Compiler-Based Code Compression for Hard Real-Time Systems

Kateryna Muts
Hamburg University of Technology
Institute of Embedded Systems
Hamburg, Germany
k.muts@tuhh.de

Arno Luppold
Hamburg University of Technology
Institute of Embedded Systems
Hamburg, Germany
arno.luppold@tuhh.de

Heiko Falk
Hamburg University of Technology
Institute of Embedded Systems
Hamburg, Germany
heiko.falk@tuhh.de

ABSTRACT

Real-Time Systems often come with additional requirements apart from being functionally correct and adhering to their timing constraints. Another common additional optimization goal is to meet code size requirements. Code compression techniques might be utilized to meet code size constraints in embedded systems. We show how to extend a compiler targeting hard real-time systems by an asymmetric compiler-based code compression/decompression, where the compression is performed at the compilation time and the decompression takes place at the execution time. Moreover, experimental results show the impact of the decompression algorithm on the estimated Worst-Case Execution Time that is one of the key properties of hard real-time systems.

CCS CONCEPTS

• **Information systems** → **Compression strategies**; • **Computer systems organization** → **Real-time systems**; • **Software and its engineering** → **Compilers**; • **Mathematics of computing** → *Integer programming*.

KEYWORDS

Compiler, Compression, Run-Time Decompression, Real-Time Systems, Integer-Linear Programming

ACM Reference Format:
Kateryna Muts, Arno Luppold, and Heiko Falk. 2019. Compiler-Based Code Compression for Hard Real-Time Systems. In *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES '19), May 27–28, 2019, Sankt Goar, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3323439.3323976

1 INTRODUCTION

An embedded system that must react within a given deadline and where missing this deadline might lead to a catastrophic system failure, is called a *hard real-time system*. For this reason, the Worst-Case Execution Time (WCET) is one of the key properties of such a system. The WCET is defined as the worst possible execution time of a program, independently from its input data.

To ensure that timing constraints are met, optimizing compilers such as the WCET-Aware C Compiler Framework (WCC) [5] can be used. WCC focuses on minimizing the WCET of the compiled program in order to guarantee that all timing constraints will always be met. It is also able to optimize systems with several tasks [12] running on multiple cores [14] with regard to their schedulability.

However, program size is another important criterion in modern embedded systems for which hard design restrictions apply due to increasing code complexity of embedded applications on the one hand and limited memory space on the other hand.

Moreover, we focus on a target architecture that has different memories for storing code and data, e.g., the TC1797 microprocessor from Infineon. If the space in the code section is not enough, then the code can be stored in the data section and moved back in parts to the code section before being executed. However, this might lead to data memory overflow. Thus, in this case, code compression is required before storing the code in the data section.

For PC systems, run-time compression techniques are used that allow to distribute compact program files, which are then extracted at execution time. However, these methods often decompress the whole application into memory, which is impractical in the world of embedded systems because of very limited memory on embedded platforms. At the same time, the hard real-time capability of these methods is not examined, so that they are not suitable for use in safety-critical systems. For the purpose described above, our approach applies compression not to the whole program simultaneously, but to the smaller parts of it, namely to individual functions.

Asymmetric compression methods are characterized by the fact that compression is very time-consuming comparing to decompression, but still achieves good compression rates, while the decompression is fast. Therefore, an asymmetric method for code compression is suitable for use in embedded real-time systems, where the execution time of a program is one of the most important criteria. Using such a method avoids a dramatic increase in the run-time performing decompression during the execution of the program and at the same time, during compilation time a good compression rate can be achieved and moreover, the speed of compression is not critical in this case.

It is expected that in any case, the additional computational effort for decompression on the embedded system is not negligible. Therefore, the compiler must weigh exactly which part of the code can be compressed such that the necessary decompression at run-time will not lead to a violation of time limits. In addition, our goal is to reduce the code size of a program with as little WCET penalty as possible. Hence, the compiler must be able to find a trade-off between code size and the WCET of the final program. Consequently, in principle, the decompression algorithm should be

SCOPES '19, May 27–28, 2019, Sankt Goar, Germany

ACM ISBN 978-1-4503-6762-2/19/05…$15.00
https://doi.org/10.1145/3323439.3323976

adjusted in such a way that it can be analyzed within the compiler in order to compute the WCET of the final program. Moreover, since the code compression takes place at compilation time, the input data for the decompression routines is known within the compiler. Thus, the WCET of the decompression method can be safely estimated. In our approach, the computed WCET is used in an Integer-Linear Programming (ILP) model to find a trade-off between the code size and the WCET of the final program at compilation time.

Due to the limitations of caches, some architectures are equipped with fully software-controllable secondary memories. These are memories that are tightly integrated with the CPU to achieve best possible performance. These scratchpad memories (SPM) can be accessed directly and are therefore in general well suited for optimizations regarding energy consumption and execution time. Hence, SPM is utilized as a buffer to which compressed code will be decompressed in our model.

The contribution of this paper is a completely novel and unique compiler framework where:

- functions are considered as candidates for compression to deal with the highest level of abstraction as the first approach;
- the compression takes place at compilation time;
- the compressed functions are decompressed into SPM at execution time before being called;
- an asymmetric compression method is utilized for the compression/decompression;
- the estimated WCET of the decompression method is computed as precisely as possible;
- the compiler is able to find a trade-off between the code size and the WCET of the final program, so that time limits are not violated;
- the optimized program is not functionally broken and can be executed on the target architecture.

This paper is organized as follows: Section 2 gives a brief overview of related work with regard to both compression and decompression techniques. Section 3 briefly introduces the WCC compiler framework used as a basis for the proposed compression/decompression technique. Section 4 explains the compression/decompression framework in detail. In Section 5, we present evaluation results.

2 RELATED WORK

Code compression/decompression has been and remains a hot topic [2] [21]. Different approaches are used depending on the compression strategy such as Huffman coding, dictionary-based or combinations of them. There exist different implementation techniques, namely software or hardware-based methods. Decompression schemes are also characterized by the location of the decompression engine: between the cache and the memory for the pre-cache approaches, between the cache and the processor for post-cache schemes or inside the processor core.

Pinter and Waldman [17] present a software-based code compression scheme that reduces the storage space of a program. In the paper, overheads in term of run-time and memory consumption were considered, but the WCET was not taken into account. In contrast to our approach where functions are considered as compression candidates, in that paper, sets of basic blocks were chosen as compression regions. The scheme compresses and embeds the regions in the code together with code that invokes a run-time library; this library is referred to as a decompression engine.

