Of course, this constraint is not added to the ILP but depicted for the sake of completeness.

2. Blocks b_i and b_j are contiguous (fall-through edge) and jumping to b_k results in backward displacement (predicted taken). If the fall-through edge is visited although the jump was predicted taken, two penalty cycles are the result:

$$jp^{i,j}_{cond32} \geq 2 * (b_i \circ b_j \wedge x_i > x_k) \tag{11}$$

3. Blocks b_i and b_k are contiguous and jumping to b_j (pass-through edge) results in forward displacement (predicted not taken). Since the fall-through target b_k is predicted to be executed, jumping to b_j would result in two cycles penalty:

$$jp^{i,j}_{cond32} \geq 2 * (b_i \circ b_k \wedge x_i < x_j) \tag{12}$$

4. Blocks b_i and b_k are contiguous and jumping to b_j (pass-through edge) results in backward displacement (predicted taken). If jumping to b_j is predicted correctly, only one cycle penalty has to be added:

$$jp^{i,j}_{cond32} \geq 1 * (b_i \circ b_k \wedge x_i > x_j) \tag{13}$$

Depending on the jump scenario (*JS*) of a basic block b_i, the overall jump penalty jp_i is defined as follows:

$$jp_i = \begin{cases} jp^i_{impl} & \text{if JS of } b_i \text{ is } implicit \text{ or} \\ & \text{initially } unconditional \\ jp^{i,j}_{cond16} & \text{if JS of } b_i \text{ is } conditional \text{ 16 bit} \\ jp^{i,j}_{cond32} & \text{if JS of } b_i \text{ is } conditional \text{ 32 bit} \\ 0 & \text{else} \end{cases} \tag{14}$$

The jump penalties are used to extend the basic control flow constraints defined in Equations 2 and 3:

$$w^L_{exit} = C^L_{exit} + jp^L_{exit} \tag{15}$$

$$\begin{aligned} \forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{succ}) \in E : \\ w_i \geq w_{succ} + C_i + jp_i \end{aligned} \tag{16}$$

3.2.4 ILP Model of the Global Control Flow

Up to this point, Equations 1 - 16 only model the intra-procedural control flow of a single function F within the ILP. Without loss of generality, we assume one dedicated entry block b^F_{entry} as first block of F. For b^F_{entry}, the ILP variable w^F_{entry} denotes the WCET of any path starting at b^F_{entry} for a single execution of F.

However, some basic block b_i of a function F' may contain a call to function F. In this situation, F's WCET represented by variable w^F_{entry} has to be added to the WCET of block b_i. Thus, the control flow constraint in Equation 16 is extended by w^F_{entry}, representing F's WCET, if block b_i calls F:

$$\begin{aligned} \forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{succ}) \in E : \\ w_i \geq w_{succ} + C_i + jp_{i,succ} + w^F_{entry} \end{aligned} \tag{17}$$

3.2.5 Objective Function

The overall goal of our ILP is to minimize a program's WCET by rearranging the order of basic blocks inside a function. Due to the nature of Equations 16 and 17, variable w^F_{entry} corresponds to the WCET of function F including the WCETs of all functions called by F extended by possible jump penalties. Function `main` is the unique entry point of an entire program; hence, variable $w^{\texttt{main}}_{entry}$ denotes the overall WCET of the program. As a consequence, the value of this variable has to be minimized by the ILP:

$$w^{\texttt{main}}_{entry} \rightsquigarrow min. \tag{18}$$

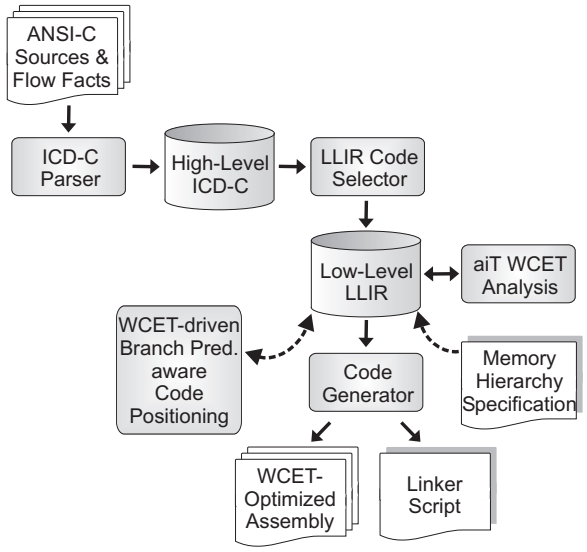

Figure 3: Workflow of the WCET-aware C compiler WCC

4. WORKFLOW

WCET-driven optimizations and especially branch-prediction-aware code positioning need support of an underlying compiler to collect WCET data and to perform the required memory layout modifications. We employ the WCET-aware C compiler framework, called *WCC* [9], which is intended to assist the development of various high- and low-level WCET-driven optimizations. It is a compiler targeted at Infineon's TriCore TC1796 processor coupling AbsInt's static WCET analyzer *aiT* [1] which provides WCET_{est} data that is imported into the compiler backend and made accessible for optimizations.

Figure 3 depicts WCC's internal structure. One or more files of a program are read in the form of ANSI-C source files with user annotations for loop bounds and recursion depths, called *flow facts*. These source files are parsed and transformed into the high-level intermediate representation (*IR*) called *ICD-C* [19]. At this level, the compiler frontend provides several standard compiler optimizations focussing on ACET minimization.

In the next step, the *LLIR Code Selector* translates the high-level IR into a low-level IR called *ICD-LLIR* [5]. Again, several standard compiler optimization can be performed – now on this TC1796-specific low-level IR. One of these optimizations is the proposed branch prediction aware code positioning which is performed as second last. As very last one, the automatic jump correction is performed (cf. Section 3) which corrects the control flow by inserting and removing unconditional jumps as well as negating test conditions.

To enable such a WCET-aware optimization, *aiT* is employed to perform static WCET analyses on the low-level IR. Mandatory information about loop bounds (among others required as constant $Count^L_{max}$ in Equation 4) and recursion depths is supplied by flow

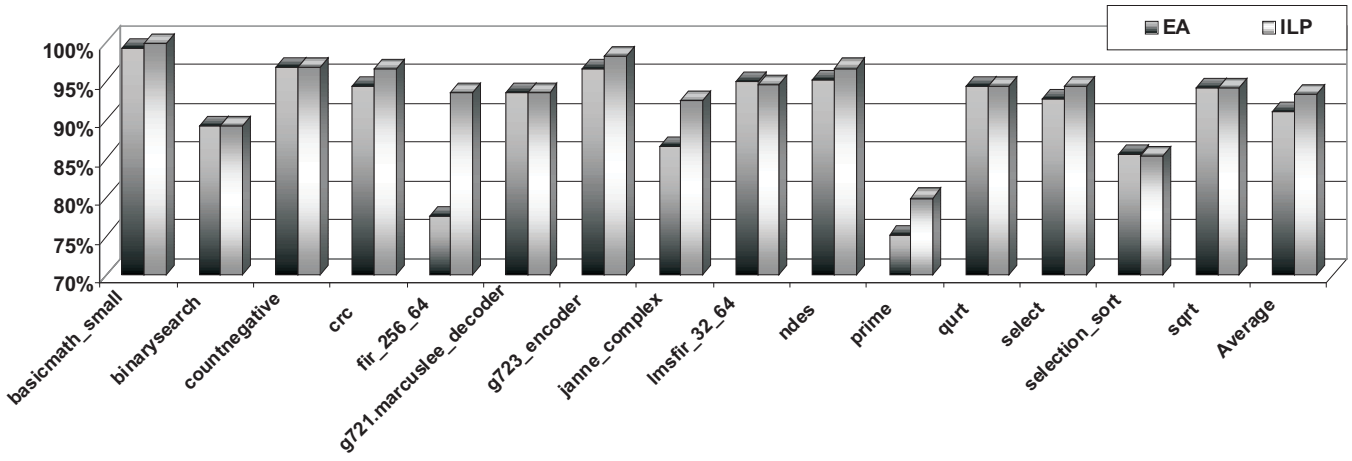

Figure 4: Relative WCETs after Code Positioning Optimizations

fact annotations. These flow facts are automatically translated from the high-level IR to the low-level IR and are always kept valid and consistent during each optimization and transformation step of the compiler.

Finally, WCC emits WCET-optimized assembly files and its own linker script in order to generate the optimized binary.

5. EVALUATION

This section evaluates the performance of our WCET-driven branch prediction aware code positioning algorithms applied to real-life benchmarks. In Section 5.1, the experimental environment which is employed to perform evaluations is presented. Section 5.2 discusses the WCET reductions achieved by the code positioning algorithms described in Section 3, whereas Section 5.3 discusses the achieved ACET reductions. Finally, Section 5.4 deals with the computational complexity of the proposed optimization.

