Multi-Criteria Optimization of Hard Real-Time Systems*

Nicolas Roeser, Arno Luppold and Heiko Falk
Institute of Embedded Systems / Real-Time Systems
Ulm University
{nicolas.roeser|arno.luppold|heiko.falk}@uni-ulm.de

ABSTRACT

Modern embedded hard real-time systems often have to comply with several design constraints. On the one hand, the system's execution time has to be provably less than or equal to a given deadline. On the other hand, further constraints may be given with regard to maximum code size and energy consumption due to limited resources. We propose an idea for a compiler-based approach to automatically optimize embedded hard real-time systems with regard to multiple optimization criteria.

1. INTRODUCTION

In hard real-time systems, a task's Worst-Case Execution Time (WCET) must be lower than or equal to its deadline. However, many embedded systems have to comply with additional requirements. Minimizing energy consumption is an issue on mobile systems in order to save battery and allow for longer operation of devices. Additionally, embedded devices commonly have to cope with limited memory and therefore have to adhere to code and data size constraints.

Current optimizing compilers like GCC or LLVM mostly base their optimization techniques on heuristics. They cannot guarantee that the compiled and linked binary will adhere to all constraints. Therefore, the system developer manually has to analyze the resulting binary for compliance with the requirements. If one or more requirements are violated, the developer manually has to modify the compiler input and analyze the system's behaviour iteratively until all requirements are fulfilled. This may be a tedious approach, especially in cases with tight requirements on a given hardware architecture.

In order to ease system development, we provide a concept for an automated compiler-based multi-criteria optimization framework. If the compiler is unable to generate a sufficient solution, it will provide direct feedback, allowing the developer to remove functionality or upgrade hardware capabilities.

Although our approach is flexible enough to be adapted to different constraints, at least those which can be expressed by linear functions, we currently focus on WCET, code size and energy consumption, as we have identified them to be both academically challenging and practically relevant.

The key contributions of this paper are:

- We propose an idea on a compiler-based approach for a multi-criteria optimization framework.
- We illustrate our approach using existing compiler optimization techniques.
- We demonstrate its capabilities by applying it to a synthetic example.

This paper is organized as follows: section 2 gives a brief overview of related projects. Section 3 explains our underlying approach. Section 4 shows an example to illustrate this approach. This paper closes with a conclusion and a view on future challenges.

2. RELATED WORK

An optimization method trying to reduce the execution time and size of the code generated by a standard compiler is suggested in [9]: NSGA-II, a genetic algorithm for multi-criteria optimization, chooses which of GCC's optimizations should be enabled. Embedded systems are not targeted, so energy is not a criterion. The proposed iterative method needs several compilations; in contrast, non-heuristic algorithms can provide results in one optimization run.

Li and Malik [4] propose an ILP-based method for the estimation of a program's WCET. Their implicit path enumeration technique (IPET) uses flow constraints to model the program's control flow. The ILP objective function is then set to maximize the execution time while adhering to the flow constraints. While this approach is able to calculate tight and safe approximations for the WCET, the *maximization* objective prevents it from being used as basis for optimizations which try to *minimize* the WCET. Suhendra et al. [8] have solved this issue by implicitly summing up the execution times of basic blocks. Branches are modeled using multiple constraints, and the WCET can be calculated by trying to minimize one ILP variable which holds the execution time of the whole program. This approach will be further described in section 3.

Many embedded platforms feature a scratchpad memory (SPM), which is a relatively small but very fast and energy efficient memory. Steinke et al. [7] show in an example that with a scratchpad of 128 bytes, the total energy consumption of a system (CPU, main memory and scratchpad) can be reduced by two thirds.

Function specialization [6] (also known as function cloning or procedure cloning), as it is used in this paper, is a common basic compiler optimization. If a function is called with one or more known arguments, the compiler can add specialized versions of this function to the program text and replace the corresponding calls to them, which usually helps to reduce the program runtime, because some overhead at call time and inside the specialized functions can be saved.

