WCET-driven Branch Prediction aware Code Positioning*

Sascha Plazar, Jan Kleinsorge, Peter Marwedel
Computer Science 12
TU Dortmund University
D - 44221 Dortmund, Germany
Firstname.Lastname@tu-dortmund.de

Heiko Falk
Institute of Embedded Systems /Real-Time Systems
Ulm University
D - 89081 Ulm, Germany
Heiko.Falk@uni-ulm.de

ABSTRACT

In the past decades, embedded system designers moved from simple, predictable system designs towards complex systems equipped with caches, branch prediction units and speculative execution. This step was necessary in order to fulfill increasing requirements on computational power. Static analysis techniques considering such speculative units had to be developed to allow the estimation of an upper bound of the execution time of a program. This bound is called worst-case execution time (*WCET*). Its knowledge is crucial to verify whether hard real-time systems satisfy their timing constraints, and the WCET is a key parameter for the design of embedded systems.

In this paper, we propose a WCET-driven branch prediction aware optimization which reorders basic blocks of a function in order to reduce the amount of jump instructions and mispredicted branches. We employed a genetic algorithm which rearranges basic blocks in order to decrease the WCET of a program. This enables a first estimation of the possible optimization potential at the cost of high optimization runtimes. To avoid time consuming repetitive WCET analyses, we developed a new algorithm employing integer-linear programming (*ILP*). The ILP models the worst-case execution path (*WCEP*) of a program and takes branch prediction effects into account. This algorithm enables short optimization runtimes at slightly decreased optimization results. In a case study, the genetic algorithm is able to reduce the benchmarks' WCET by up to 24.7% whereas our ILP-based approach is able to decrease the WCET by up to 20.0%.

Categories and Subject Descriptors

D.3.4 [**Programming Languages**]: Processors—Compilers; Optimization; C.1.3 [**Processor Architectures**]: Other Architecture Styles—Pipeline processors; C.3 [**Real-time and embedded systems**]; B.2.2 [**Arithmetic and Logic Structures**]: Performance Analysis and Design Aids—Worst-case analysis

*The research leading to these results has received funding from the European Community's ArtistDesign Network of Excellence and from the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement n° 216008.

CASES '11, October 9-14,2011, Taipei, Taiwan.

General Terms

Algorithms, Performance

Keywords

Branch Prediction, Code Positioning, WCET

1. INTRODUCTION

Complex processor designs employing branch predictors have become popular to increase the computational power of embedded systems. This feature is developed to work transparently with regard to the software running on a system by integrating a fully autonomous hardware controller. The effectiveness of such a speculative unit highly depends on the control flow of a program and the arrangement of its basic blocks in memory. If the target of a jump instruction can be predicted correctly, the next instruction to be executed can be fetched in advance and thus, the performance can be increased. But if the branch target was mispredicted, the processor pipeline has to be stalled until the next instruction has been fetched from memory. These penalty cycles can lead to a performance decrease.

Branch prediction units are designed to decrease the average-case execution time (*ACET*). Their disadvantage is the limited predictability since it is hard to determine if a jump instruction results in a pipeline stall due to a mispredicted branch target or not. Since many embedded systems have to meet hard real-time constraints, static analysis techniques have been developed to allow safe estimations of the impact of branch prediction units on the worst-case performance of a system [18].

The worst-case execution time (*WCET*) of a program is the upper bound of its execution time for all possible input data and all possible initial system states. The WCET is a key parameter for real-time scheduling and the development of hardware platforms which have to satisfy critical timing constraints. Since the real WCET of a system can not be determined, static timing analyzers are employed to determine WCET estimations (WCET_{est}).

The WCET of a program corresponds to the length of the worst-case execution path (*WCEP*) which is that path of the control flow graph (*CFG*) with the highest execution time. Optimizations of elements like functions on the WCEP can shorten this longest path in such a way that another path becomes the new WCEP. Optimizations of elements not lying on the WCEP will not result in a reduction of the WCET. Hence, possible switches of the WCEP have to be taken into account during optimizations.