5.1 Experimental Environment

For benchmarking, we used the optimization level *O3* for which the WCC compiler (cf. Figure 3) applies 42 different optimizations in order to evaluate the performance of our new algorithms on highly optimized code. 23 of these optimizations, comprising various loop optimizations, are performed on the high-level IR ICD-C. The remaining optimizations are performed on the low-level IR ICD-LLIR. The compiler supports the Infineon TriCore TC1796 processor which integrates a 2 MB program Flash as main memory and 48 kB scratchpad as a fast, tightly coupled memory (*SPM*). The TriCore v1.3 family is equipped with a static branch prediction unit for which equations 7 - 16 are tailored.

For our measurements, we used 15 benchmarks stemming from the benchmark suites *Mediabench* [16], *MiBench* [13], *MRTC* [12] and *UTDSP* [22]. The number of benchmarks was limited since the evolutionary optimization algorithm would otherwise require several weeks of optimization runtime as discussed in Section 5.4. The code size of the benchmarks ranges from 1.7 kB (*fir_256_64*) up to 15 kB for the *basicmath* benchmark.

Today's embedded systems are equipped with main memories in megabyte ranges. Nevertheless, all evaluations are performed with the program code residing in the fast SPM in order to avoid undesired side-effects by hardly predictable access latencies of FLASH memories due to their page buffers. This enables comparable results for both optimization algorithms. Otherwise, the evolutionary algorithm would have a slight advantage since the repetitive creation and evaluation of individuals implicitly considers memory hierarchy effects by repetitive WCET analyses. In contrast, these effects are very difficult to express adequately within an ILP-based optimization.

In order to evaluate the achievable WCET reduction, the evolutionary approach is invoked with an initial population of $\alpha = 20$ individuals. Each offspring generation has $\mu = 20$ parents and also comprises $\lambda = 20$ individuals. The maximum number of generations amounts to $maxGen = 20$.

For solving the ILP-model generated by the algorithm in Section 3.2, *IBM ILOG CPLEX* [15] is employed which is a sophisticated solver for integer programming problems.

5.2 WCET Estimations

Figure 4 depicts the results achieved by our branch prediction aware code positioning algorithms for the considered 15 benchmarks. For each benchmark, the left bar represents the result for by the evolutionary code positioning technique, whereas the right bar represents the result if the ILP-based algorithm is applied. The 100% line is equal to the estimated WCET of the benchmarks compiled with the optimization level O3 without code positioning (the code layout mainly matches the logical order found in the *C* source code). The bars depict the WCET_{est} of the optimized program computed by the static WCET analyzer as percentage of its "unoptimized" version.

The evolutionary algorithm reduces the WCET_{est} of the benchmarks by up to 24.7% (*prime* benchmark). For the same benchmark, the ILP-based algorithm is able to achieve WCET_{est} reductions of up to 20.0%. Significant WCET_{est} reductions could be achieved for almost all benchmarks except for *basicmath_small*. Here, the EA achieved only a marginal WCET_{est} reduction of 0.6% whereas the ILP-bases approach did not achieve any improvement. On average, we were able to reduce the WCET_{est} for all benchmarks by 8.9% by applying the evolutionary approach and by 6.7% for the ILP-based optimization, respectively.

Table 2 shows the ratio of executed unconditional (columns labeled "Uncond.") and mispredicted conditional jump instructions (columns labeled "Mispred.") on the WCEP. For each benchmark, values for the unoptimized version, the evolutionary and the ILP-based approach were collected. Here, 100% equals the overall number of executed jump instructions on the WCEP (conditional

