Impact of Code Compression on Estimated Worst-Case Execution Times

Haluk Ozaktas, Karine Heydemann, Christine Rochange, Hugues Cassé

In Proc. of the 17th International Conference on Real-Time and Network Systems
RTNS'2009, Paris, ECE, 26-27 October, 2009

Impact of Code Compression on Estimated Worst-Case Execution Times

Haluk Ozaktas[b], Karine Heydemann[b], Christine Rochange[a], Hugues Cassé[a]

[a] IRIT – Université de Toulouse
118 route de Narbonne
31062 Toulouse cedex 9, France
{rochange, casse}@irit.fr

[b] LIP6 – Université de Paris VI
4, place Jussieu
75252 Paris cedex 5
{haluk.ozaktas, karine.heydemann}@lip6.fr

Abstract

Code compression techniques might be useful to meet code size constraints in embedded systems. In the average case, the impact of code compression on the performance is double-edged: on one side, the number of accesses to memory hierarchy is reduced because several instructions are coded in a single word, and this is likely to reduce the execution time; on the other side, the decompression penalty increases the processing time of compressed instructions. Nevertheless, experimental results show that the execution time might be lowered by code compression.

In this paper, our goal is to analyze 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. Changes in the access patterns to the instruction cache are indeed likely to alter the accuracy of the cache analysis within the process of determining the WCET.

Experimental results show that, besides reducing the code size, our code compression scheme also improves the WCET estimates in most of the cases.

1. Introduction

Embedded systems are often constrained in terms of different criteria like code size, execution time or energy consumption. Various techniques have been proposed to improve one of these criteria: code compression schemes aim at reducing the code size, compiler optimizations help in improving the execution time and various approaches determine the best placement of instructions and data in the memory space to limit the energy requirements.

However, the impact of the techniques that improve one of the criteria onto the other ones is seldom analyzed. It is the goal of the French MORE[1] project to get insight into such board effects and to determine sets of code transformations that jointly improve several criteria.

In this paper, we focus on the impact of code compression techniques on the execution time and more particularly on the Worst-Case Execution Time (WCET) of time-critical software.

Code compression reduces the code size by compacting the original code into a non executable format. At runtime, a decompression step is needed to retrieve the initial code. Code compression has been and is still an active research area [5][24]. In this paper, we consider a compression scheme that combines two state-of-the-art approaches [7][21]. A dictionary-based compression algorithm is used to replace sequences of instructions by special instructions that trigger decompression at runtime. Decompression takes place in the processor pipeline, between the fetch and decode stages. This is compatible with wide-issue high performance superscalar architectures, contrary to post-cache decompression, while still leaving compressed code in the instruction cache. Thus, the number of cache misses is reduced which might improve both the execution time and the energy consumption.

To estimate Worst-Case Execution Times, we consider state-of-the-art techniques: the worst-case execution costs of basic blocks are computed using parameterized execution graphs [22], the behavior of the instruction cache is analyzed using abstract interpretation [1][3] and an upper bound of the whole program execution time is derived using the IPET method [19]. All these algorithms can be invoked within the OTAWA framework [6] and have been adapted for this study to take into account the effects of code compression: the execution costs of basic blocks include decompression penalties and the instruction cache analysis considers the instruction addresses in the compressed code.

The paper is organized as follows. Section 2 gives an overview of code compression techniques and details the algorithm considered in this study. Section 3 presents the strategy used to estimate WCETs and discusses the expected (theoretical) impact of code compression on the

[1] MORE stands for Multicriteria Optimization for Real-time Embedded systems. This project is supported by the ANR French National Agency for Research.

accuracy of the estimates. The methodology for experiments is detailed in Section 4 and experimental results are provided and analyzed in Section 5. Section 6 concludes the paper.

2. Code compression

2.1. State-of-the-art

Code compression has been and remains a hot topic [5][24]. The proposed approaches differ in the compression strategy (statistical as Huffman coding, dictionary-based or any combination of both) as well as in the implementation (by software or in hardware) and in 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.

A pre-cache decompression engine is only invoked on cache misses: decompression operations are then less frequent than for post-cache schemes but, since the cache contains original (uncompressed) code, the decompression time penalty cannot be balanced by a reduced number of cache misses. This makes it necessary to trade-off between the code size improvement and the execution time degradation [18]. IBM's Code Pack is an example of pre-cache dictionary-based compression scheme used in some processors of the PowerPC family [16]. Every half-word of a cache line is encoded using a variable-size encoding word. On an instruction cache miss, two compressed cache lines are decompressed and fetched into the cache. For some programs, this might act as a prefetch and balance the decompression timing overhead [18].

Besides reducing the size of the code, compression can also improve the performance and reduce the energy requirements if the compressed code is stored in the instruction cache [21]. However, since post-cache decompression is done on the critical path and is potentially needed on every access to the cache, it must be fast to avoid increasing the processor cycle time or the cache access time. This approach requires coping fast with two addressing spaces, one related to the compressed code and the other one seen by the processor for which the code compression is completely transparent [14][21]. Moreover, post-cache decompression is very hard to implement for superscalar processors and might impair the efficiency of a branch predictor.