In this paper, we present two novel WCET-driven code positioning algorithms to rearrange the order of basic blocks of a function. They aim at improving the performance of static branch predictors and at avoiding unnecessary jumps. The first algorithm employs a

genetic approach which starts with a random population. By exploiting the evolutionary techniques crossover and mutation, offspring individuals are generated which desirably converge to the optimal solution w. r. t. the WCET of a program.

Usually, evolutionary strategies can be implemented with small effort and often without understanding the mechanism behind the optimization problem. Even small memory layout modifications, for instance, can have hardly predictable effects on the instruction fetch unit of a processor. Evolutionary algorithms (*EA*) implicitly consider such effects by evaluating the WCET to determine the fitness values of newly created individuals. Thus, we employ an EA to understand the mechanisms behind the optimization problem and to explore the possible optimization potential of code positioning techniques before developing complex algorithms.

The second optimization discussed in this paper is such a "complex" algorithm. A more promising order of basic blocks is determined based on an integer-linear programming (*ILP*) approach. The ILP explicitly models the WCEP as well as the impact on the branch prediction and thereby avoids repetitive WCET analyses. To enable such WCET-centric optimizations, we employ the sophisticated static WCET analyzer *aiT* developed by AbsInt [1]. The main contributions of this paper are as follows:

- We rearrange the order of basic blocks of a function based on its impact on the WCET of a program.
- An evolutionary approach is presented which fathoms the possible potential of code positioning techniques for sophisticated WCET optimizations.
- Our ILP-based approach determines a more promising order of basic blocks w. r. t. the WCET of a program by explicitly modeling the WCEP of a program. For the first time, both branch penalty costs and the amount of executed unconditional jumps are modeled in the ILP.
- By applying on a set of benchmarks, we show that WCET reductions of up to 24.7% can be achieved.

In this paper, the influence of code positioning techniques on the cache performance is not considered since we already tackle this problem in [8].

This paper is organized as follows: In the next section, an overview of related work is provided. Section 3 presents our new WCET-driven code positioning algorithms. Section 4 introduces the WCET-aware C compiler *WCC* employed to develop our novel algorithms. An evaluation of the performance which is achieved by our WCET-driven branch prediction aware code positioning optimizations is presented in Section 5. Finally, we conclude our work and give a brief overview of future work.

2. RELATED WORK

Burguière et al. [4] compare static and dynamic branch prediction in terms of suitability for WCET analysis. They argue for employing static instead of dynamic branch prediction and show that static branch prediction can achieve lower WCETs in most cases. Mitra et. al, however, present schemes for estimating the effect of dynamic branch predictors on the WCET of a program in [18]. They derive linear inequations, which can be integrated into ILP models for WCET analyses, to bound the number of mispredicted branches during execution.

In [10], a technique for rearranging the positions of tasks to improve the cache performance is presented. The interdependency relation of tasks is evaluated in order to determine a memory layout which maximizes the number of persistent cache sets for each task.

A technique for procedure placement to reduce the cache miss ratio of programs is presented in [11]. Guillon et al. provide an optimal algorithm for memory placement which is improved regarding the unavoidable code size increase caused by gaps in the address space. In contrast to our optimizations, the presented approach does not target WCET reductions and the order of basic blocks stays untouched which wastes optimization potential.

The authors of [17] present their basic block reordering method based on neural networks. For this purpose, Liu et al. detect typical structures in the control-flow graph and employ a branch cost model to choose the layout with minimal costs. Unlike our approach, their model focuses on the optimization of the average-case execution time and is unaware of the WCET of a program.

Zhao et al. also address the problem of determining improved code layouts which decrease the WCET of a program [23]. As opposed to our optimizations, only architectures without branch predictors are considered where unconditional and taken conditional branches always stall the pipeline for a constant number of cycles. An iterative approach is proposed which selects single edges to be contiguous in memory in order to avoid transfers of control. In [3] Bodin et. al, however, aim at improving the WCET of processors supporting compiler-directed branch predictions. By setting a dedicated bit of conditional branch instructions during optimization, the direction to predict is indicated. Optimization potential is wasted, compared to the work presented in this paper, since unconditional branches can not be removed due to missing reordering techniques. In contrast to [23] and [3], our approaches are able to optimize both unconditional and statically predicted conditional branches. Both works also do not explore the space of possible solutions in order to evaluate the quality of their results as it is done in this paper by employing an EA as basis of comparison. Finally, our ILP-based algorithm avoids time consuming repetitive WCET analyses required by state-of-the-art optimization techniques. This is done by explicitly modeling all possible control flow paths as part of the ILP in order to always optimize along the WCEP. Therefore, only *a single* WCET analysis is required.