*This work was partially supported by Deutsche Forschungsgesellschaft (DFG) under grant FA 1017/1-2.

3. APPROACH

Our approach combines and extends WCET-aware function specialization [5] and SPM allocation [3] to a multi-criteria optimization which optimizes for WCET, energy consumption and code size. SPM allocation affects both energy consumption and WCET with very small influence on the total code size. Function specialization increases code size and may decrease the WCET, but has a negligible effect on energy consumption. These properties are taken into account by the integer linear programming formulas described in the following sections. The ILP program decides which functions should be specialized and which ones are to be put into SPM in order to

- meet the system's timing deadline,
- fit the program into the available memory, and
- minimize the average-case energy consumption of the program.

At the current stage of our work, we only consider single-tasking systems running on memory without any caches. Functions specialized in more than one way are not considered, although adding support for that case to our formulas should be possible by interpreting such specializations as additional functions. We restrict the SPM allocation to function level. Therefore, each function may either be completely located in SPM or in Flash.

Prior to performing our optimization, safe approximations for WCET and average-case energy consumption have to be obtained for each function, both specialized and not, and whether it is located in SPM or in Flash memory. Additionally, the code size must be determined for both the specialized and the original version of each function. This analysis may be performed using tools like AbsInt aiT [1] for the system's WCET or by measurements for the energy consumption. The analysis itself is not part of our approach and will not be discussed any further.

An integer linear programming (ILP) problem is used to model the challenge and calculate an ideal solution. Energy consumption is usually a soft constraint, because hard upper bounds on a system's operational time cannot be easily predicted, so minimization of the average energy consumption is usually chosen as the multi-criteria optimization goal.

Throughout this paper, we use the notational conventions shown in table 1. Capital letters are used for constants in the ILP formulas, while small letters depict ILP variables.

3.1 ILP Description of the System

Suhendra et al. [8] introduce an ILP-based method for optimizing hard real-time systems. In order to model a program's control flow as set of integer linear constraints, the task is split into its basic blocks. A basic block is defined as a set of instructions that must be traversed from top to bottom without any jumps or branches. The control flow is then modeled by calculating the accumulated WCET w_B of a basic block B as the sum of the net execution time c_B of B and the accumulated execution time of its successor. This implicit summation stops at function borders. Therefore, for a block B that calls a function, $w_B = c_B + w_U + w_T$ (U being the succeeding block within B's surrounding function and T being the called function's entry block). Multiple successors can be modeled by using multiple constraints. The ILP's objective function is then defined to minimize the accumulated execution time of the task's entry basic block.

$\mathcal{F}$	set of all functions in the program after function specialization ($\mathcal{F}$ of course always includes the entry point, function `main`),
f	a function,
f_0	a specialized form of function f,
$N_{f,g}$	number of times function f calls function g,
S_f	code size of function f,
s_{Flash}	amount of non-SPM main memory (like flash EEPROM) needed by the program code,
s_{SPM}	amount of SPM needed by the program code,
E_f	energy consumption of function f,
D	deadline of the task,
W_f	accumulated WCET of function f,
r_f	binary decision variable for SPM usage of f,
p_f	binary decision variable for specialization of f.

Table 1: Notational conventions

For an illustration of how the control flow graph is transformed to enable this ILP-based analysis of the WCET, refer to our `bench` example program in section 4: the original control flow graph is shown in figure 1. Figure 2 depicts the final step of the graph transformation. The resulting graph is of course no longer suitable for generation of the program binary, it is solely used for WCET analysis.