Decompression can also be done within the pipeline: it is then very close to the translation engine for micro-coded instructions [7]. This is the solution that we have considered in this paper since it suits any superscalar architecture and avoids handling two address spaces.

Another approach to reduce the size of binary codes consists in using shorter instructions. Some processors support dual-width instruction sets: 16-bit instructions can be used to limit the code size while 32-bit instructions might be preferred to fit performance requirements. The ARM Thumb is the best known example of dual-width instruction sets [11]. The translation of 16-bit instructions into 32-bit codes is immediate in the decode stage. A binary code that uses 16-bit Thumb instructions is typically smaller by 30% than regular code and suffers longer execution times due to the limited expressiveness of 16-bit instructions. To limit the performance degradation, the most frequently executed code regions are usually compiled with 32-bit instructions while less frequently executed regions are compiled with 16-bit instructions.

Code compression techniques are orthogonal to the use of reduced instruction sets since, besides shortening the instruction codes, they exploit their redundancy.

Earlier works report a mean reduction of the code size by 20% with dictionary-based approaches and performance and energy gains that vary according to the applications and to the cache sizes.

As far as we know, the only paper on reducing the code size for real-time applications focuses on the use of a 16-bit instruction set [17]. It shows that it is necessary to trade-off between the reduction of the code size and an increase of the Worst-Case Execution Time. The proposed strategy then consists in limiting the use of 16-bit instructions to code regions that have a little impact on the overall WCET, so that it is not too much degraded.

In this paper, we show that the code compression technique that we considered can improve both the code size and the WCET of hard critical software.

2.2. Compression scheme

In the MORE project, we decided to use a post-cache code compression technique that is likely to optimize at the same time the code size and the energy consumption. Since our intention is to consider high-performance processors, we have opted for in-pipeline decompression that, in addition, avoids the complexity of handling different address spaces. Since the decompression overhead was critical, we designed dictionary-based compression scheme that might be less efficient (in terms of compression rate) than statistical algorithms but that allows faster decompression.

In our solution, the dictionary contains full instructions. In order to limit the cost of the dictionary and to keep its access time short, it is desirable to restrict its size. Keeping the dictionary small is also necessary to limit the width of the dictionary index (*log(n)* bits are required for an *n*-entry dictionary), which is important to insure the efficiency of the code compression scheme: the smaller the index width, the better the compression rate. Moreover, a dictionary does not need to hold all

the instructions that appear in the code: when an instruction in the dictionary appears only once in the code, the code size is not improved and even degraded [4] (since the instruction is stored twice: once in the code, in a compressed form, and once in the dictionary).

As far as the dictionary does not hold all the instructions, the compressed code contains both compressed and uncompressed instructions. For our compression scheme design, we have fixed the dictionary size to 256 entries, which is a standard size for hardware implementation and one-cycle decompression [12][21]. Besides, this size allows covering a significant part of the static code and reaching good compression rate even with large applications (the most redundant instructions are generally not numerous).

Our compression scheme replaces two or three successive instructions present in the dictionary by one 32-bit encoding instruction (ISA-width encoding avoid alignment issues). This encoding instruction is composed of an invalid code operation of the target ISA, two information bits and three 8-bit slots that contain the index of the dictionary entries that store the corresponding instructions. This is illustrated in Figure 1. Absolute branch instructions can be included in the dictionary by patching them afterwards. Relative jumps can also be included if the jump displacement is nullified and the patched relative value is encoded into the encoding instruction.

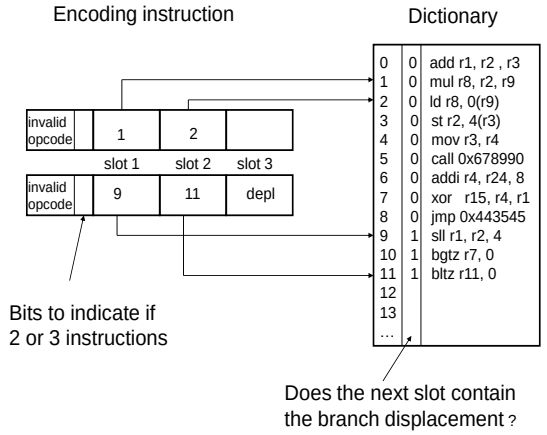

Figure 1. Encoding instructions

The main issue of a dictionary-based compression scheme is how the dictionary is built. To maximize code size reduction, it is preferable to include the most statically repeated instructions whereas selecting the most executed instructions favours the reduction of the number of instruction cache misses. To benefit from both code size and cache miss rate improvement, our compression scheme builds *P*% of the dictionary with the most executed instructions and fills the remaining entries with the most statically repeated instructions. Once the dictionary is built, sequences of instructions that are in the dictionary are encoded. To avoid impairing branch prediction, only instructions that belong to the same basic block can be encoded together.

2.3. Decompression

Decompression is done in the processor pipeline. A decompression stage must be added except if the processor already has a stage for translation of micro-coded instructions into instructions as in the Intel IA-32 architecture. The decompression stage is placed between the fetch and the decode stages. Non-compressed instructions are simply forwarded to the decode stage. In case of a compressed instruction, extra cycles are needed to access the dictionary. As the dictionary is much smaller and less complex than a cache, a one-cycle access is feasible. The dictionary access fills the pipeline with two or three new instructions depending on the number of instructions encoded into a single one.

