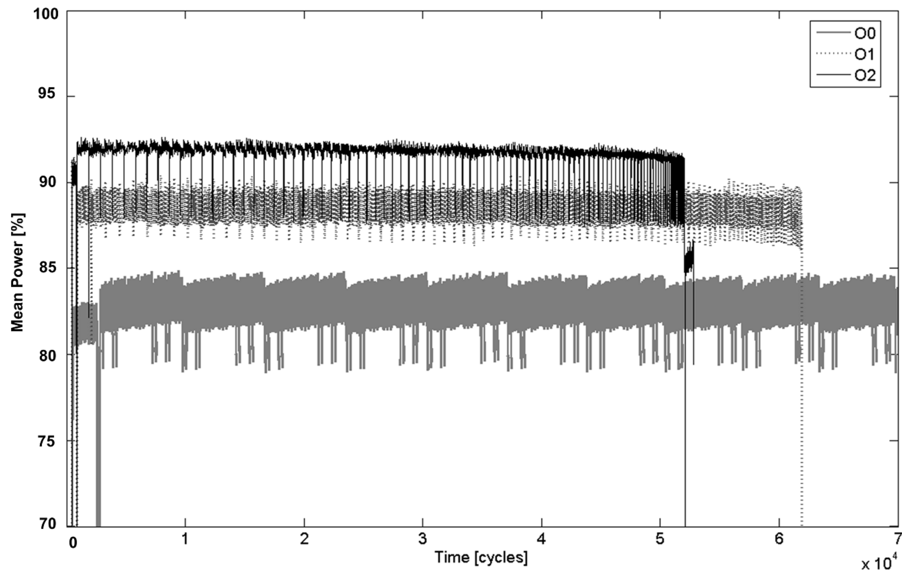

Fig. 4. Power profile segment of *bubblesort* with different optimization levels.

passes on a basic block or function level in an iterative process. The resulting code is a trade-off between performance and system stability.

4.2 Peak Elimination

Following the proposed strategy we have implemented a peak elimination framework. A preliminary compile cycle is necessary to get a first power profile. The power profile is produced from an instruction set simulator, which was enhanced by an energy model. The energy model developed is a flexible and accurate combination of an instruction-level energy model and a data dependent model based on Tiwari et al. [3] and was presented in [14]. The first compilation is done with the highest optimization level. The peak analysis unit delivers all code segments which produce a critical peak. At the moment we use the mean power windowed over a certain amount of cycles for this purpose. In a next step a model of the RF-interface, which can calculate the voltage level from a given current profile, will be integrated. As the pass manager of GCC works on a function level, in a first approach the framework works on this level as well.

The peak elimination unit deselects the highest pass of the compilation for all functions with critical peaks. An interface to the GCC pass manager, which allows an easy selection of passes, has been implemented for this purpose . We use an external file to communicate with the compiler. The file has an entry for each function of the application with the corresponding optimization level. During the compilation process, the pass manager reads from this file the passes to execute

for every function. After each compile cycle the resulting power profile is analyzed again and compared to the previous profile. If the peak size has increased, the deselected pass is selected again. Otherwise, if the peak has decreased but is still to high, the next pass is deselected. The main steps of the algorithm are the following. MaxPower stands for the maximum of the windowed mean power of each function:

1. *Compile with highest optimization level.*
2. *(While the **MaxPower** of any function > threshold) and (the lowest optimization level is not reached):*
   (a) *For all function with **MaxPower** > threshold:*
      i. *Deselect highest pass.*
   (b) *Recompile application.*
   (c) *If the new **MaxPower** > the old **MaxPower**:*
      i. *Select pass again.*

This whole loop is repeated either until there is no peak left or there is no pass remaining to be deselected. In the last case, the system reports that there are still critical peaks in the code, which can not be eliminated from the framework.