Dias and Moreno [3] present a code compression method for ARM embedded processors where different dictionaries are used, all of them based on the traditional Huffman algorithm. It was developed with the main objective of reducing the number of accesses to the instruction cache and therefore the main memory. The method was designed for pre-cache decompression architectures. Code size and compression rate were analyzed, while the running time was not taken into account in the paper.

Ros and Sutton [19] described the application of single- and multiple-instruction dictionary methods for code compression to decrease overall code size for the TI TMS320C6xxx DSP family. The compression is applied at the instruction level and not to functions as in our model. In contrast to our approach, hardware was used to analyze instructions as they are fetched from memory and decide whether to allow the instruction to pass on to the CPU unaltered, or whether to decompress the recognized code-word by looking up a dictionary and passing-on the dictionary word instead.

Helan et al. [7] present a technique that can be used in embedded systems to reduce the memory usage. Two methods were discussed in the paper, namely Bit Mask code compression and dictionary-based code compression. The Bit Mask code compression is to record mismatched values and their positions to reduce the greater number of instructions. In addition, a dictionary selection algorithm was proposed to increase the instruction match rates. So, various steps of code compression were combined into a new algorithm here to improve the compression performance in smaller hardware. Furthermore, the separated dictionary architecture was proposed to improve the performance of the decompression engine. The implementation technique described in the paper is a hardware-based method, while we consider a pure software approach.

Ozaktas et al. [15] analyzed the impact of code compression on the estimated Worst-Case Execution Time of critical tasks that must meet at the same time code size constraints and timing deadlines. They used a post-cache code compression technique that is likely to optimize at the same time the code size and the energy consumption. Since their intention was to consider high-performance processors, they opted for in-pipeline decompression that, in addition, avoids the complexity of handling different address spaces. However, in this case a decompression stage must be added except if the processor already has a stage for translation of microcoded instructions into instructions. Thanks to the use of an in-pipeline decompression engine, the decompression time penalty was hidden by pipelined execution. This is why experiments show an improvement of the observed execution time besides the reduction of the code size. In contrast to this approach, we consider a technique that can be done using only a compiler framework.

Ozturk et al. [16] present an approach for automated data compression/decompression. The goal of the paper was to study how automated compiler support can help in deciding the set of data elements to compress/decompress and the points during execution at which these compressions/decompressions should be performed. The proposed compiler support achieves this by analyzing the source code of the application to be optimized and identifying the order in which the different data blocks are accessed. Based

on this analysis, the compiler then automatically inserts compression/decompression calls in the application code. In the paper, a compiler-based implementation is described, however, the technique is designed for data compression/decompression and cannot be used for code compression/decompression that is considered in this paper.

3 WCC FRAMEWORK

We use the WCET-aware C compiler framework WCC [5] as a basis for the compression and decompression optimization. The main components are a parser, the high-level representation, a code selector, the low-level representation, followed by a code generator and the integration of a WCET analyzer. WCC is tightly coupled to a static WCET analyzer, the tool aiT [1].

The parser is compatible with ANSI-C and creates the high-level intermediate representation from the source files. The intermediate representation of the C code is machine independent and features code analyses and high-level optimizations, that can be considered as multi-objective optimizations [13]. The considered C programs can be annotated with flow facts within the ANSI-C source code. This data provides information about the code structure, such as the number of loop iterations or recursion depths, and is mandatory for a static WCET analysis. We consider this information as given for the original C programs and it is otherwise out of scope for this work. However, loop bounds for the decompression routine are computed and annotated at compile time as described in Section 4.4 to perform a static WCET analysis of the decompression code.

A code selector generates the low-level intermediate representation which is a framework to model any kind of machine instruction. Numerous optimizations are available at the low-level representation, e.g., to minimize energy consumption [20]. Finally, the low-level representation is then processed by an assembler and a linker to produce the final executable.

Our approach makes the following **assumptions**:

(1) Recall that SPM is used as a buffer. The execution of the decompressed functions from SPM, which is much faster than the main memory, will definitely of itself improve the final WCET. For this reason, the final WCET and code size of the program are not compared with the initial values where all code is executed from slow Flash memory. Instead, the functions to be compressed are first moved to the SPM and then, the initial WCETs and code sizes are computed. In this case, only the influence of compression/decompression to the objectives is considered;
(2) All compressed functions are decompressed right upfront when starting an optimized program. This assumption is rather strict, future work will relax it by adding a more sophisticated method for choosing the positions in the code where the compressed functions have to be decompressed before being called;
(3) Any function from the original program can be considered as a candidate for compression except the entry point of the program. This is a natural assumption, since then, at least one function remains in the code for calling the decompression routines;
(4) Data compression is not considered, since we concentrate on code compression.

The code compression is fully integrated within WCC. The phases of the WCC compression and decompression are shown in Figure 1. The process starts with selecting functions for compression using an ILP model as described in Section 4.1. The bytes corresponding to the selected functions are extracted from the original binary file and compressed as presented in Section 4.2. The compressed bytes corresponding to the selected functions are stored as data objects in the original code. In the next step, the calls of the decompression routine are inserted in the code with the compressed bytes passed as a parameter in order to enable the run-time decompression. The details of the decompression phase can be found in Section 4.3. Next, the code for decompression is annotated with flow facts as shown in Section 4.4. This step is necessary for the more precise computation of the final estimated WCET. Finally, the original and uncompressed functions, that were stored as compressed objects, are deleted from the code.

The output of WCC is a program that can be executed on the target architecture, where compressed bytes are represented as data objects. Furthermore, the final executable contains the decompression function. Before calling the original function the decompression function is utilized at execution time to decompress the compressed bytes corresponding to the original function.

4 CODE COMPRESSION

Our goal is to reduce the code size of a program with as little WCET penalty as possible. Therefore, we use FastLZ library [8] for compression and decompression. FastLZ is an improvement over Herman Vogt's LZV and Marc Lehmann's LZF algorithms. It is a lossless data compression library meaning that the original data is perfectly reconstructed from the compressed data, in contrast to lossy compression which allows only an approximate reconstruction of the data and therefore it is not suitable in our case. FastLZ decompression is very simple and quite fast which is important in our approach, since decompression occurs at execution time and may increase the WCET of a program dramatically. FastLZ is implemented in portable C, so it can be easily compiled and analyzed by WCC. To the best of our knowledge, the FastLZ library has never been analyzed in terms of the WCET.