3.2 WCET Optimization

ILP-based optimizations like [3] assign individual basic blocks to the SPM. Due to the linearity inherent to ILP and to reduce overall complexity, effects of platform-dependent details like timing costs for jump instructions to and from blocks on the SPM are highly overapproximated. The optimizations usually aim at leading the ILP solver in the right direction to minimize the overall WCET, but due to the overestimated timing penalties, the ILP program's estimation of the system's WCET will be a huge overapproximation. When optimizing for one single criterion, this problem is easily handled by trying to minimize the WCET, and afterwards performing a static WCET analysis to calculate tight WCET estimates. With multi-criteria optimizations, however, we may be using the WCET as a design constraint, and not as the primary optimization goal. As stated above, our approach focuses on minimizing the average-case energy consumption, while still meeting the system's timing constraints. Therefore, instead of minimizing for the WCET, we add an additional constraint to the ILP rules to define an upper bound for the system's maximum execution time:

$$w^*_{\texttt{main}} \leq D \tag{1}$$

$w^*_{\texttt{main}}$ denotes the system's accumulated WCET in the ILP formulation, D is defined as the system's deadline. To avoid the aforementioned overestimation of the system's WCET, we limit the SPM optimization to whole functions. We define a binary decision variable r_f which is set to 1 if function f is located on SPM, and 0 else. We use aiT to calculate each function's WCET $W_{f,\text{SPM}}$ if the function is located in SPM, and $W_{f,\text{Flash}}$ if the function is located in flash memory. These costs contain the execution time of the function f without the costs of any functions which are called by f. This way, we mostly avoid overestimations of the WCET in the ILP formulations and are not bound to the WCET minimization as the ILP's objective function.

Function specialization is handled by introducing another binary decision variable into the ILP problem. We define p_f to be 1 if the function is to be specialized, and 0 else. With this in mind, the accumulated execution time of function f may be written as

$$w_f^* \geq N_{f,f} \left[W_{f,\text{SPM}} \cdot r_f + W_{f,\text{Flash}} \cdot (1 - r_f) \right] + \sum_{g \in \mathcal{C}_f} N_{f,g} \cdot w_g^* \quad (2)$$

for all calls to the unspecialized, original function f, and as

$$\begin{aligned} w_{f_0}^* \geq & N_{f,f} \left[W_{f,\text{SPM}} \cdot r_f \left(1 - p_f\right) + \right. \\ & W_{f,\text{Flash}} \cdot \left(1 - r_f\right) \left(1 - p_f\right) + \\ & \left. W_{f_0,\text{SPM}} \cdot r_f p_f + W_{f_0,\text{Flash}} \cdot \left(1 - r_f\right) p_f \right] + \\ & \sum_{g \in \mathcal{C}_f} N_{f,g} \cdot w_g^* \end{aligned} \quad (3)$$

for calls to a potentially specialized version f_0. In both cases, $\mathcal{C}_f$ is defined as the set of functions which are called by f, and is determined by the compiler. If a function g is called in its unspecialized and in a potentially specialized version, g and g_0 are added to $\mathcal{C}_f$. $N_{f,f}$ is 1 for non-recursive functions, otherwise it denotes the maximum recursion depth, as determined by the writer of the task, and e.g. annotated in the source code. If the control flow of a function is split into several branches which call different functions, we use several inequations, one for each possible path. Though equation (3) seems to be of quadratic complexity, the multiplication of binary ILP variables equals a logical AND operation and can easily be linearized, as shown in [2]. $w_{\texttt{main}}^*$ can be used as safe overapproximation of the WCET of the whole task.

3.3 Energy Optimization

To calculate and minimize the average energy consumption of the system, we need detailed knowledge about the execution count of each function. These information may be obtained from simulation or measurements. The maximum energy savings are not necessarily achieved by optimizing the functions which need most energy. Instead, the product of a function's energy consumption E_f and the function's estimated execution count over the operation time of the system needs to be taken into account.