5 Experimental Results

For first experiments we have defined a threshold for the mean value over each segment of a defined amount of cycles of the power profile. The values for threshold and amount of cycles allow easily to define critical peaks, which have to be eliminated.

function	Cycles	Max. Mean Power [%]	Cycles	Max. Mean Power [%]
	bubblesort O0		bubblesort O1	
init	2166	94.36	330	103.97
sort	355290	96.47	52193	102.94
check	108	90.10	37	97.29
main	64	87.97	28	95.81
all	357628	96.47	52782	103.97
	bubblesort O1 no-tree-dominator-opts		bubblesort peak optimized	
init	421	102.84	734	94.36
sort	56280	99.14	56280	99.14
check	8	82.39	17	97.29
main	42	91.60	45	95.81
all	56751	102.24	57076	99.14

Table 3. Results of *bubblesort* for different optimization levels on function level.

Table 3 shows the results for selected optimization levels of the benchmark *bubblesort*. Column three of the table shows the highest mean value of the corresponding function over all cycle windows in per cent of the threshold.

While in *bubblesort O0* all functions are below the threshold, in *bubblesort O1* the function *sort* and *init* produce a critical peak. In *bubblesort O1 no-tree-dominator-opts* the function *sort* is again under the threshold, but *init* still remains above. The optimized configuration is thus composed of *init* with *O0*, *sort* with *O1 no-tree-dominator-opts* and *main* and *check* with *O1*. The peak optimized code was obtained after 13 compile cycles. The power profile of the peak optimized code is depicted in Fig.5. The whole power profile is below the threshold. As a tradeoff we loss 1.2% of the performace compared to the *O1*-optimized code, but still we only need 16% of the cycles compared to the *O0*-level.

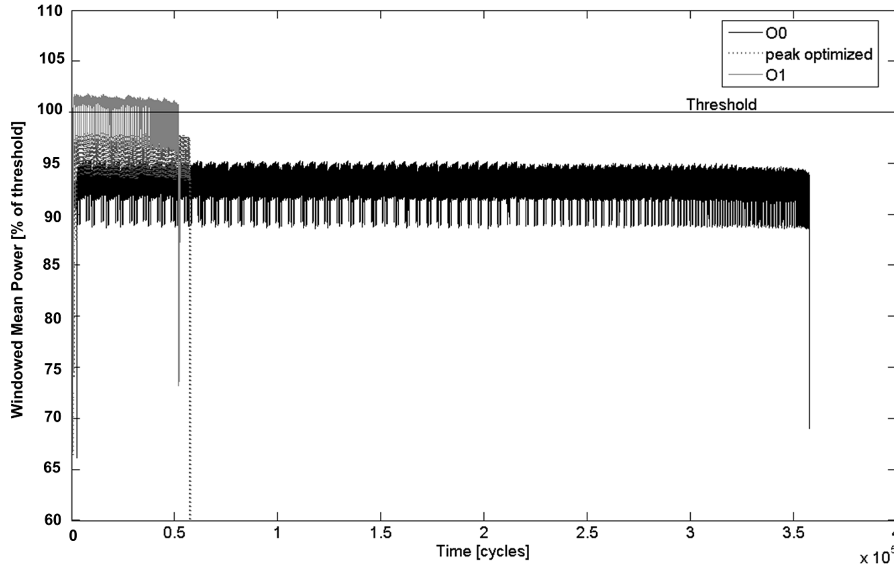

Fig. 5. Power profile with O0, O1 and the resulting code of the peak elimination framework for *bubblesort*.

For the benchmark *quicksort* the power profile is depicted in Fig.6. The first marked peak is not critical for the system caused by his shortness and does not have to be eliminated. The second marked region in the O2-profile contains several longer peaks, which are critical for the system. It was possible to reduce these peaks by lowering the optimization level of the corresponding functions. In this case we lose 3.9% of the performace compared to the *O2*-optimized code, but still we only need 47% of the cycles compared to the *O0*-level.