Falk et al. counteract possible predictability problems of caches with a static allocation of program code to so-called scratchpad memories (*SPM*) [7]. They employ integer-linear programming to select the optimal content of the SPM w. r. t. the program's WCET. Although basic blocks are moved between different memories, the branch prediction can not be influenced. This is due to the unchanged logical order of blocks in each considered memory. For the same reason, unconditional jumps are only optimized indirectly: a jump instructions bypassing some basic blocks can be removed if the bypasses bocks are allocated to a different memory.

Another work considering scratchpad allocation is presented in [20]. Suhendra et al. developed an ILP-based allocation of frequently accessed data objects to faster memories in order to decrease the overall WCET. Their model of the program's WCET and possible execution paths serves as basis for the ILP-based algorithm presented in [7] and was also employed for the technique discussed in Section 3.2.

3. WCET-DRIVEN BRANCH PREDICTION AWARE CODE POSITIONING

Nowadays, embedded systems employ complex CPUs equipped with caches, branch prediction units and speculative execution.

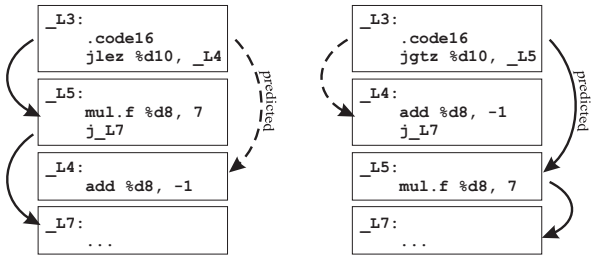


Figure 1: Rearranging code layout to support branch prediction

Such techniques are a source of unpredictability, but are necessary in order to fulfill increasing computational power requirements of current and future embedded applications.

Branch prediction units try to determine the target address of a branch instruction before the instruction is completely processed. Thereby, it is possible to fetch the next instruction to be executed from memory in advance to avoid a performance-decreasing pipeline stall. This can either be the instruction which is directly following the branch instruction in memory (*fall-through edge*) or the branch target (*pass-through edge*).

Either dynamic or static techniques are applied for predicting the branch target: Dynamic branch prediction units store a history, for instance, by implementing a simple counter denoting whether a branch was taken in the past or not. If the branch has been taken in the past, it also tends to be taken in the future. Thus, the branch target is fetched in advance as next instruction to be executed. Otherwise, the fall-through edge was predominantly taken in the past and the instruction immediately following the branch instruction is fetched from memory. Due to this dynamic behavior, the branch predictor can adapt to changing situations if, for instance, the outcome of a branch condition changed due to different input data. This often improves the average case performance but also has the disadvantage that the impact of the branch prediction is hardly predictable by static timing analyzers.

In contrast, a static branch prediction unit determines if a branch will be taken based on static features like the branch direction, the instruction bit width or a dedicated bit in the instruction code. The Infineon TriCore processor TC1796 [14], which is considered in this paper, predicts 16 bit jumps as always taken whereas 32 bit jumps are predicted depending on the branch direction. Forward jumps (to higher addresses) are predicted to be not taken while backward jumps are predicted to be taken. These static features influencing the branch prediction can be evaluated by analyzing the object code without executing the program. Thereby, the influence of the branch prediction can easily be modeled within a timing analyzer.

If the control flow and the memory layout of a program are ill-arranged, it can happen that the branch targets are predominantly mispredicted by a static branch prediction unit. A high number of mispredicted branches leads to an increased number of pipeline stalls and, as a result, in a performance decrease. Therefore, rearranging the order of basic blocks may be promising in order to support the branch prediction unit.