Table 2: Ratio of unconditional and mispredicted jumps

	Unoptimized			EA			ILP		
	#Jumps	*Uncond.*	*Mispred.*	*#Jumps*	*Uncond.*	*Mispred.*	*#Jumps*	*Uncond.*	*Mispred.*
basicmath_small	76691	0.0%	0.5%	77407	0.9%	0.6%	76691	0.0%	0.5%
binarysearch	15	0.0%	66.7%	14	14.3%	35.7%	17	29.4%	29.4%
countnegative	1240	32.3%	0.1%	1041	19.3%	0.1%	840	0.0%	0.1%
crc	8878	23.1%	24.0%	8892	23.2%	24.0%	9388	27.3%	16.4%
fir_256_64	768	33.2%	0.1%	517	0.8%	49.3%	514	0.2%	0.2%
g721.marcuslee_decoder	14442	0.0%	50.0%	21664	33.3%	0.0%	16849	14.3%	14.3%
g723_encoder	95980	2.3%	23.5%	96133	2.5%	23.8%	141217	35.0%	9.2%
janne_complex	70	5.7%	47.1%	75	12.0%	22.7%	71	7.0%	16.9%
lmsfir_32_64	14802	17.0%	24.5%	14962	17.9%	17.0%	14931	17.7%	1.1%
ndes	4369	0.1%	74.3%	6007	27.3%	41.6%	6617	34.0%	33.3%
prime	865	0.0%	49.7%	865	0.0%	0.2%	866	0.0%	0.1%
qurt	240	0.0%	27.5%	246	2.4%	1.2%	277	13.4%	13.4%
select	969	0.0%	1.8%	970	0.1%	1.8%	1004	3.5%	1.8%
selection_sort	341160	0.0%	74.9%	433315	21.3%	18.9%	425300	20.0%	19.8%
sqrt	547	0.0%	24.3%	547	0.0%	0.0%	620	11.8%	11.8%
Average	37402	7.6%	32.6%	44177	11.7%	15.8%	46347	14.2%	11.2%

and unconditional) in columns labeled with "#Jumps". Multiple executions of a jump are counted multiple times.

Considering the amount of mispredicted branches, both algorithms perform best for the *selection_sort* benchmark where reductions of 56% (EA) and 55.1% (ILP) were achieved. Conversely, unconditional jumps had to be inserted in order to correct the control flow. They amount to 21.3% (EA) and 20% (ILP) of the overall executed number of jumps. Thereby, WCET_{est} reductions of 14.4% and 14.6% were achieved, respectively.

The amount of unconditional jumps could be decreased by 32.4% and 33% for the benchmark *fir_256_64* by applying the evolutionary and the ILP-based approach, respectively. Even though considerable reductions can be achieved, in the majority of cases, the number of executed unconditional jumps was increased by up to 33.3% (*g721.marcuslee_decoder, EA)* and 33.9% (*ndes*, ILP).

On average, the number of mispredicted branches was reduced by 16.8% (EA) and 21.4% (ILP) whereas the unconditional jumps were increased by 4.1% (EA) and 6.7% (ILP).

Corner cases

For the benchmark with the highest WCET_{est} reductions (*prime*), it turned out that both reordering algorithms were able to eliminate almost all mispredicted branches. No unconditional jumps are executed at all. Nevertheless, the EA algorithm was able to outperform the ILP-based approach by 4.7% w. r. t. WCET_{est} reduction.

This behavior is caused by the impact of the resulting memory layout of the modified program on the WCET_{est}: It is, for instance, worthwhile to align loop headers at the beginning of memory lines. Otherwise, the crossing of memory lines is more frequent. This often results in a decreased performance since multiple memory lines have to be fetched in order to execute the loop header. The ILP is not aware of any memory addresses during the rearranging of basic blocks with the result that line crossing effects can not be modeled. In contrast, the evolutionary approach always implicitly takes the impact of memory layout modifications on the WCET_{est} into account by evaluating the fitness of each newly created individual using *aiT*.

For the *fir_256_64* benchmark, both algorithms were able to remove almost all executed conditional jump instructions (0.8% respectively 0.2% remaining). But the evolutionary algorithms was able to outperform the ILP optimization by 16% WCET_{est} reduction (77.7% vs. 93.7% resulting WCET_{est}) although the number of mispredicted branches is increased by 33%. The ILP generates a basic block order where 99.6% of the conditional branches are correctly predicted taken, each resulting in one cycle pipeline stall (cf. Table 1). The EA, however, determines a memory layout where half of the branches are correctly predicted not taken resulting in no penalty cycles. The remaining conditional jumps were mispredicted, each resulting in two cycles penalty. Since the executed jumps induce the same amount of overall pipeline stall cycles, the evolutionary algorithm only performs better due to a memory layout causing less line crossings of instructions.

Although the evolutionary approach should have a small advantage compared to the ILP-based optimization, there are cases, for instance the benchmark *select*, where the ILP-based approach performs better (in finite time): The ILP optimization explicitly models absolute positions of basic blocks and the influence of possible branch penalties on the WCET. The corresponding constraints are always considered during solving the equations. Thus, the complexity only depends on the number of constraints which in turn depend on the control flow of a program. Hence, the overhead for solving the ILP is independent of the number of basic blocks which have to be moved.