3. WCET analysis

3.1. General overview

The estimation of Worst-Case Execution Times (WCETs) usually includes three steps: the *flow analysis* determines flow facts like loop bounds and infeasible paths [2][9][10][13][15]; the *low-level* analysis computes the worst-case execution costs of basic blocks taking into account the specifications of the target hardware [20][22][23]; and finally the WCET computation combines the flow facts and the execution costs to find out the longest path and its execution time [19].

The low-level analysis step is in turn split into two sub-steps: the first one examines the behavior of history-based components (mainly the instruction and data caches) and the second-one computes the execution cost of each basic block when executed in the pipeline.

Since code compression has no impact on flow facts, we focus, in this paper, on the low-level analysis.

3.2. Instruction cache analysis and computation of execution costs

Instruction cache analysis. The most popular technique to analyze the behavior of the instruction cache is based on the determination of Abstract Cache States (ACS): an ACS is the set of concrete cache states that are possible at a given point in the Control Flow Graph (CFG) during the execution of the program [1]. It associates a set *s* of possible *l*-blocks[2] to each cache line.

[2] An *l*-block results from the projection of the CFG on the cache line map: a cache line that contains instructions belonging to *n* different basic blocks is considered as *n* *l*-blocks.

Abstract interpretation techniques [8] are used to compute abstract cache states in input and output of each basic block. The *Update* function computes the output ACS of a basic block from its input ACS, and the *Join* function merges the output ACS of all the predecessors of a basic block to produce its input ACS. The *Update* and *Join* functions are applied repeatedly until the algorithm reaches a fixed point.

This process is applied to *May* and *Must* analyses that determine the set *s* of *l*-blocks that *may* (resp. *must*) be in the cache at each program point. A third analysis, called *Persistence* analysis, is used to detect *l*-blocks that belong to a loop body and remain in the cache between successive iterations (but might miss at the first iteration).

Finally, the results of the *May*, *Must* and *Persistence* analysis are used to assign a category to each *l*-block among: *Always Hit* (each fetch is guaranteed to hit in the cache), *Always Miss* (each fetch is guaranteed to miss), *Not Classified* (the analysis is not able to predict a fixed issue for this fetch) and *Persistent* (the fetch misses each time the heading loop is entered and hits while the loop iterates).

Execution cost computation. The execution cost of a basic block also depends on the history. The possible states of the pipeline when the block starts executing can be determined using abstract interpretation techniques, as in [23]. However, to keep the analysis cost (as well in terms of memory space as in terms of computation time), we have developed another technique that considers any possible pipeline state without enumerating them one by one [22]. It is based on execution graphs that express the data, control and structural dependencies between instructions and computes the possible instruction schedules as a function of the state of the pipeline when the block starts executing. From these possible schedules, an upper bound of the execution cost is derived. This technique is much faster than the one that uses abstract interpretation, at the cost of a limited loss of accuracy.

Integration of cache miss penalties in execution costs. The instruction cache and the pipeline are often analyzed in a totally decoupled manner: the block execution costs are estimated considering cache hits and a penalty is added for each possible miss detected by the instruction cache analysis. While very convenient, this approach is not safe when the processor has not been proved "timing-anomaly-free". The term of "timing anomaly" refers to situations where, by example, an increase of the latency of an instruction by *i* cycles leads to an increase of the block execution time by more than *i* cycles. As far as the instruction cache is concerned, this means that the block execution cost with a cache miss might be shorter than with a cache hit.

It is generally hard to prove that a processor is not prone to timing anomalies. In this case, a safe approach is to compute the possible costs of each basic block considering all the possible cache behaviours (for all the instructions of the block). As said before, *l*-blocks that have been classified as *Always Hit* or *Always Miss* have a fixed latency, while those labelled as *Not Classified* can experience either a hit or a miss latency. Thus, for the latter, both latencies must be considered when computing the block cost which means that, if *n l*-blocks in the basic block are *Not Classified*, as many as 2^n costs must be evaluated (and the maximum value is kept). Fortunately, cache analysis is usually accurate enough to limit the number of *Not Classified l*-blocks. *Persistent l*-blocks might undergo a miss latency when the heading loop is entered and always hit when the loop iterates. Again, both cases must be considered and two block costs must be computed: one for each entrance into the loop, and one for the other iterations. If *n l*-blocks in the block are *Persistent*, they generally have the same heading loop and then exhibit the same behaviour (they all hit or all miss). Then only two costs have to be computed for all these instructions.

This can be illustrated considering the example given in Figure 2. In this example, basic block b_j contains six *l*-blocks that belong to different categories. Three *l*-blocks are *Persistent* with two different headers. For this basic block, eight cost values must be computed. They are listed in Table 1 ('H' stands for *hit* and 'M' for *miss*). For *Persistent* and *Not Classified l*-blocks, both cases (hit and miss) must be considered. When two *l*-blocks are *Persistent* with the same header (lb_3 and lb_4), they must have the same behaviour.