4.1 ILP Selection Model

In our approach, we consider functions as candidates for compression. The selection process is divided into two parts: the pre-phase and the ILP selection model.

For the reasons described in Assumption 3 in Section 3, the pre-phase considers all functions, except the entry point of the program, as candidates for compression. In this step, a compression ratio is used as the initial criterion to measure the efficiency of compressing a function. A *compression ratio CR* is defined as the size required to store the compressed bytes corresponding to the function divided by the size required to store the original function.

Definition 4.1. The compression ratio is

$$CR = \frac{size(compressed\,function)}{size(original\,function)} \quad (1)$$

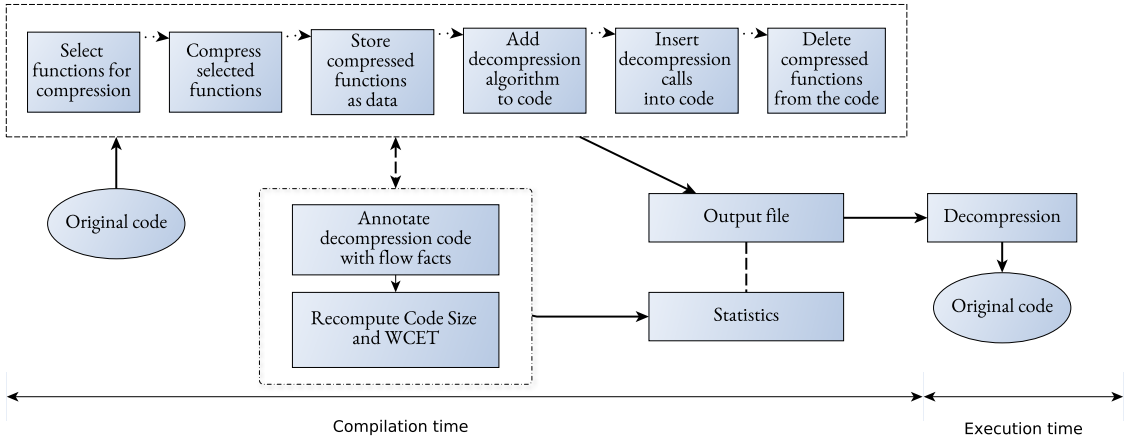

Figure 1: Compression and decompression phases.

If the compression ratio of a function is less than 1, then the function is considered as a candidate for compression by the ILP model.

We consider two objectives in our approach, namely the code size and WCET, both of which have to be minimized. On the one hand, we deal with multi-objective optimization, but on the other hand, the main aim of the compression techniques is to minimize the code size, so we do not accept solutions that lead to an increase in the code size. Moreover, due to Assumption 1, the WCET can only increase after compressing a function because of the WCET of the decompression routine that decompresses the compressed functions at run-time. For these reasons, we aim to reduce code size as much as possible without violating WCET constraints. In this case the problem has a unique solution.

Let us assume that the pre-phase described above has selected N functions $\{f_1, f_2, .., f_N\}$ as candidates for compression. Moreover, CS_{init} and $WCET_{init}$ denote the initial code size and WCET of the original program, respectively, taking into consideration Assumption 1. Then, the increase in the WCET compressing the function f_i, $i = 1, 2, ..N$ is defined as follows:

$$\Delta WCET_i = \frac{WCET_i^{decomp}}{WCET_{init}}, \tag{2}$$

where $WCET_i^{decomp}$ is the WCET of the decompression routine decompressing function f_i.

Calling the decompression routine in the code to decompress the compressed functions requires additional changes of some variables, e.g., the original size of a function, the size of the compressed bytes, the memory address to which the function has to be decompressed, etc. Since these changes definitely have an influence on WCET and code size, they are also taken into account in the ILP model. The WCET change due to calling the decompression code is already included in $WCET_i^{decomp}$ (cf. equation (2)), while the increase in the code size is defined as CS_i^{decomp} for every function f_i, $i = 1, 2, .., N$. The code size of the decompression routine itself is not included in CS_i^{decomp}, because the decompression code is inserted just once as an additional function and will be considered later.

Consequently, the change in code size after compressing the function f_i, $i = 1, 2, ..N$ is defined as follows:

$$\Delta CS_i = \frac{CS_i^{decomp} - CS_i}{CS_{init}}, \tag{3}$$

where CS_i is the code size of the original function f_i.

It should be mentioned, ΔCS_i is defined in such a way that it usually results in a negative value and consequently, has to be minimized to achieve better compression of the final executable.

In the ILP model, we consider the binary variables x_i, $x_i \in \{0, 1\}$ that correspond to the functions f_i, $i = 1, ..N$ identified by the pre-phase as compression candidates.

$$x_i = \begin{cases} 1, & f_i \text{ is being compressed,} \\ 0, & \text{otherwise.} \end{cases} \tag{4}$$

Then, the objective function F is defined as follows:

$$F(x) = \sum_{i=1}^{N} (C \cdot \Delta WCET_i + \Delta CS_i) \cdot x_i, \tag{5}$$

where C is a positive weight describing how much slower the WCET has to increase in comparison to the decrease of the code size compressing functions. C is a user-defined parameter that can be set by a system designer. Choosing a value for the parameter C is a difficult task and will be considered in the future work. However, in the simplest case, when the change in the code size is only of importance and the WCET is simply bounded by a constant as defined in (6), C can be set to 0.

The quantities $\Delta WCET_i$ and ΔCS_i are defined in such a way that both of them have to be minimized. Consequently, the objective function F is also being minimized.