In contrast to the ILP-model, the evolutionary approach can only apply small modifications to the order of basic blocks of a function by employing the operators crossover and mutation when a new offspring individual is created. Thus, for benchmarks with complex control flows with a lot of control flow edges, a large number of offspring individuals has to be created until the space of valid solutions can be explored extensively in order to find an improved solution. The *selection_sort* benchmark is an example where the optimal result is typically found after 40 generations, leading to up to 800 WCET analyses.

Such an insufficiency could be tackled by tuning the evolution parameters; for instance, each position of the solution vector could be mutated with a certain probability instead of mutating only a single position. In this paper, the evolutionary approach only serves as a case study if reordering basic blocks w. r. t. the WCET of a program pays off. Another aim is to obtain WCET reference data for a comparison with the ILP-based approach. Thus, we omit improving the evolutionary approach at this point.

5.3 ACET Estimations

Figure 5 depicts the impact of our branch prediction aware code positioning algorithms on the ACET. The commercial, cycle true instruction set simulator CoMET [21] was employed to measure

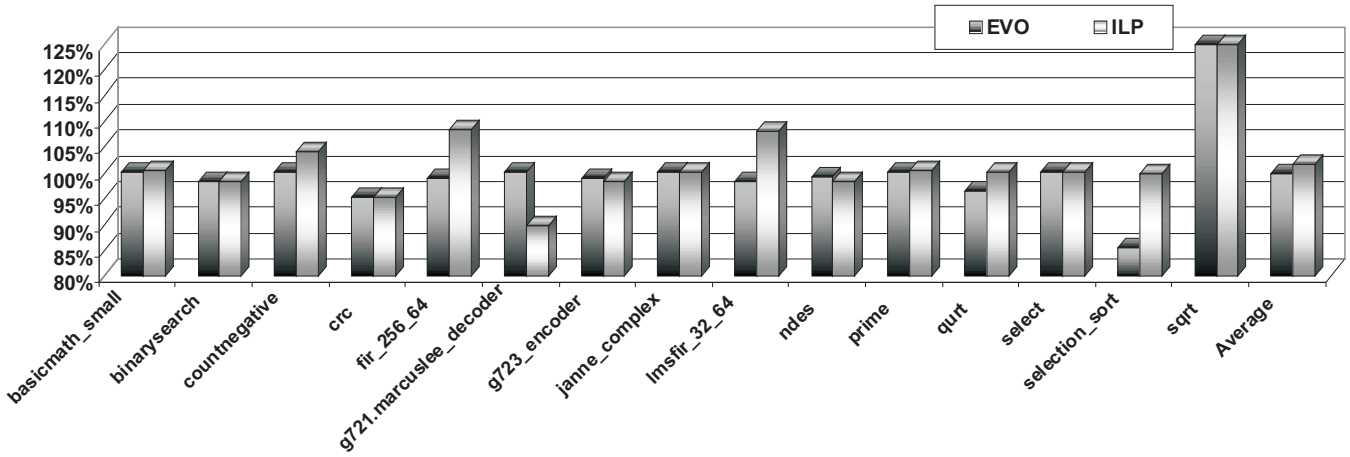

Figure 5: Relative ACETs after Code Positioning Optimizations

ACETs. Once again, the bars depict the resulting ACET of the optimized program as percentage of its "unoptimized" version. For each benchmark, the left bar represents the result achieved by the evolutionary code positioning technique, whereas the right bar represents the result if the ILP-based algorithm is applied. The 100% line is equal to the ACET of the benchmark compiled with the optimization level O3 without any code positioning optimization (the code layout matches the logical order found in the *C* source code).

The evolutionary algorithm was able to decrease the ACET of the considered benchmarks by up to 14.4% (*selection_sort*). But there are also cases, where the ACET is increased (*sqrt*, 24.7% increase). On average, a marginal ACET decrease of 0.3% can be achieved.

Evaluating the ILP-based optimization shows that the algorithm does not have the ability to perform better than the EA. Here, too, ACET reductions can be observed (only up to 10.4% for *g721.marcuslee_decoder*). But once again, there are cases of increased ACETs (up to 24.7% for *sqrt*). On average, the ACET is even worsened by 1.7%.