On the left-hand side of Figure 1, a code example for the TC1796 processor and a disadvantageous memory layout is depicted. If register `d10` less or equal zero, the instruction at the end of `_L3` branches to `_L4`. Else, the fall-through edge is passed and `_L5` is executed which in turn jumps to `_L7`. The floating point multiplication in `_L5` takes more cycles than an integer addition. Thus, a static timing analyzer has to assume `_L3, _L5, _L7` as WCEP (represented by solid arrows) if a data flow analysis can not identify the path as infeasible. Due to the static branch prediction which assumes 16 bit jumps to be taken, the first instruction of `_L4` is fetched in advance if the `jlez` instruction leaves the decode stage of the processor pipeline. After evaluating the instruction in the execute stage, the first instruction of `_L5` has to be fetched as actual branch target resulting in two cycles where the pipeline is stalled.

Rearranging the code structure as can be seen on the right-hand side of Figure 1 helps the branch prediction to fetch the correct instruction in advance. Therefore, the position of `_L4` and `_L5` has to be switched and the test condition of `_L3` has to be negated. In this way, one processor cycle can be saved for each execution of `_L3`. Since the unconditional jump at the end of `_L5` is now superfluous, two additional cycles on the WCEP can be saved; one cycle for executing the jump instruction and one for evaluating and fetching the branch target. Inserting a jump at the end of `_L4` to correct the control flow does not worsen the WCET as long `_L4` does not reside on the WCEP.

The *WCC* compiler comprises a jump optimization which *automatically corrects* the control flow by inserting unconditional jumps as well as a correction of branch conditions. Superfluous unconditional branches are removed as well. Thus, we do not explicitly mention its application to optimized programs in the following.

In order to support an automatic optimization of the code layout which is aware of possible WCEP switches, Section 3.1 presents an evolutionary approach, whereas Section 3.2 presents our ILP-based optimization technique.

3.1 Evolutionary Approach

Finding an improved order of basic blocks inside a function w. r. t. the WCET of a program is a complex task which – in most cases – can not be done manually. In order to explore the possible space of solutions automatically, we developed an evolutionary approach. In this way, it is possible to fathom the optimization potential of code positioning algorithms with small implementation effort. Such a procedure is advisable before spending time and effort on developing complex optimization techniques with vague practical effect.

Evolutionary algorithms stem from the domain of artificial intelligence and try to implement the principles of biological evolution. By employing reproduction mechanisms including mutation and recombination, offspring generations are created. From a new generation, stronger individuals w. r. t. a certain fitness function are selected as parents for the next generation. With such an approach, preferably improved solutions are "cultured" instead of tackling an optimization problem analytically.

We employed the PISA framework [2] which defines a common interface for the communication of the so-called selector and variator modules. The selector picks out individuals for the *archive* containing promising individuals for a later use and as parents for an offspring generation, whereas the variator implements the problem representation. The variator is responsible for creating offspring individuals from parents by applying the mentioned crossover and mutation operators as well as for the evaluation of the fitness value. There are several optimization algorithms for single- and multi-objective optimizations problems providing an interface for PISA. The *Strength Pareto Evolutionary Algorithm 2* (SPEA-2) [24] shows good performance for different numbers of objectives at negligible computational power requirements. Hence, it was chosen as selector although a simpler algorithm could be applied as well.

Our implementation of the variator creates individuals which represent the order of basic blocks by a mapping of consecutively numbered positions in memory to basic block names: for each function, a vector stores the basic block names and thereby encodes their position. A one-point crossover at a random position i is applied as recombination operator to create new individuals. Therefore, the first i of k functions of the first parent individual are combined with the last $k - i$ functions of the second parent individual to create an offspring individual. Such a newly created one is mutated with a probability of 1 by exchanging two randomly chosen basic blocks of a randomly chosen function solution vector.

For the evaluation of the fitness of an individual which corresponds to the WCET of the modified program, the sophisticated WCET analyzer *aiT* is employed. A method to avoid redundant WCET analyses is a lookup table for already evaluated solution vectors: if an older individual represents the same solution vector as a newly created one, the appropriate WCET is read from the lookup table instead of invoking *aiT*.

A more sophisticated way to determine an improved order of basis block positions without the need of repetitive WCET estimations is explained in the following section.