Additional **constraints** are considered in the ILP model:

(1) To avoid the violation of the time limits the increase in the WCET must not be greater than the predefined value

$\Delta WCET_{limit}$.

$$\sum_{i=1}^{N} \Delta WCET_i \cdot x_i \leq \Delta WCET_{limit}; \tag{6}$$

(2) The code size never increases even though the decompression code is inserted in the final binary file.

$$\frac{CS_{decomp}}{CS_{init}} \cdot \max_i(x_i) + \sum_{i=1}^{N} \Delta CS_i \cdot x_i \leq 0, \tag{7}$$

where CS_{decomp} is the code size of the decompression routine.
To rewrite the non-linear operator *max* in inequality (7) as a linear operator, we define a new artificial variable $y \in \{0, 1\}$. Then, inequality (7) is equivalent to the following system of inequalities:

$$\begin{cases} \frac{CS_{decomp}}{CS_{orig}} \cdot y + \sum_{i=1}^{N} \Delta CS_i \cdot x_i \leq 0, \\ y \geq x_i, \qquad \forall i \in \{1, 2, .., N\}; \end{cases} \tag{8}$$

(3) Due to the limited data storage of the target architecture the total size of the compressed bytes must not be greater than the predefined value DS_{limit}.

$$\sum_{i=1}^{N} DS_i \cdot x_i \leq DS_{limit}, \tag{9}$$

where DS_i is the size of the compressed bytes after compressing the function f_i;

(4) Since the size of the SPM is limited, the total code size of the decompressed functions must not be greater than the predefined value SPM_{limit}.

$$\sum_{i=1}^{N} CS_i \cdot x_i \leq SPM_{limit}. \tag{10}$$

This constraint has to be added to the ILP model due to Assumption 2.

The final minimization problem is

$$\begin{aligned} &\min F(x) \\ &\text{subject to} \quad (6), (8), (9), (10) \\ &\text{and} \quad x = (x_1, .., x_N), \\ &\quad x_i \in \{0, 1\} \quad \forall i \in \{1, 2, .., N\}. \end{aligned} \tag{11}$$

The ILP model contains 4 parameters in total. However, DS_{limit} and SPM_{limit} model the restrictions of the target architecture and are computed by a compiler automatically. $\Delta WCET_{limit}$ and C are values that on the one hand control the possible WCET increases and on the other hand model the dependence between the code size decrease and WCET increase as shown in Section 5 for the benchmark *ndes, MRTC*.

4.2 Compression

In the compression phase, the functions that are selected as described in Section 4.1 are compressed as shown in Figure 2.

First, the start address in the original binary file and the code size of a function are used to extract bytes corresponding to the function from the executable file of the original program. The start address and code size of the functions are known within WCC.

Next, the extracted bytes are passed to the FastLZ [8] compression algorithm as input data and the compressed data is returned to the WCC compression routine.

After the bytes corresponding to the function are compressed, they are stored as a data object in the original code and thus can be used by the decompressor at execution time.

The compression of the code is done only at compile time in our model, hence, the code of the compression routine is not included in the final executable.

4.3 Decompression

Decompression takes place at execution time and since we consider the software-based approach in our model, the decompression code has to be included in the final binary file. Moreover, static WCET analysis for the decompression code can be performed within WCC and the estimated WCET is used in the ILP from Section 4.1 to model the WCET increases per compressed function (cf. equation (2)). For these reasons, in contrast to the compression, the C file with the decompression routine is automatically included in the code generation process by WCC.

WCC takes the original source code and the code of the decompression as input parameters. Then, functions for compression are selected and the compressed bytes corresponding to the selected functions are stored in the original code as described in Sections 4.1 and 4.2. At this point, the compressed data is already available in the code and is ready to be used by the decompression routine at execution time.

Then, the preparation phase for the run-time decompression can be started (cf. Figure 3). First, for every compressed function, the calls of the decompression routine are inserted into the code with the compressed data passed as a parameter. Next, the decompression code has to be analyzed in terms of WCET, namely the necessary loop bounds being computed and annotated in the decompression code to enable a static WCET analysis as described in the next Section. Finally, the estimated WCET can be computed and used for the optimization.

As described in Assumption 3 from Section 3, the compressed bytes corresponding to the functions are decompressed at the beginning of the entry point of the original program into a buffer. Recall that we consider SPM as a buffer, since after compressed data being decompressed into SPM, the function can be called directly from there.

4.4 WCET Estimation

To estimate the WCET of the final program, not only is the WCET of the original code needed but the WCET of the decompression code must also be taken into account, since we consider a run-time software-based decompression in our approach.

As already mentioned, WCC is coupled to the static WCET analyzer aiT [1]. The ANSI-C source code has to be annotated with flow facts in order to enable the WCET analysis by aiT. R. Kirner defined flow facts in his Ph.D thesis [10] as follows:

Definition 4.2. Flow facts are meta-information which provide hints about the set of possible control flow paths of a program.

Moreover, there are two types of flow facts:

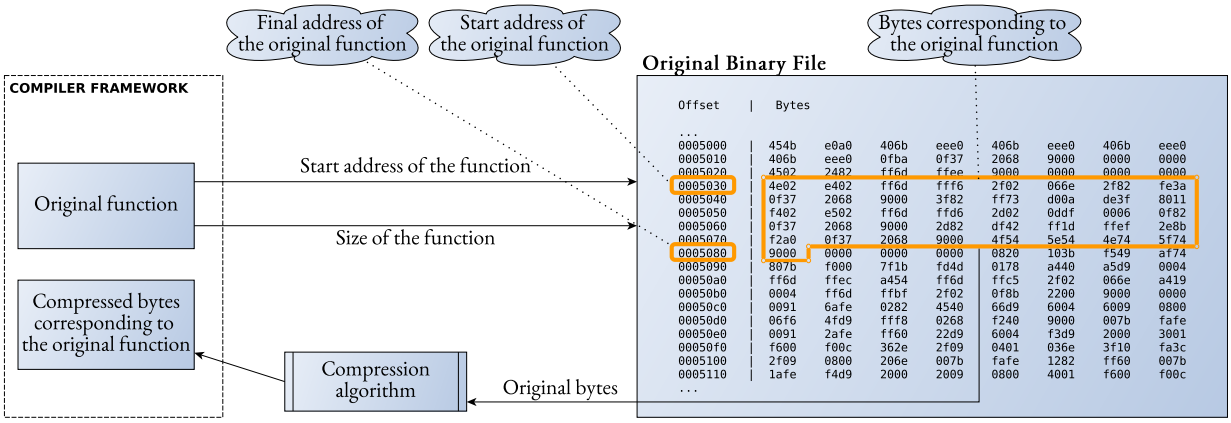

Figure 2: Compression phase.