Comparing Figures 4 and 5 shows that both algorithms behave completely different for the estimated WCET and the ACET: Our WCET-driven branch prediction aware code positioning is best suited to achieve WCET reductions but performs worse for ACET optimization. This is caused by the fact that the WCET serves as metric during the optimization of the programs. The EA evaluates the fitness of individual by invoking a static timing analyzer. The ILP-based optimization employs WCETs of basic blocks as well as worst-case execution frequencies to set up the constraints modeling the objective function. Hence, the impact of any reordering on the ACET can not be taken into account.

The WCET of the benchmark *sqrt*, for instance, can be decreased by 6% whereas the ACET is worsened. In *sqrt*, a loop performs square root computations on floats. For average case scenarios, this loop exits usually after few iterations. Regardless of this fact, a WCET analyzer has to assume the maximum iteration count of the loop as worst case. Thus, the WCEP is different from the most frequently used path. The positions to the benchmark's basic blocks are changed in order to shorten the WCEP. In order to correct the control flow, unconditional jumps are inserted on the most frequently used path which (in this case) does not contribute to the WCEP. The inserted jumps lead to an ACET increase of 24.7%.

These observations obtained comparing the ACET and WCET performance of WCET-tailored optimizations conform to the observations presented in [6].

5.4 Optimization Time

To consider the optimization time, we utilized an Intel Xeon E5506 (2.13 GHz). Most of the time necessary for our novel WCET-driven branch prediction aware code positioning algorithms was consumed by the repetitive WCET analyses using *aiT*. The maximal number of WCET analyses during an evolutionary optimization run amounts to

$$n = \alpha + \lambda * (maxGen - 1) = 400 \tag{19}$$

where α is the size of the initial population and λ the number of offspring individuals. For the ILP-based approach, only a single WCET analysis is necessary to determine the costs C_i for each basic block b_i (cf. Section 3.2).

For a single WCET analysis, up to 20 CPU minutes are required for the *basicmath* benchmark. Thereby, the evolutionary approach requires almost 2 days for the optimization of this benchmark which is not feasible in practice. The ILP-based optimization, in contrast, merely spends 20 minutes for the one required WCET analysis which is highly suitable for most application scenarios. The immense WCET analysis time for the evolutionary approach is the reason why we had to limit the number of benchmarks to 15 in order to avoid optimization times of months.

The complexity of solving the ILPs generated by the optimization discussed in Section 3.2 is of no practical relevance. For a CFG with n nodes, the ILP has a size of $O(n^2)$ constraints and variables. The employed ILP solver $CPLEX$ takes up to 2 CPU minutes (*basicmath*) but mostly terminates within a few seconds for the considered benchmarks. Compared to the WCET analysis required to determine the cost constants C_i for each basic block and the immense time required for the evolutionary algorithm, these values are convenient.

6. CONCLUSIONS AND FUTURE WORK

In this paper, we presented two code positioning techniques considering the influence on branch predictors with the objective to decrease the WCET of a program. The order of basic blocks inside a function is changed with the goal to avoid unconditional branches and to support the branch prediction of conditional branches in order to fetch the frequently visited successor in advance.

We presented an evolutionary approach which employs the well-known techniques mutation and crossover to create offspring individuals whose order of basic blocks hopefully tends to decrease the

WCET of the represented program. Applying this technique, we were able to achieve a WCET decrease of up to 24.7%, with the drawback of repetitive WCET estimations incurring high optimization times. On average, WCET reductions of 8.9% were achieved.

An ILP-based approach was introduced avoiding time-consuming repetitive WCET analyses. Therefore, the control flow of a program and the resulting jump penalties were explicitly modeled to determine an improved order of basic blocks w. r. t. the WCET. In so doing, our algorithm was able to decrease WCET of programs by up to 20.0%. On average, WCET reductions of 6.7% were achieved.

Considering the amount of mispredicted branches, the EA is able to reduce the number of mispredicted branches by up to 56% whereas the ILP-based approach achieves up to 55.1%. On average, the ILP-based optimization is able to outperform the EA by 4.6% (21.4% vs. 16.8% reduction).

In the future, we intend to improve our integer-linear programming based approach to take the influence of different instruction types on the different pipelines of a processor into account. The TriCore processor, for instance, executes unconditional branches on the Load Store pipeline. Inserting branches behind integer instructions can be *dual issued* which means a parallel execution of two instructions on different pipelines. Additionally, line crossing effects should be modeled in the ILP in order to close the gap to the results achieved by the evolutionary algorithm.