The energy consumption of a function heavily depends on whether the function is located in SPM or not. We assume that energy profiling data is available for both cases. Just as for the WCET formulation, E_f does not include the energy consumed by functions which f calls. The total energy consumption by function f, including all called functions, is given by a formula along the lines of the one presented as equation (3). Again, we directly include the distinction whether a specialized function is called or not.

$$\begin{aligned} e_f^* \geq & N_{f,f} \cdot E_f \\ & + \underbrace{\sum_{g \in \mathcal{F}} N_{f,g} \cdot e_g^*}_{\text{if the call to } g \text{ can not use a specialized form}} \\ & + \underbrace{\sum_{g \in \mathcal{F}} N_{f,g} \cdot \left(e_{g_0}^* \cdot p_g + e_g^* \cdot (1 - p_g) \right)}_{\text{if the call to } g \text{ can use the specialized form } g_0} \end{aligned} \quad (4)$$

With this formula, $e_{\texttt{main}}^*$ is an overapproximation of the system's typical energy consumption.

3.4 Code Size

Choosing to specialize a function increases the overall code size of the program. For each function, we a priori determine the effect on the code size if this function is chosen to be specialized.

In order to make our integer linear program aware of how much SPM and main memory the code takes, we add ILP constraints to model the code size:

$$s_{\text{Flash}} \geq \sum_{f \in \mathcal{F}} S_f \cdot (1 - r_f) + S_{f_0} \cdot p_f \cdot (1 - r_{f_0}) \quad (5)$$

$$s_{\text{SPM}} \geq \sum_{f \in \mathcal{F}} S_f \cdot r_f + S_{f_0} \cdot p_f \cdot r_{f_0} \quad (6)$$

Again, linearization of the ILP formulas has to be performed, cf. equation (3).

Additional constraints can be added to limit the maximum amount of memory that may be used by the program:

$$s_{\text{Flash}} \leq M_{\text{Flash}} \quad (7)$$

$$s_{\text{SPM}} \leq M_{\text{SPM}} \quad (8)$$

With M_{Flash} and M_{SPM} being the physically available SPM and Flash memory.

Finally, the ILP's objective function may be chosen. As stated above, usually the energy consumption is chosen to be minimized:

$$\min \left(e_{\texttt{main}}^* \right) \quad (9)$$

4. EXAMPLE

We will illustrate our approach with a hypothetical benchmark `bench` which is to be run on an embedded microprocessor equipped with 200 bytes of scratchpad memory. Execution of the `bench` task starts with function `main`. This function either calls function `f` twice, or function `g` twice; which function is actually called depends on some external data. If functions are specialized, one call of the two calls to `f`, or `g`, respectively, can use the specialized function. Figure 1 shows the control flow with the basic blocks of the unmodified function `main` in the example program.

Analysis of the `bench` task as described in section 3.2 results in the data given in table 2 as input for our algorithm.

f	`main`	`f`	`g`	`f0`	`g0`
S	100	50	100	50	100
For function in main memory:					
W	50	50	40	30	9
E	50	50	60	50	50
For function in scratchpad memory:					
W	10	10	9	5	5
E	20	20	30	20	20

Table 2: Net WCET and energy values and sizes per function

We assume that collecting profile data over several thousand executions of the task reveals that the two branches in function `main` have very different execution probabilities: the branch calling `f` is executed in 10 out of 1010 cases, the path over the calls to function `g` is executed 1000 out of 1010 times. The probability distribution *non-specialized*:*specialized* for which form of the function is used is 10:10 for `f` and

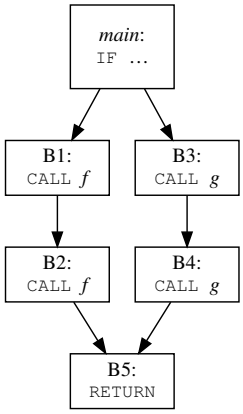

Figure 1: Control flow graph showing the `main` function of the `bench` task and its basic blocks.

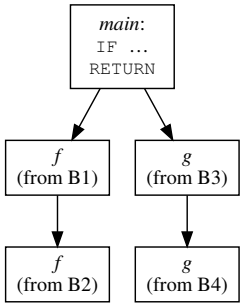

Figure 2: WCET analysis graph for `main`, i.e., the entire task, generated from the original CFG.