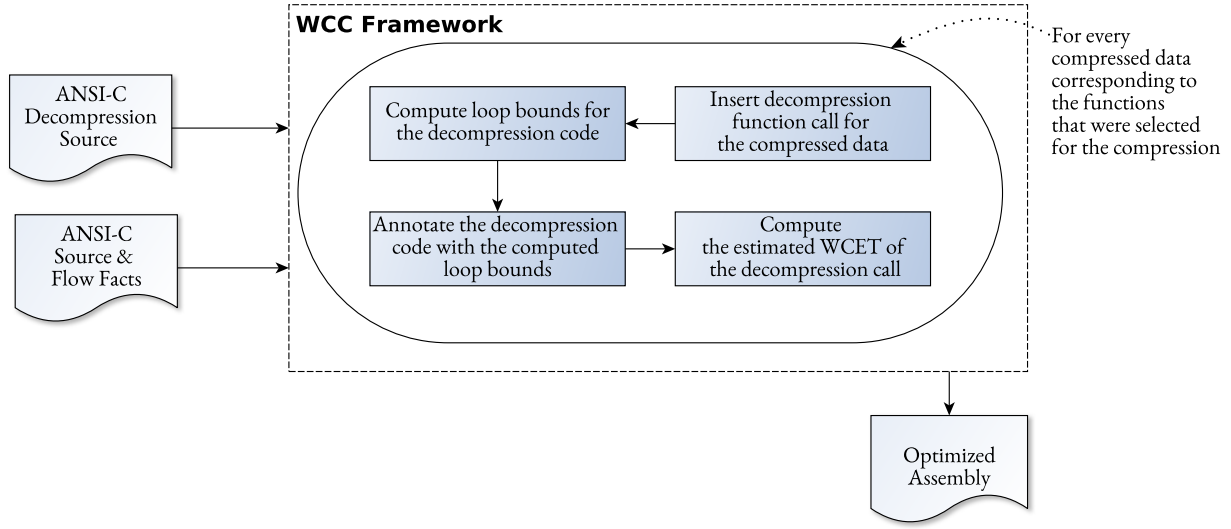

Figure 3: Preparation phase for the run-time decompression.

Definition 4.3. Flow facts which exist due to the structure and semantics of the program code are called *implicit flow facts*. Flow facts which are provided by the user are called *annotated flow facts*.

User-provided flow facts are important for a precise WCET-analysis, because the implicit flow facts are usually only of limited quality and the control flow of a program depends on the input data, which is not considered in the computation of implicit flow facts.

We consider the flow facts of the original program as given, since it is out of scope of this work, while the flow facts of the decompression code are computed automatically by WCC due to the fact that they depend on the input data, namely compressed bytes, that are not known by a user.

The flow facts that are attached to the decompression code at compile time are so-called *loop bounds*. They provide an upper and lower bound for number of iterations of the annotated loop. Let us consider as an example the part of the decompression code that is shown in Listing 1. It contains a loop at line 7 that has loop bounds that depend on the variable *ctrl* which, in turn, depends

```
1  int FASTLZ_DECOMPRESSOR(const void* input, int
       length, void* output, int maxout)
2  {
3    ...
4    int ctrl = (*input++) & 31;
5    ...
6    _Pragma("loopbound min Num1 max Num2")
7    for( --ctrl; ctrl; ctrl++ ){
8     *op++=*ip++;
9    }
10   ...
11 }
```

Listing 1: Example of parametric loop bounds

on the input data passed to the decompression function. In other words, the values of the loop bounds depend on the compressed data that has to be decompressed. Hence, for every data object that represents the compressed function in the final binary file, the loop bounds are computed and updated within WCC, so that the WCET of the decompression calls can be calculated precisely.

5 EVALUATION

Evaluations are conducted using the WCET-aware C compiler framework WCC for the Infineon TriCore TC1797 micro-controller which is commonly used in the automotive domain. All computation runs, ILPs and WCET analyses are executed on a dual CPU Intel XEON server with 96 GB RAM on Ubuntu 18.04.1 LTS. Each CPU consists of 20 cores with a nominal speed of 2.30 GHz. aiT 18.04 is used within WCC to enable a static WCET analysis. The ILPs were solved using Gurobi 8.1.0. For checking the correctness of the final binary file the Synopsys CoMET platform simulator [9] was used.

In the selection phase (cf. Section 4.1), we assume weight C is equal to 1 in the objective function (5) in order to get balanced results, i.e., the WCET increase is not greater than the code size decrease. Moreover, we assume that 50 % is an admissible increase in the WCET from the original one, consequently, $\Delta WCET_{limit}$ is set to 0.5 in the ILP Constraint 1. The constants DS_{limit} and SPM_{limit} from Constraints 3 and 4 are set to the available free space of the data memory and SPM, respectively. The size of data memory of the considered target architecture is 88K and the size of the SPM is 39K.

The benchmarks of the test suites MRTC [6], JetBench [18] and MediaBench [11] with annotated loop bounds from the TACLe-Bench project [4] were used to evaluate the proposed compression and decompression technique.

Figure 4 shows the total number of functions in the benchmarks from MRTC with the excluded function *main* by Assumption 3 from Section 3. Moreover, Figure 4 shows the number of functions that were selected as candidates in the selection pre-phase and the number of functions that were finally compressed after solving the ILP as described in Section 4.1.

In MRTC, there are 3 benchmarks that contain functions which are compressed during the optimization, namely *adpcm_decoder*, *adpcm_encoder* and *lms*. For these three benchmarks, the WCET, code size and the code size of the decompression routine are presented in Figure 5, where 1 corresponds to the initial WCET and code size. We observe for the benchmarks *adpcm_decoder* and *adpcm_encoder* that the increase in the WCET is less than 25 % and the decrease in the code size is about 60 %. At the same time, for the benchmark *lms* the decrease in code size that is achieved, is just 18.6 %, however, the WCET increases also only by 3 %. The code size of the decompression routine is about 11% of the initial code size for the benchmarks *adpcm_decoder* and *adpcm_encoder*, while for the benchmark *lms* is 23 %.