3.2 ILP-based optimization

Trial-and-error approaches enhanced by evolutionary techniques as presented in the last section are often time consuming. This is due to the repetitive creation of individuals and evaluation of the fitness function. In contrast, an analytic strategy may yield a problem-aware optimization technique which also leads to good or even optimal solutions. Often, the disadvantage of such "methodological techniques" is their complexity. They also require a high level of knowledge w. r. t. the optimization problem on the part of the developer. Especially for WCET-driven optimizations, the recognition and handling of possible WCEP switches makes optimizations challenging.

This section presents our ILP-based optimization technique which is capable to model a program's control flow in order to always optimize along the WCEP. It determines an improved order of basic blocks w. r. t. the WCET of a program. Our algorithm requires only *a single* WCET analysis and is able to consider the influence of the code layout on the branch prediction. Section 3.2.1 describes the modeling of a function's control-flow in the ILP whereas Section 3.2.2 introduces constraints steering the position of basic blocks. In Section 3.2.3, jump penalties for various jump scenarios are modeled. Finally, Section 3.2.4 models the global control flow whereas Section 3.2.5 describes the ILP's objective function.

3.2.1 ILP Model of the Control Flow of Functions

In the following, ILP variables are represented using lowercase letters whereas constants are represented by uppercase letters. For a function F, the order of its basic blocks in memory is consecutively numbered. The position of a basic block b_i inside F is represented by an integer variable x_i as part of the ILP model. Thus, the value of x_i represents the absolute position inside F which consists of N basic blocks:

$$x_i \in \{1, ..., N\} \tag{1}$$

The costs C_i of basic block b_i represent the WCET of this block for a single execution as part of the unoptimized program.

For reducible CFGs, an innermost loop L of F has exactly one basic block b^L_{entry} being the loops unique entry point, and possibly several back-edges turning it into a cyclic graph. Not considering these back-edges turns L's CFG into an acyclic graph. $G_L = (V, E)$ denotes this acyclic graph in the following. Without loss of generality, it can be assumed that there is at least one basic block b^L_{exit} in G_L being the loop's exit node. The WCET w^L_{exit} of block b^L_{exit} is equal to its costs:

$$w^L_{exit} = C^L_{exit} \tag{2}$$

The WCET of a path leading from a node $b_i \neq b^L_{exit}$ of G_L to one of the exit nodes b^L_{exit} must be greater than or equal to the WCET of any successor of b_i in G_L, plus the cost C_i of b_i:

$$\forall b_i \in V \setminus \{b^L_{exit}\} : \forall (b_i, b_{succ}) \in E : w_i \geq w_{succ} + C_i \tag{3}$$

Since paths are built bottom-up, variable w^L_{entry} models the WCET of all paths of a loop L if it is executed exactly once. In order to model multiple executions of L, all CFG nodes $v \in V$ of G_L are represented by a super-node v_L. The costs of v_L are the product of L's WCET for a single execution and L's maximal loop iteration count:

$$c_L = w^L_{entry} * Count^L_{max} \tag{4}$$

Replacing a loop L by a super-node v_L in the CFG may turn another loop L' of F directly surrounding L into an innermost loop with acyclic CFG G'_L. Hence, the constraints of Equation 3 and 4 can be formulated for L'. This way, the innermost loops of F are successively collapsed in the CFG so that ILP constraints modeling F's control flow are created from the innermost to the outermost loops.

During optimization, a WCEP switch of a program can only happen at such points in the CFG where a basic block b_i has more than one successor. Only there, forks in the control flow are possible where the outgoing paths can have different WCETs. But since Equation 3 is formulated for each successor of b_i, variable w_i always reflects the WCET of any path starting at b_i – irrespective of the fact which successors are actually part of the current WCEP. This way, the constraint of Equation 3 realizes the implicit consideration of WCEPs and their changes in the ILP.

The fundamental structure of the ILP constraints of Equations 2 – 4 stem from the work of Suhendra et al. proposed in [20]. In order to implement a fully functional code positioning technique, these basic constraints had to be refined substantially. Our extensions of the original ILP formulation are described in the following sections.