Furthermore, we plan to combine our optimal static WCET-aware scratchpad allocation of program code [7] with the ILP-based optimization presented in this paper. Basically, the SPM allocation does not change the logical order of basic blocks residing in the same memories. It should be possible to trigger synergy effects by simultaneous modeling of basic block reordering and the memory allocation in a coupled optimization.

Acknowledgments

The authors would like to thank AbsInt Angewandte Informatik GmbH for their support concerning WCET analysis using the *aiT* framework. A special thank also goes to the Computer Engineering and Networks Laboratory (TIK) of the Swiss Federal Institute of Technology Zurich (ETHZ) for supplying the PISA framework. We would also like to thank Synopsys for the provision of the instruction set simulator CoMET enabling determination of ACETs.

7. REFERENCES

[1] AbsInt Angewandte Informatik GmbH. Worst-Case Execution Time Analyzer aiT for TriCore. 2011. `http://www.absint.com/ait`.

[2] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA — A Platform and Programming Language Independent Interface for Search Algorithms. In *Proc. of EMO*, Faro, Portugal, 2003.

[3] F. Bodin and I. Puaut. A WCET-oriented static branch prediction scheme for real-time systems. In *Proc. of ECRTS*, Palma de Mallorca, Spain, 2005.

[4] C. Burguière, C. Rochange, and P. Sainrat. A Case for Static Branch Prediction Modeling in Real-Time Systems. In *Proc. of RTCSA*, Hong Kong, China, 2005.

[5] J. Eckart and R. Pyka. ICD-LLIR Low-Level Intermediate Representation. `http://www.icd.de/es/icd-llir`, 2011. Informatik Centrum Dortmund.

[6] H. Falk. WCET-aware Register Allocation based on Graph Coloring. In *Proc. DAC*, San Francisco, USA, 2009.

[7] H. Falk and J. C. Kleinsorge. Optimal Static WCET-aware Scratchpad Allocation of Program Code. In *Proc. of DAC*, San Francisco, USA, 2009.

[8] H. Falk and H. Kotthaus. WCET-aware Cache-aware Code Positioning. In *Proc. CASES*, Taipei, Taiwan, 2011.

[9] H. Falk and P. Lokuciejewski. A compiler framework for the reduction of worst-case execution times. *Journal on Real-Time Systems*, 46(2), Oct. 2010.

[10] G. Gebhard and S. Altmeyer. Optimal Task Placement to Improve Cache Performance. In *Proc. of EMSOFT*, New York, USA, 2007.

[11] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure Placement using Temporal-Ordering Information: dealing with Code Size Expansion. *Journal of Embedded Computing*, 1(4), 2005.

[12] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET Benchmarks: Past, Present And Future. In *Proc. of WCET Workshop*, 2010.

[13] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and T. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *Proc. of IISWC*, Austin, Texas, USA, 2001.

[14] Infineon Technologies AG. *TC1796 User's Manual, v2.0*. Munich, Germany, 2007.

[15] International Business Machines Corporation (IBM). IBM ILOG CPLEX V12.1, 1987, 2009.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Proc. of MICRO*, Washington, DC, USA, 1997.

[17] X. Liu, J. Zhang, K. Liang, Y. Yang, and X. Cheng. Basic-block reordering using neural networks. In *Proc. of SMART Workshop*, Ghent, Belgium, 2007.

[18] T. Mitra and A. Roychoudhury. A framework to model branch prediction for worst case execution time analysis. In *Proc. of WCET Workshop*, 2002.

[19] R. Pyka and J. Eckart. ICD-C Compiler Framework. `http://www.icd.de/es/icd-c`, 2011. Informatik Centrum Dortmund.

[20] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. of RTSS*, December 2005.

[21] Synopsys. CoMET, Virtual Prototyping Solution1. 2011. `http://www.synopsys.com`.

[22] UTDSP Benchmark Suite. `http://www.eecg.toronto.edu/~corinna/DSP/infrastructure/UTDSP.html`, 2011.

[23] W. Zhao, D. Whalley, C. Healy, and F. Mueller. WCET Code Positioning. In *Proc. of RTSS*, 2004.

[24] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm for Multiobjective Optimization. In *Proc. of EUROGEN*, Athens, Greece, 2001.