In the ILP selection model, the decision variables $x_i, i = 1, 2, .., N$ are binary variables. Moreover, as already mentioned we assumed that $C = 1$ in the objective function (5) which is to be minimized. This means that solving the ILP variable x_i can be potentially set

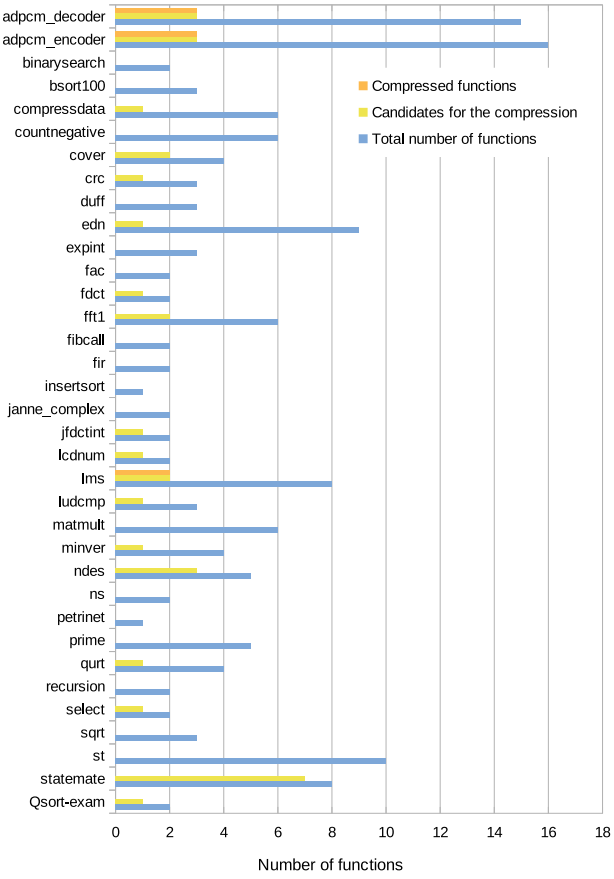

Figure 4: MRTC: Statistics of the functions.

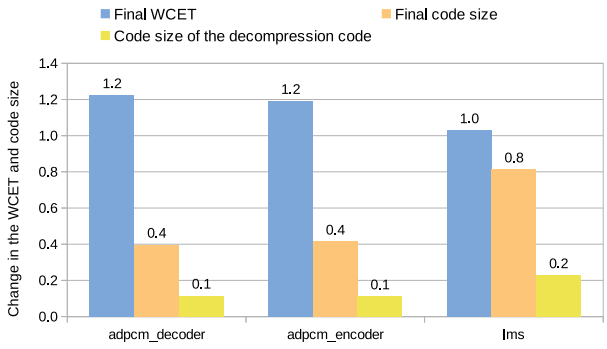

Figure 5: MRTC: Change in the WCET and code size for the selected benchmarks. 1.0 corresponds to the initial WCET and code size.

to 1 only if for the function f_i

$$\text{(decrease in the code size)} \geq \text{(increase in the WCET)}. \quad (12)$$

Otherwise, x_i will always be set to 0 in order to minimize the objective function (5).

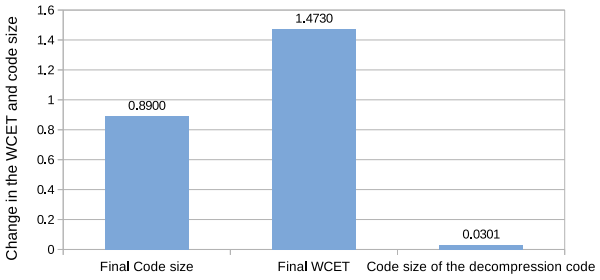

Figure 6: MRTC: Change in the WCET and code size for the benchmark *ndes* with $C = 10$. 1.0 corresponds to the initial WCET and code size.

Table 1: MRTC: Change in the WCET and code size for the function-candidates from benchmark *ndes*.

Benchmark	WCET Increase (%)	Code Size Decrease (%)
ks	25.72	22.22
des	50.85	36.7
cyfun	47.3	27.27

As shown in Figure 4, during the optimization of the benchmark *ndes*, 3 out of 5 functions are selected as candidates for compression in the selection pre-phase. However, after solving the ILP, none of them are finally selected for compression, since as shown in Table 1, for every candidate function the increase in the WCET is higher than the decrease in the code size and inequality (12) is violated. However, if we assume that the decrease in the code size is more important, from the designer point of view, than the WCET increase, then it can be controlled by changing the value of constant C in the objective function (5). For instance, setting $C = 10$ for the benchmark *ndes* leads to the compression of function *cyfun* meaning that by compressing the function we allow the WCET increase be 10 times faster than the code size decrease. However, due to Constraint 1 the WCET still has to be smaller than the WCET limit. As shown in Figure 6 the final WCET increase that is achieved is 47.3 %, while the code size decreases by 11 %. The code size of the decompression routine constitutes 3 % of the initial WCET.

Analogously to MRTC, Figures 7 and 8 show the results for the JetBench benchmarks suite. All benchmarks contain 18 functions without the function *main* which is excluded from consideration by Assumption 3. In every benchmark, 13 functions are selected and finally compressed during the optimization. This leads to the similar results for all 3 benchmarks: the increase in the WCET is only about 0.04 %, while the code size decreases by about 85 % and the code size of the decompression routine is less than 10 % of the initial code size.

MediaBench contains 9 benchmarks in total and in 6 of them, functions are compressed during the optimization as shown in Figure 9. The changes in the WCET and code size for the benchmarks are presented in Figure 10.

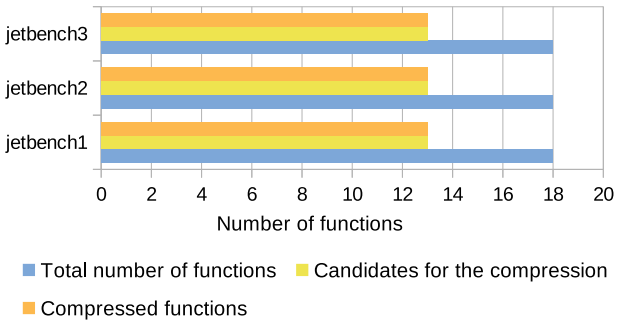

Figure 7: JetBench: Statistics of the functions.

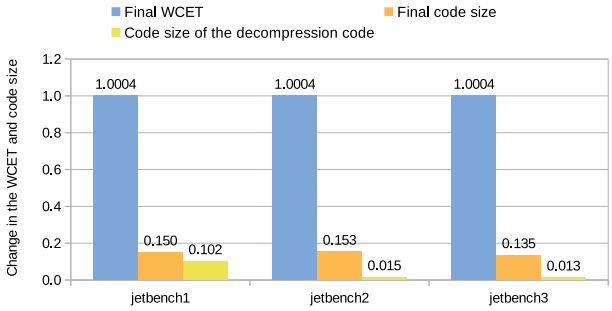

Figure 8: JetBench: Change in the WCET and code size for the benchmarks. 1.0 corresponds to the initial WCET and code size.