3.2.2 Position Constraints

The decision variable x_i for a basic block b_i allows a free positioning of each basic block inside a function. However, there are some constraints which have to be regarded. First, without loss of generality, each function F has one dedicated entry block b^F_{entry} with its corresponding decision variable x^F_{entry} which has to be kept as first block of the function:

$$x^F_{entry} = 1 \tag{5}$$

Furthermore, at each logical position of a function F, exactly one block must be assigned. Let V be the set of F's basic blocks, b_i and b_j two basic blocks with their corresponding decision variables x_i and x_j, respectively. Then, a number of constraints has to be formulated to ensure that there are no two variables with the same value in order to avoid several basic blocks at the same position:

$$\forall b_i, b_j \in V : x_i \neq x_j \tag{6}$$

3.2.3 ILP Model of Jump Penalties

The WCET of a basic block b_i does not only depend on its own WCET and the WCET of the outgoing paths starting at b_i, but also

Table 1: Jump Penalties [cycles]

	Predicted	
Outcome	**Taken**	**Not Taken**
Taken	1	2
Not Taken	2	0

on possible jump penalties resulting from rearranging the order of blocks inside a function.

By default, the WCET for a single execution of a basic block b_i also comprises possible jump penalties of unconditional and conditional jump instructions. Table 1 depicts the resulting penalty cycles if a branch was predicted to be taken or not and the real outcome during execution. As it can be seen, the worst-case is a mispredicted branch causing two cycles pipeline stall. In order to simplify the jump penalty constraints presented in the following, the penalty cycles of jump instructions are distilled in advance. Differing from the ILPs presented in [7] and [20], the costs C_i of a basic block b_i used in Equations 2 – 4 thus have to be redefined:

- If block b_i does not end with a jump instruction, its costs C_i equal the WCET for a single execution.
- For a block b_i with an unconditional jump as last instruction, two cycles are subtracted from the WCET to derive the costs C_i. These two cycles are composed of one cycle for executing the jump and of one cycle pipeline stall for determining the jump target.
- The WCET of a block b_i ending with a conditional jump instruction is determined by evaluating all outgoing edges to find the edge resulting in the highest execution time of b_i. Depending on whether this is the fall- or pass-through edge, it is also known if the branch is taken or not. To determine the jump penalties for a conditional branch instruction from Table 1, the bit width and the jump direction has to be evaluated, too. If, for instance, a conditional branch was mispredicted, two cycles pipeline stall occur. In contrast to an unconditional jump, only these two cycles are subtracted from the WCET of the corresponding block b_i in order to calculate the costs C_i. This is done since the conditional jump itself can not be removed.

To determine the jump penalties as ILP constraints, the different jump scenarios which are depicted in Figure 2 and their impact on the branch prediction of the employed processor have to be modeled: The simple case in Figure 2a is an implicit edge between two contiguous basic blocks b_i and b_j where b_i does not end with a jump instruction. If the ILP decides to not allocate these blocks contiguously ($x_i \neq x_j - 1$), then an unconditional jump has to be inserted resulting in two cycles penalty:

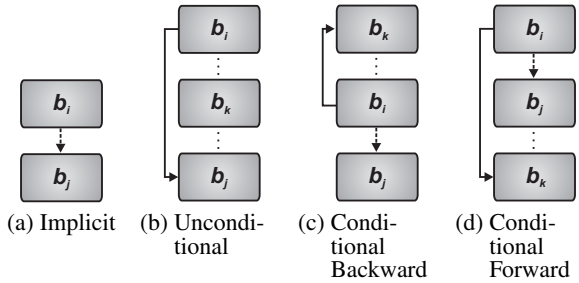
$$jp^{i}_{impl} = 2 - 2 * (b_i \circ b_j) \tag{7}$$

In Equation 7, the operator $\circ$ checks if two blocks are contiguous in memory by evaluating the corresponding decision variables:

$$(b_i \circ b_j) = \begin{cases} 1 & \text{if } x_i = x_j - 1 \\ 0 & \text{else} \end{cases} \tag{8}$$

The $\circ$ operator can be modeled within an ILP, but we omitted the listing of these constraints for the sake of simplicity.