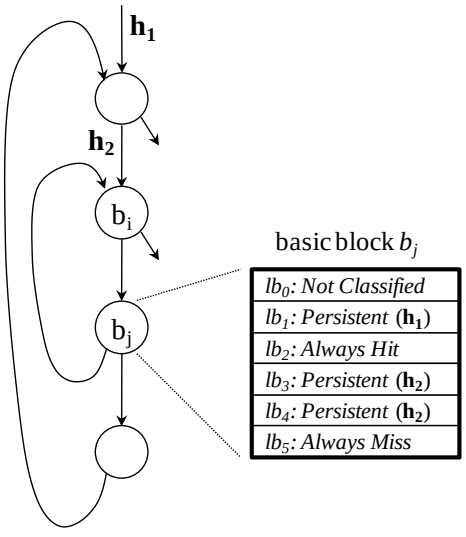

Figure 2. Example.

cost value	considered behaviours					
	lb_0	lb_1	lb_2	lb_3	lb_4	lb_5
$C^{[0]}$	H	H	H	H	H	M
$C^{[1]}$	H	H	H	M	M	M
$C^{[2]}$	H	M	H	H	H	M
$C^{[3]}$	H	M	H	M	M	M
$C^{[4]}$	M	H	H	H	H	M
$C^{[5]}$	M	H	H	M	M	M
$C^{[6]}$	M	M	H	M	M	M
$C^{[7]}$	M	M	H	M	M	M

Table 1. Possible cache behaviors for basic block b_j of Figure 2.

3.3. Expected impact of code compression on estimated WCETs

The decompression penalty of compressed instructions must be taken into account when estimating block costs. It is expected to have an impact equivalent to the one it has on the observed execution time.

In addition, code compression is likely to have an impact on the results of the instruction cache analysis. The reason for this is that it is expected to alter the number of *l*-blocks in the program as well as their size.

Figure 3 is a reminder of how *l*-blocks are built: in this example, basic block *b* has three *l*-blocks, one that we describe as *full* since it corresponds to a complete cache line and two that we describe as *partial* since they share their cache lines with other *l*-blocks that belong to basic blocks *b*-1 *and* b+1. Each basic block contains *f* full *l*-blocks, where *f* can take any value, including zero, and *p* partial *l*-blocks with *p* in {0, 1, 2}. The number of full *l*-blocks depends on the basic block length and the number of partial *l*-blocks depends on its alignment with respect to cache line boundaries.

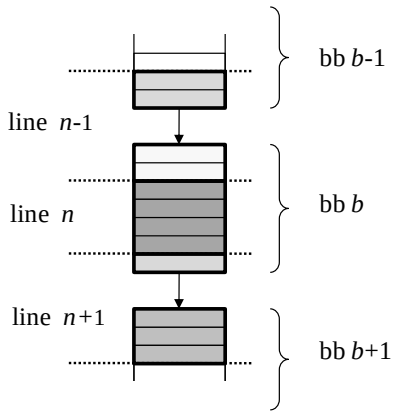

Figure 3. Construction of *l*-blocks.

Let us now discuss what can change when the code is compressed. Figure 4 shows the possible impact on the example code of Figure 3. Here, we assume that several instructions are compressed. As a result, the length of basic block *b* is decreased from 7 to 3 instructions and it has now a single (partial) *l*-block. More generally, code compression shortens the basic blocks and is then likely to reduce their number of full *l*-blocks. The impact on the number of partial *l*-block is less predictable since it depends on the alignment to cache line boundaries.

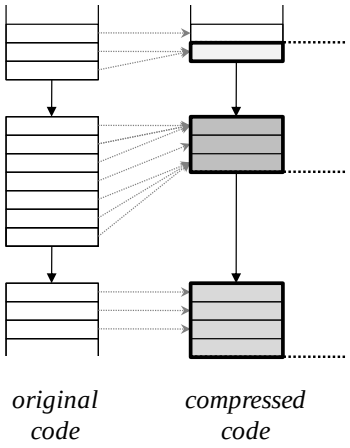

Figure 4. Construction of *l*-blocks in the compressed code.

Now, how these changes on the number and size of the *l*-blocks might impact the cache-related contribution to the WCET?

First, a smaller number of *l*-blocks means a smaller number of accesses to the instruction cache, and this is prone to reduce the Worst-Case Execution Time (as well as the average-case execution time).

Second, an increase of the proportion of partial *l*-blocks might change the distribution into categories. On one hand, partial *l*-blocks are more likely to *Always Hit* than full *l*-blocks since a cache line that contains the beginning of a basic block might have been fetched on the execution of the previous basic block that shares the cache line. In other words, partial *l*-blocks benefit from spatial locality. On the other hand, partial *l*-blocks are prone to generate inaccuracy in the cache analysis: it often cannot be determined whether an *l*-block will hit or miss in the cache when the basic block it belongs to has several possible predecessors. As a result, partial *l*-blocks are prone to be annotated as *Not Classified*.