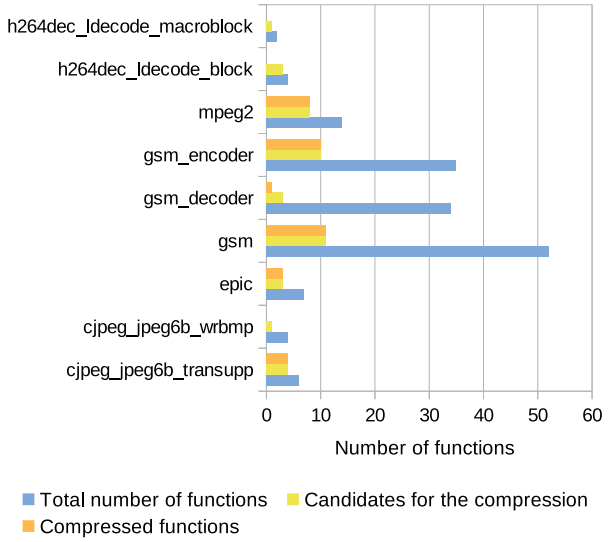

Figure 9: MediaBench: Statistics of the functions.

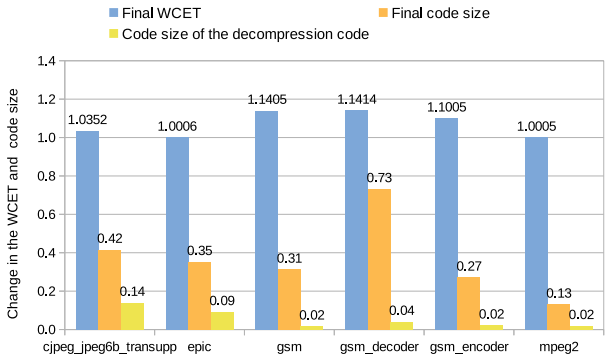

Figure 10: MediaBench: Change in the WCET and code size for the benchmarks. 1.0 **corresponds to the initial WCET and code size.**

Table 2: MediaBench: Change in the WCET and code size for the function-candidates from benchmark *gsm_decoder*.

Benchmark	WCET Increase (%)	Code Size Decrease (%)
gsm_decode	14.14	30.9
Decoding_of_the _coded_Log _Area_Ratios	22.82	21.29
LARp_to_rp	4.72	3.51

For the benchmark *gsm_decoder*, we observe a 14.14 % increase in the WCET, while the code size decreases only by 27 %. According to Figure 9, in this benchmark 3 functions out of 34 are selected as candidates for compression, nevertheless, only one function is finally compressed.

In Table 2 the increase in the WCET and the decrease in the code size for these 3 function candidates are presented. All 3 function candidates cannot be compressed simultaneously, since this would lead to the violation of Constraint 1 in the ILP model. Moreover, only compressing the function *gsm_decode* the code size decreases more quickly than the WCET increases. For the reasons mentioned above only the function *gsm_decode* is selected for compression by solving ILP.

In the benchmark *cjpeg_jpeg6b_wrbmp* according to Figure 9, 1 function out of 4 is a candidate for compression. However, in Figure 11 it is easy to see that even if the function is compressed, the code size will increase due to the fact that the decompression code has to be inserted into the final binary to enable the run-time decompression. Consequently, Constraint 2 of the ILP model is violated and the function is not compressed.

For the benchmarks discussed above, Table 3 lists the initial code size, the size of the compressed data, the run-time of the optimization and the free space of SPM that is needed to decompress all compressed functions during the execution as described in Assumption 2. For the examined benchmarks, Figure 12, shows that as the initial code size grows, the decrease in the code size that can be

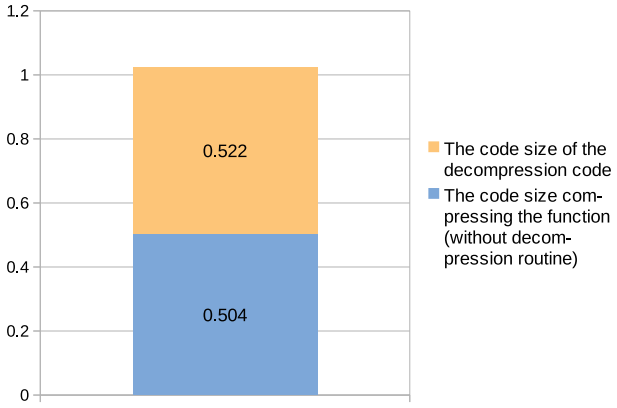

Figure 11: MediaBench: Change in the code size for the benchmark *cjpeg_jpeg6b_wrbmp*. 1.0 **corresponds to the initial code size.**

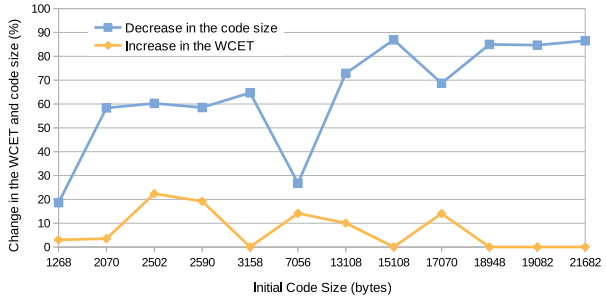

Figure 12: The final code size and WCET depending on the initial code size.

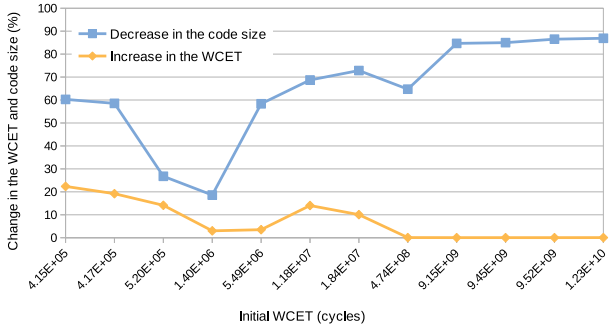

Figure 13: The final code size and WCET depending on the initial WCET.

achieved is higher, while the increase in the WCET is much lower. According to Figure 13, we observe almost the same behavior considering the code size decrease and WCET increase as a function of the initial WCET. However, it should be mentioned that more tests for other benchmark suites are needed to understand the real behavior of the curves discussed above.

Table 3: Statistics of the benchmarks.