An unconditional branch, as shown in Figure 2b, also connects exactly two basic blocks b_i and b_j and usually bypasses a number of other basic blocks with a jump instruction. Due to the fact, that the



(a) Implicit (b) Unconditional (c) Conditional Backward (d) Conditional Forward

Figure 2: Typical Jump Scenarios

jump costs were distilled from the costs C_i in advance, this case can be also handled by Equation 7: If the ILP decides to allocate these blocks contiguously, nothing has to be done since removing the jump results in jump scenario 2a. Otherwise, two cycles penalty have to be readded for each execution of b_i.

Compared to the unconditional branch instructions, conditional jumps require a rather complex modeling of jump constraints: The static branch prediction of the TriCore TC1796 distinguishes between 16 bit conditional jumps which are always predicted as taken and 32 bit conditional jumps where the prediction depends on the jump direction. A 32 bit conditional jump with backward displacement (cf. Figure 2c) is predicted as taken whereas the same instruction with forward displacement (cf. Figure 2d) is predicted as not taken.

Jumps with 16 bit width are always predicted taken; in compliance with the second column of Table 1 either one cycle for a correctly predicted (pass-through edge) or two cycles for a mispredicted branch (fall-through edge) have to be added as penalty. Since the costs C_i of a block b_i are free of any jump penalties, the jump penalty constraints for both successors can now be handled in the same way. For each edge to successor $b_{succ} \in \{b_j, b_k\}$, a separate constraint determines the jump penalty depending on whether blocks b_i, b_{succ} are contiguous in memory or not:

$$jp^{i,succ}_{cond16} = 1 + (b_i \circ b_{succ}) \tag{9}$$

If $b_i \rightarrow b_{succ}$ is not contiguous (pass-through edge), the penalty for visiting b_{succ} from b_i is only one cycle for a correctly predicted branch. But if the edge to b_{succ} is the fall-through edge ($(b_i \circ b_{succ}) = 1$), a second cycle for a mispredicted branch is added to the penalty.

In contrast, 32 bit jumps require more complex constraints since the prediction of the target depends on the direction of the jump. For a backward jump which can be seen in Figure 2c, the second column of Table 1 has to be modeled whereas for a forward jump, the third column applies. As done for the 16 bit jump penalties, there is no need to care about the initial order of the basic blocks and the resulting jump penalties due to the distilled costs C_i. Instead, the four possible jump scenarios for a block b_i and its successors b_j and b_k are modeled as constraints. In the following, we present the jump penalties of edge $b_i \rightarrow b_j$, but edge $b_i \rightarrow b_k$ is modeled analogously:

1. Blocks b_i and b_j are contiguous (fall-through edge) and jumping to b_k results in forward displacement (predicted not taken). According to Table 1, the correctly predicted implicit edge results in no penalties:

$$jp^{i,j}_{cond32} \geq 0 * (b_i \circ b_j \wedge x_i < x_k) \tag{10}$$

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:

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

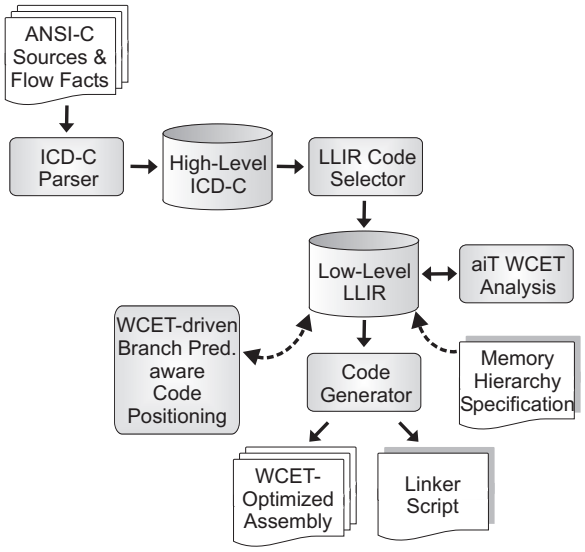


Figure 3: Workflow of the WCET-aware C compiler WCC

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

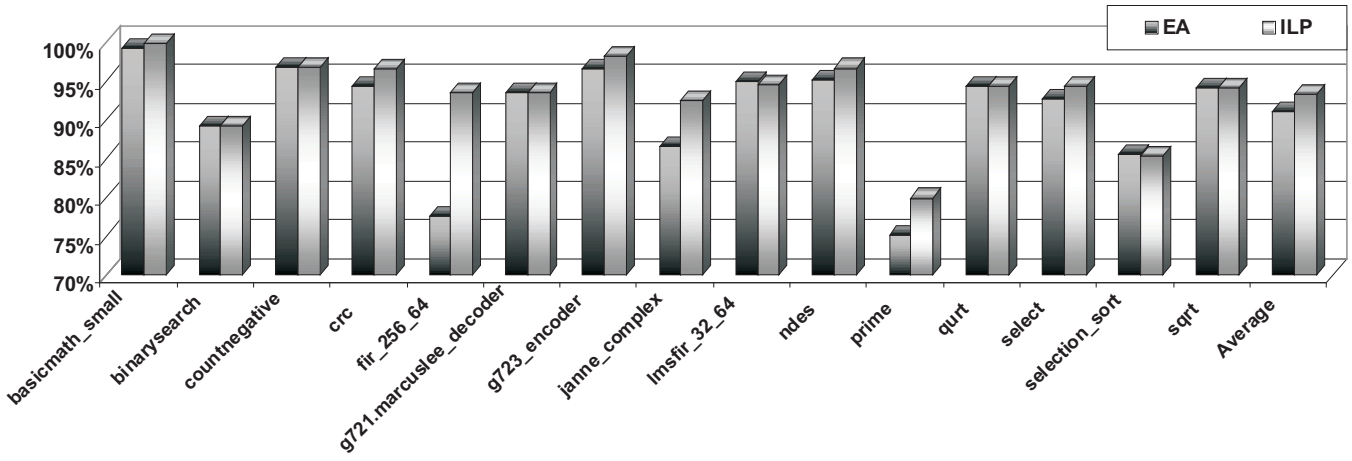


Figure 4: Relative WCETs after Code Positioning Optimizations

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

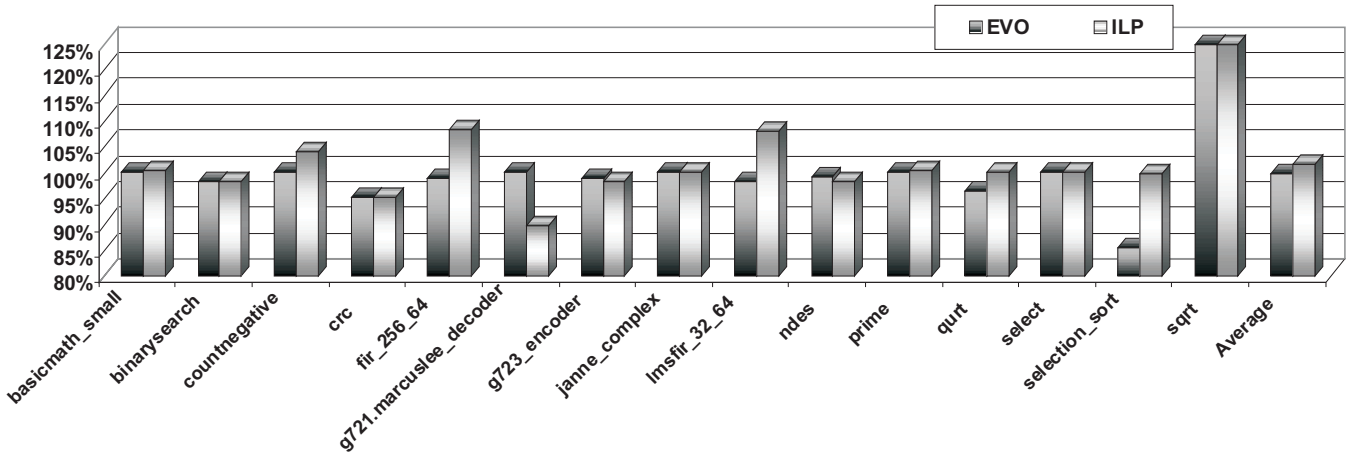


Figure 5: Relative ACETs after Code Positioning Optimizations

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.