To conclude, it is difficult to predict whether code compression will improve or degrade the WCET. It might improve it because it reduces the number of accesses to the instruction cache and because the proportion of remaining *l*-blocks classified as *Always Hit* is likely to increase. On the contrary, it might degrade the WCET because the proportion of *Not Classified* *l*-blocks should increase. The goal of this study is to decide between these two possibilities through an experimental approach.

4. Methodology

4.1. Implementation of code compression and WCET analysis

All the techniques involved in this study have been implemented within the OTAWA framework [6]. OTAWA comes as a library that provides a series of classes and tools used for WCET analysis.

Our code compression scheme has been implemented within OTAWA. Two new *Code Analysis* passes have been developed: the first one scans the binary code to compute the frequency of static instructions and the second one simulate the program execution to determine the dynamic frequency of the instructions. These passes have been complemented with a *Dictionary Builder* that is parameterized by the proportion P of the dictionary that must be filled with the instructions that exhibit the highest dynamic frequency. Finally, we implemented the *Code Compressor* that tries to build as many full (i.e. including three original instructions) encoding instructions as possible, while respecting basic block boundaries. The computation of the addresses in the compressed code, including the branch targets, is done after the encoding.

In order to avoid the cost of developing a compressed code generator and a compressed code loader for WCET analysis, our code compression algorithm annotates the Control Flow Graph of the program under analysis to indicate which instructions are compressed and what their addresses in the compressed code are. Then the same CFG can be used to analyze both the original and the compressed codes.

To handle compressed code, both the execution cost computation part (building of execution graphs) and the *l*-block builder have been modified to consider the addresses of compressed instructions.

OTAWA also includes a cycle-level simulator built on the SystemC library. This simulator has been modified to include the decompression engine.

4.2. Experimental procedure

So far, OTAWA is not able to consider several cost values for each basic block, related to the different possible cache behaviors found by the preliminary cache analysis. It considers instead a single cost value which is the maximum of all the computed values for the basic block.

In order to obtain results that correctly reflect the accuracy of the cache analysis, we have decided to compute estimated WCETs from flow information determined by profiling. The program under analysis is simulated and the execution count $x_{i,j}$ of each two-block sequence b_i-b_j is observed. Then what we refer to as the WCET in this paper is estimated as:

$$WCET = \sum_{s_{i,j} \in S} \left(x_{i,j} - \max_{h \in H_{i,j}}(x_h) \right) . c_{i,j}^{>1} + \max_{h \in H_{i,j}}(x_h) . c_{i,j}^{=1}$$

where S is the set of possible two-block sequences, $c_{i,j}^{=1}$ is the maximum cost of block b_j in sequence b_i-b_j, computed considering that the *l*-blocks of b_j that have been classified as *Persistent* miss (first loop iterations) and $c_{i,j}^{>1}$ the maximum cost computed when they hit (other iterations). These maximum cost values are estimated considering both possibilities for all the *Not Classified* *l*-blocks. The set of loop header edges related to the *Persistent* *l*-blocks is denoted as $H_{i,j}$ and x_h is the execution count observed for an header edge. Whenever the block contains several *Persistent* *l*-blocks with different headers, a maximum value is computed considering all the possible cost values, which is likely to generate overestimation. Fortunately, this case is rather infrequent. When the block does not contain any *Persistent* *l*-block, max(x_h) is null.

To illustrate this formula, let us consider the example given in Figure 2 and Table 1. In this example, the contribution of basic block b_j to the estimated WCET would be computed as:

$$C_{i,j}^{=1} = \max(C_{i,j}^{[1]}, C_{i,j}^{[2]}, C_{i,j}^{[3]}, C_{i,j}^{[5]}, C_{i,j}^{[6]}, C_{i,j}^{[7]})$$

$$C_{i,j}^{>1} = \max(C_{i,j}^{[0]}, C_{i,j}^{[4]})$$

This way, we estimate the Worst-Case Execution Time *related to the flow facts obtained by profiling*. For some of the benchmarks, the input data really drive the execution on to the longest path. For other ones, we were not able to determine the worst-case input data and thus the profiled execution path might not be the worst-case path. Nevertheless, we estimated the WCET for this path which makes sense since our goal is to analyze the accuracy of the cache and pipeline analysis, not that of the flow analysis.

4.3. Benchmarks

For the experiments, we used the benchmarks listed in Table 2. Most of them come from the collection hosted on the Mälardaalen University website [26], which is often used for WCET analysis experiments. The `seg` code, that we have developed, implements well-known algorithms, includes three functions that are considered as three benchmarks (but reside in the same executable file): `seg1` corresponds to the function that finds regions of adjacent similar pixels in the image, `seg2` refers to the function that fuses adjacent regions and `seg3` relates to the function that fuses pixels that belong to fused regions. We also have developed the `airbag` benchmark that implements the algorithms described in [25].

4.4. Processor architecture and cache configuration

Since we were mainly interested in the effects of code compression on the analysis of the cache instruction, we have considered a simple pipeline configuration: two-way superscalar, with in-order execution, no branch prediction and a perfect data cache (i.e. all the accesses to data hit in the cache).