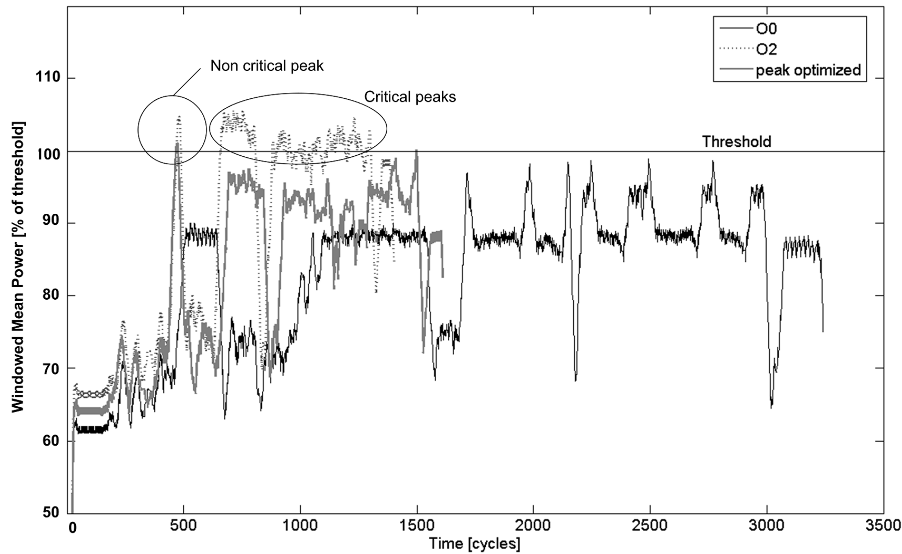

Fig. 6. Power profile with O0, O2 and the resulting code of the peak elimination framework for *quicksort*.

6 Conclusion

The elimination of power peaks in the power profile of smart cards represents an important aspect for better system stability. In this paper we presented a new approach to eliminate power peaks of the power profile of an application executed on an embedded processor. The results have shown that the compiler can be used to eliminate critical peaks. Iterative compiling can be used to find the optimal tradeoff between performance and system stability. To also allow for the elimination of hardware peaks, in a next step the framework will be enhanced by further peak elimination methods like presented in section 3.

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, Graz, Austria (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: EMSOFT '05: Proceedings of the 5th ACM international conference on Embedded software, Jersey City, NJ, USA, ACM Press (2005) 214–217
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, IEEE Computer Society Press (1994)
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**(2-3) (1996) 223–238
5. Knijnenburg, P.M.W., Kisuki, T., O'Boyle, M.F.P.: Iterative compilation. (2002) 171–187
6. 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
7. Fursin, G., O'Boyle, M., Knijnenburg, P.: Evaluating Iterative Compilation. (2002) 305–315
8. Cooper, K.D., Grosul, A., Harvey, T.J., Reeves, S., Subramanian, D., Torczon, L., Waterman, T.: ACME: adaptive compilation made efficient. In: LCTES '05: Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems, Chicago, Illinois, USA, ACM Press (2005) 69–77
9. Cooper, K.D., Schielke, P.J., Subramanian, D.: Optimizing for reduced code space using genetic algorithms. In: LCTES '99: Proceedings of the ACM SIGPLAN 1999 workshop on Languages, compilers, and tools for embedded systems, Atlanta, Georgia, United States, ACM Press (1999) 1–9
10. Cooper, K.D., Subramanian, D., Torczon, L.: Adaptive Optimizing Compilers for the 21st Century. J. Supercomput. **23**(1) (2002) 7–22
11. Kulkarni, P., Zhao, W., Moon, H., Cho, K., Whalley, D., Davidson, J., Bailey, M., Paek, Y., Gallivan, K.: Finding effective optimization phase sequences. SIGPLAN Not. **38**(7) (2003) 12–23
12. Fursin, G., Cohen, A.: Building a Practical Iterative Interactive Compiler. In: 1st Workshop on Statistical and Machine Learning Approaches Applied to Architectures and Compilation (SMART'07), colocated with HiPEAC 2007 conference, Gent, Belgium (January 2007)
13. 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
14. 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