200:1800 for `g`. The collected profiling data has no impact on the WCET of the task, which is 150 cycles if no optimizations are applied.

An optimization which solely targets minimization of the WCET will specialize `g` and put `main` and `f`, but not `g` or `g0` (the specialized version of function `g`), into the SPM, which results in a WCET of 59 cycles.

The best solution when optimizing only for energy is to specialize `g` and `f` and put functions `main` and `g0` into SPM. The WCET for the example program optimized in this way is 90 cycles. For comparison, the best WCET we could get if the system had no SPM at all would be 130 cycles.

When applying our multi-criteria optimization algorithm explained in section 3 to the example program, we will now assume the deadline of 70 cycles, until which the task must have completed its work. Table 3 shows the optimization results which are achieved by applying single optimizations or the multi-criteria algorithm with the policy to minimize overall energy consumption. In the case of more than one possible solution which is considered equal by the relevant optimization, the figures in this table have been chosen to show the solution which is also better in terms of the other possible optimization criteria.

Target	WCET	s_{SPM}	s_{Flash}	Energy
unoptimized	150	0	250	85,750
WCET	59	150	200	61,300
Energy	90	200	200	34,600
Multi-criteria	70	150	100	55,450

Table 3: Results of optimizing the `bench` program for different optimization targets (WCET in cycles; energy consumption averaged over 505 executions)

As can be seen, if the WCET is selected as the only optimization target, optimization will produce a program binary that satisfies the deadline, but wastes a lot of energy.

If the system's energy consumption is selected as optimization objective, the execution path which is taken most of the time is optimized, and code size is increased, but the compilation result is still worthless, as it cannot be guaranteed to adhere to the deadline.

If `bench` is not optimized for execution time, code size, and energy consumption at the same time, we get unsatisfying results. Only a multi-criteria optimization like the algorithm presented by us above allows us to create a program which stays within all our optimization constraints.

5. CONCLUSION

We have provided a concept for multi-criteria oriented compiler optimizations which respect both multiple hard constraints and an additional optimization target without any hard upper bound. By means of a synthetical example, we have illustrated the potential benefits of our concept.

In the future, we are going to implement our approach into an existing compiler framework and evaluate its capabilities using real-world benchmarks. We aim at constructing a generic interface for multi-criteria compiler optimizations in the long run which can make use of arbitrary code optimizations. A further challenge is the extension of multi-criteria optimization to multi-tasking systems.

6. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. aiT Worst-Case Execution Time Analyzers. *Available online at `http://www.absint.com/ait/`*, Aug 2014.

[2] J. Bisschop. *AIMMS. Optimization Modeling*. Paragon Decision Technology, Haarlem, The Netherlands, AIMMS 3 edition, June 2009.

[3] H. Falk and J. C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proceedings of the 46th Design Automation Conference*, DAC '09, pages 732–737, New York, NY, USA, 2009. ACM.

[4] Y.-T. S. Li and S. Malik. Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32nd Design Automation Conference*, DAC '95, pages 456–461, New York, NY, USA, 1995. ACM.

[5] P. Lokuciejewski, H. Falk, M. Schwarzer, P. Marwedel, and H. Theiling. Influence of Procedure Cloning on WCET Prediction. In *Proceedings of the 5th International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '07, pages 137–142, New York, NY, USA, 2007. ACM.

[6] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann Publishers, 2011.

[7] S. Steinke, L. Wehmeyer, B.-S. Lee, and P. Marwedel. Assigning Program and Data Objects to Scratchpad for Energy Reduction. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, DATE 2002, pages 409–415, 2002.

[8] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the 26th Real-Time Systems Symposium*, RTSS 2005, pages 223–232, 2005.

[9] Y.-Q. Zhou and N.-W. Lin. A Study on Optimizing Execution Time and Code Size in Iterative Compilation. In *Proceedings of the 3rd International Conference on Innovations in Bio-Inspired Computing and Applications*, IBICA 2012, pages 104–109, 2012.