We have considered several instruction cache configurations with a cache line size of 16 or 32 bytes and a cache size ranging from 128 to 2048 bytes (to get realistic results for small benchmarks). In all cases, the instruction cache has been considered as 4-way set associative.

adpcm	Adaptative Differential Pulse Code Modulation
crc	Cyclic Redundancy Check
compress	Data compression
matmul	Matrix multiplication
nsischneu	Simulation of a Petri net
seg1, seg2, seg3	Image segmentation (3 steps)
airbag	Airbag control software

Table 2. Set of benchmarks.

4.5. Code compression

Code compression is parameterized by the proportion of the dictionary that is built from dynamic instruction profiles instead of static code information. In a preliminary study, we have found that, for most of the benchmarks, a value of P=75% limits the degradation of the reduction in code size (compared to *P=0*) while increasing the quantity of compressed code fetched into the cache at runtime (which is maximum when *P=100%*). This way, code compression is effective while also improving the execution time and the energy consumption. This is why we considered *P=75%* in our experiments.

5. Experimental results

5.1. Impact of code compression on the code size and on the observed execution time

Let us first examine how the code compression scheme is efficient in reducing the code size. Table 3 gives the compression rate for all the benchmarks considered in this paper: the first column indicates the raw compression rate of the text section while the second column accounts for the dictionary data into the compressed code size (these data might be included in the executable file and loaded into the dictionary before starting the execution). Since we do not analyze the execution time of the whole codes, but only that of the main function, we report in Table 4 the code size reduction of this function (this ignores the prologue and epilogue as well as unreached library functions). Sizes are given in bytes.

Benchmark	Compression rate	Compression rate including dictionary cost
adpcm	19.2%	9.5%
crc	24.5%	10.4%
compress	22.1%	10.1%
nsischneu	29.1%	18.4%
seg(1,2,3)	18.5%	11.3%
airbag	31.8%	25.1%

Table 3. Code size reduction of the whole benchmarks

Benchmark	Size of original code	Size of compressed code	Compression rate
adpcm	4 040	3 164	21.7%
crc	656	308	53.0%
compress	1 820	1 212	33.4%
nsischneu	3 092	1 744	43.6%
seg1	1 052	1 020	3.0%
seg2	1 600	1 564	2.3%
seg3	972	932	4.1%
airbag	9 076	5 196	42.8%

Table 4. Code size reduction of the analyzed functions

Now, as said before, the impact of code compression on the execution time is hard to predict because the penalty due to the decompression scheme might be balanced by the gain due to a lower number of accesses to the instruction cache. Figure 5 shows the variation in the observed execution time when the code is compressed. The different sets of bars relate to different cache configurations. For most of the benchmarks, the execution time is sometimes noticeably decreased, in particular for small caches and small cache lines.

These results can be explained considering the impact of code compression on the number of instruction cache misses per instructions. For some of the benchmarks (`adpcm`, `seg1`, `seg2` and `seg3`) and cache configurations (larger than 512 bytes for `crc`, `compress` and `nsischneu`), the number of misses per executed instruction in the original code is very low, as shown in Table 5. This means that cache misses do not contribute much to the execution time. As a consequence, the reduction in cache misses due to code compression improves only slightly the execution time. Note that the execution time is even increased for `seg3` with cache

configuration 32-128: this is due to the fact that the number of accesses to the cache is unexpectedly increased by code compression, which might be due to changes into the alignment of the code with respect to cache line boundaries.

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
adpcm	0.7%	0.7%	0.3%	0.4%	0.4%	0.2%
crc	7.9%	0.3%	0.2%	6.3%	0.2%	0.1%
compress	27.4%	1.9%	1.9%	18.6%	1.0%	1.0%
nsischneu	17.4%	4.5%	4.5%	16.9%	2.3%	2.3%
seg1	2.0%	1.4%	0.2%	1.2%	0.8%	0.1%
seg2	0.5%	0.2%	0.1%	1.0%	0.1%	0.0%
seg3	1.6%	0.5%	0.5%	6.1%	0.3%	0.3%
airbag	25.8%	7.9%	4.8%	5.8%	4.3%	2.5%

Table 5. Mean number of instruction cache misses per executed instruction in the original code

On the contrary, when the original code exhibits a significant number of cache misses per instruction (which is the case with 128-byte cache configurations for `airbag`, `crc`, `compress` and `nsischneu`), the decrease of the number brought by code compression is larger.

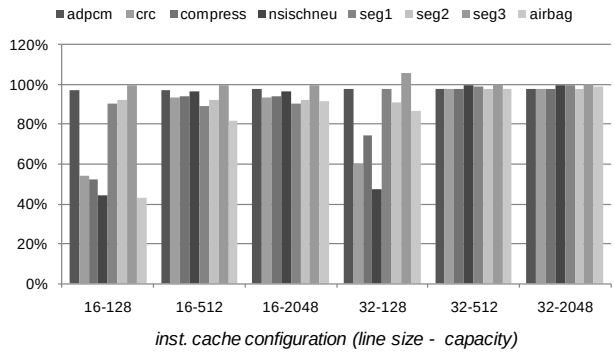

Figure 5. Impact of code compression on the observed execution time

The results above show that code compression, while mainly intended to reduce the code size, also improves the execution time in the average case. In the following section, we will check whether this is still true in the worst-case.