Benchmark	Benchmarks Suite	Initial code size (bytes)	Compressed Data (bytes)	SPM size (bytes)	Optimization Runtime (secs)
adpcm_decoder	MRTC	2502	1263	1884	119.265
adpcm_encoder	MRTC	2614	1290	1892	125.564
lms	MRTC	1268	461	582	51.867
jetbench1	JetBench	18948	8967	16886	2260.597
jetbench2	JetBench	19082	8987	16938	2321.138
jetbench3	JetBench	21682	9829	19538	2931.964
cjpeg_jpeg6b_transupp	MediaBench	2070	1405	1614	76.364
epic	MediaBench	3158	1290	3718	107.786
gsm	MediaBench	17070	5920	12346	1255.449
gsm_decode	MediaBench	7056	1040	4014	402.368
gsm_encode	MediaBench	13108	4878	10134	817.846
mpeg2	MediaBench	15108	8495	13656	1086.763

6 CONCLUSION

In this paper, a framework for the WCET-Aware compression of functions has been presented. We demonstrate a compiler-based compression/decompression approach such that the compression takes place at design time in the compiler/linker, but decompression is performed at execution time. An asymmetric compression method, namely FastLZ, was chosen, since it is characterized by the fact that compression is time-consuming but achieves good compression rates, while decompression is fast.

We presented an ILP-based approach for selecting functions for compression. The ILP model describes the restrictions due to the limitations of the target architecture. Moreover, it is shown that the selection process can be controlled by changing parameters in the objective function. The decompression algorithm was adjusted in such a way that it can be analyzed to compute the WCET of the final program at compilation time. Additionally, the WCET of the decompression code was safely estimated.

The experimental results show that a good compression rate with only a little WCET penalty can be achieved for the tested benchmarks suites.

Further investigation will look at developing a model to find the positions in the source code where to perform the code extraction before calling a function. Another improvement is to consider a more sophisticated method to choose functions for the compression. The problem is multi-objective and appropriate methods could be utilized to discover a wider variety of solutions. Furthermore, additional criteria could be considered, e.g., energy consumption. Moreover, the current approach can be also applied at other levels of abstractions, e.g., to consider basic blocks instead of functions as candidates for compression.

ACKNOWLEDGMENTS

This work received funding from Deutsche Forschungsgemeinschaft (DFG) under grant FA 1017/3-1.

REFERENCES

[1] AbsInt Angewandte Informatik, GmbH. 2018. aiT Worst-Case Execution Time Analyzers.

[2] Á. Beszédes, R. Ferenc, T. Gyimóthy, A. Dolenc, and K. Karsisto. 2003. Survey of Code-size Reduction Methods. *ACM Comput. Surv.* 35, 3 (2003), 223–267.

[3] W. R. A. Dias and E. D. Moreno. 2012. Code Compression in ARM Embedded Systems Using Multiple Dictionaries. In *CSE 2012*. 209–214.

[4] H. Falk, S. Altmeyer, P. Hellinckx, B. Lisper, W. Puffitsch, C. Rochange, M. Schoeberl, R. B. Sorensen, P. Wägemann, and S. Wegener. 2016. TACLeBench: A Benchmark Collection to Support Worst-Case Execution Time Research. In *WCET 2016*.

[5] H. Falk and P. Lokuciejewski. 2010. A Compiler Framework for the Reduction of Worst-Case Execution Times. *Real-Time Systems* 46, 2 (2010), 251–298.

[6] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. 2010. The Mälardalen WCET Benchmarks: Past, Present And Future. In *WCET 2010*, Vol. 15. 136–146.

[7] E.S. Helan, P.M. Sandeep, V.Suresh Babu, and M. Varatharaj. 2017. Compression and Decompression of Embedded System Codes. *International Journal of Engineering Trends and Technology* 45, 7 (2017), 325–330.

[8] A. Hidayat. 2007. FastLZ - lightning-fast lossless compression library. http://www.fastlz.org

[9] Synopsys Inc. Online. CoMET System Engineering IDE. http://www.synopsys.com

[10] Raimund Kirner. 2003. *Extending Optimising Compilation to Support Worst-Case Execution Time Analysis*. Ph.D. Dissertation. Technische Universität Wien, Treitlstr. 3/3/182-1, 1040 Vienna, Austria.

[11] C. Lee, M. Potkonjak, and H. Mangione-Smith. 1997. MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems. In *Micro 1997*.

[12] A. Luppold and H. Falk. 2017. Schedulability-Aware SPM Allocation for Preemptive Hard Real-Time Systems with Arbitrary Activation Patterns. In *DATE 2017*. 1074–1079.

[13] K. Muts, A. Luppold, and H. Falk. 2018. Multi-Criteria Compiler-Based Optimization of Hard Real-Time Systems. In *SCOPES 2018*. 54–57.

[14] D. Oehlert, A. Luppold, and H. Falk. 2017. Bus-aware Static Instruction SPM Allocation for Multicore Hard Real-Time Systems. In *ECRTS 2017*. 1:1–1:22.

[15] H. Ozaktas, K. Heydemann, C. Rochange, and H. Cassé. 2009. Impact of Code Compression on Estimated Worst-Case Execution Times. In *RTNS 2009*. 55–66.

[16] O. Ozturk, G. Chen, M. Kandemir, and I. Kolcu. 2006. Compiler-guided data compression for reducing memory consumption of embedded applications. In *ASP-DAC 2006*. 6 pp.–.

[17] S. S. Pinter and I. Waldman. 2007. Selective Code Compression Scheme for Embedded Systems. In *Transactions on High-Performance Embedded Architectures and Compilers I*. 298–316.

[18] M. Y. Qadri, D. Matichard, and K. D. McDonald Maier. 2010. JetBench: An Open Source Real-time Multiprocessor Benchmark. In *ARCS 2010*. 211–221.

[19] M. Ros and P. Sutton. 2003. Compiler Optimization and Ordering Effects on VLIW Code Compression. In *CASES 2003*. 95–103.

[20] M. Roth, A. Luppold, and H. Falk. 2018. Measuring and Modeling Energy Consumption of Embedded Systems for Optimizing Compilers. In *SCOPES 2018*. 86–89.

[21] M. Thuresson, M. Själander, and P. Stenstrom. 2009. A Flexible Code Compression Scheme Using Partitioned Look-Up Tables. In *High Performance Embedded Architectures and Compilers*. 95–109.