5.2. Impact of code compression on the Worst-Case Execution Time

The impact of code compression on the estimated Worst-Case Execution Time is shown in Figure 7 and Table 6 compares the average variation of the observed execution time (over all the benchmarks) to that of the WCET. On a mean, code compression improves the WCET less than the observed execution time for small caches and more than the observed execution time for larger caches. This respectively corresponds to a decrease or an increase of the WCET estimation accuracy. However, the impact in the WCET is noticeably different from one benchmark to the other one.

To validate the hypothesis that the lower improvement on the WCET than on the observed execution time is due to a loss of accuracy in the cache analysis, we have carried out some experiments considering perfect (always-hit) instructions caches. They showed that the WCET of the compressed code is almost the same as that of the original code, for every benchmark and cache configuration. This confirms that code compression sometimes impairs the instruction cache analysis.

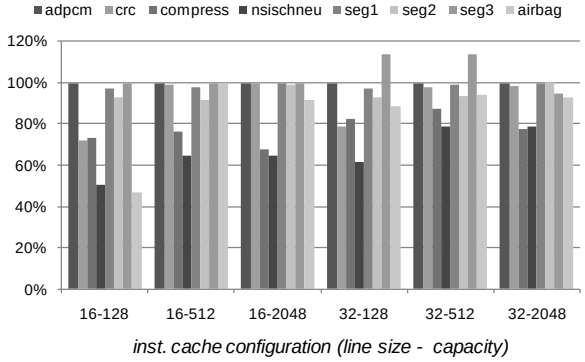

Figure 6. Impact of code compression on the WCET

	cache size (bytes)					
	line = 16 bytes			line = 32 bytes		
	128	512	2048	128	512	2048
observed	-26.0%	-7.0%	-5.6%	-16.0%	-1.5%	-1.2%
WCET	-19.2%	-8.6%	-8.7%	-10.0%	-4.4%	-6.4%

Table 6. Impact of code compression on the mean observed and worst-case execution times

As mentioned in Section 3.3, code compression has an impact on the profiles of *l*-blocks. The curves in Figure 7 show that the rate of *partial l*-blocks is significantly increased for most of the benchmarks. The increase is greater with 16-byte cache lines because the proportion of partial *l*-blocks is already high in the original code with 32-byte lines (many basic blocks are shorter than 8 instructions).

Benchmarks that have few instruction cache misses per instruction do not see their estimated WCET much improved by code compression (in the same way as their observed execution time is not impacted). This is the case of `adpcm`, `seg1`, `seg2` and `seg3`.

Other benchmarks, like `compress` and `nsischneu` have their estimated WCET improved by code compression while their observed execution was not impacted (larger cache configurations). A look at the *l*-block categories for `nsischneu` reveals that the number of *Always Miss l*-blocks is cut by about 45% in the compressed code compared to the original code, which reflects that the spatial locality is improved. At the same time, the number of *Not Classified l*-blocks is cut by 50% to 70% (depending on the cache configuration) and this significantly helps the accuracy of WCET estimation. This explanation also holds for `compress`.

Finally, two benchmarks, `crc` and `airbag`, exhibit a reduction of their WCET mainly for small cache configurations.

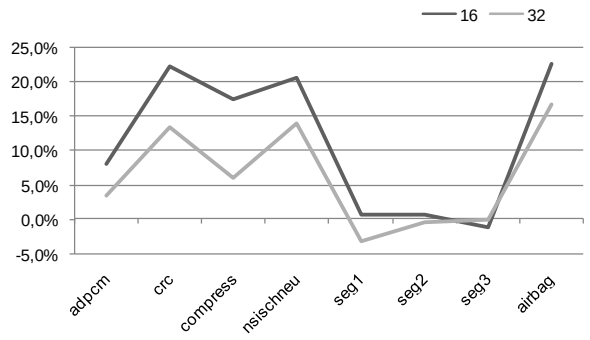

Figure 7. Impact of code compression on the number of *partial l*-blocks

To sum up, our code compression scheme tends to improve the accuracy of WCET estimates through an increased precision of the cache analysis. It can then beneficially be implemented in systems subjected both to time and memory size constraints.

6. Conclusion

Embedded systems often have to meet constraints of different nature: time deadlines for real-time applications, limitation of code size related to low memory capacity and restrictions on energy consumption imposed by requirements on autonomy and low power dissipation.

The goal of the MORE project is to develop a framework to optimize embedded software with respect to two or three of these criteria (WCET, code size and energy consumption) at the same time.

In this paper, we focus on the impact of code compression, techniques used to reduce the code size on the Worst-Case Execution Time.

The impact of code compression on the average-case execution time could be two-edged: on one side, the number of accesses to the instruction cache and the number of cache misses are likely to be reduced; on the other side, the overhead of decompression might increase the execution time. Thanks to the use of an in-pipeline decompression engine, the decompression time penalty is hidden by pipelined execution. This is why experiments show an improvement of the observed execution time besides to the reduction of the code size.

The impact on the Worst-Case Execution Time is more difficult to predict: it is expected that the decompression overhead would not have more impact on the WCET than on the observed execution time. But the changes in the placement of code in memory engendered by code compression are likely to impact the results of the cache analysis. This is confirmed by our experiments that show that the profile of *l*-blocks is modified (the proportion of partial *l*-blocks, shared by several basic blocks, is greater in the compressed code) and that the distribution of *l*-block categories is changed. The result is, in most of the cases, an improvement of the WCET that is more significant than that of the average-case execution time. In other words, the impact of the code compression on the statistics of the *l*-blocks translates into a more accurate cache analysis and then more accurate WCET estimates.

These results show that code compression can be used in real-time critical systems without negative impact on Worst-Case Execution Times.

As future work, we plan to study how WCET-related information could be used in addition to static and dynamic information within the compression process to increase the accuracy of the cache analysis. This might help to improve further the WCET estimates.

Acknlowledgements

The authors would like to thank the ANR French Research Agency for financial support of the MORE project.

References

[1] M. Alt, C. Ferdinand, F. Martin, R. Wilhelm, "Cache behavior prediction by abstract interpretation", *Static Analysis Symposium (SAS)*, 1996.

[2] P. Altenbernd, "On the false path problem in hard real-time programs", *8th Euromicro Workshop on Real-Time Systems,* 1996

[3] C. Ballabriga, H. Cassé, "Improving the First-Miss Computation in Set-Associative Instruction Caches", *Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.

[4] L. Benini, F. Menichelli, and M. Olivieri. "A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems". *IEEE Transaction on Computers, 54(4)*, 2004.

[5] A. Beszedes, R. Ferenc, T. Gyimothy, A. Dolen and K. Karsisto, "Survey of code-size reduction methods", *ACM Computing Survey, 35(3)*, 2003.

[6] H. Cassé, P. Sainrat, "OTAWA, a framework for experimenting WCET computations", *3rd European Congress on Embedded Real-Time Software*, 2006.

[7] M. L. Corliss, E. C. Lewis, A. Roth, "The implementation and evaluation of dynamic code decompression using DISE", *ACM Trans. Embedded Comput. Syst. 4(1)*, 2005.

[8] P. Cousot, R. Cousot, "Static determination of dynamic properties of programs", *2nd International Symposium on Programming*, 1976.

[9] M. De Michiel, A. Bonenfant, H. Cassé, P. Sainrat, "Static loop bound analysis of C programs based on flow analysis and abstract interpretation", *IEEE Int'l Conf. on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, 2008.

[10] C. Ferdinand, F. Martin, R. Wilhelm, "Applying Compiler Techniques to Cache Behavior Prediction", *ACM SIGPLAN Workshop on Languages, Compilers and Tool Support for Real-Time Systems,* 1997

[11] L. Goudge, S. Segars, "Thumb: Reducing the cost of 32-bit RISC performance in portable and consumer application", *Proceedings of COMPCON6*, 1996.

[12] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: a free, commercially representative embedded benchmark suite", IEEE Workshop on Workload Characterisation, 2001.

[13] D. Hardy, I. Puaut, "WCET analysis of multi-level non-inclusive set-associative instruction caches", *29th IEEE Real-Time Systems Symposium*, 2008

[14] J. Henkel, H. Lekatsas, V. Jakkula,. "Design of an one-cycle decompression hardware for performance increase in embedded systems", *ACM Design Automation Conference (DAC)*, 2002.

[15] N. Holsti, "Analysing Switch-Case Tables by Partial Evaluation", *7th Workshop on WCET Analysis*, 2007.

[16] T. M. Kemp, R. K. Montoye, J. D. Harper, J. D. Palmer, D. J. Auerbach, "A decompression core for PowerPC", *IBM J. Res. Dev., 42(6)*, November 1998

[17] S. Lee, J. Lee, C.Y. Park, S. L. Min, "A Flexible Tradeoff between Code Size and WCET using a Dual Instruction Set Processor", *Workshop on Software and Compilers for Embedded Systems*, *LNCS 3199*, 2004.

[18] C. Lefurgy. *Efficient Execution of Compressed Programs*. PhD thesis, University of Michigan, 2000.

[19] Y.-T. S. Li, S. Malik, "Performance Analysis of Embedded Software using Implicit Path Enumeration", *Workshop on Languages, Compilers, and Tools for Real-time Systems*, 1995.

[20] X. Li, A. Roychoudhury, T. Mitra, "Modeling out-of-order processors for WCET analysis", Real-Time Systems, 34(3), 2006

[21] E. W. Netto, R. Azevedo, P. Centoducatte, G. Araujo, "Multi-profile based code compression", *ACM Design Automation Conference (DAC)*, 2004.

[22] C. Rochange, P. Sainrat, "A Context-Parameterized Model for Static Analysis of Execution Times", *Trans. on High-Performance Embedded Architectures and Compilers*, 2(3), Springer, 2007.

[23] S. Thesing, *Safe and Precise WCET Determination by Abstract Interpretation of Pipeline Models*, PhD thesis, Universität des Saarlandes, 2004.

[24] M. Thuresson, M. Själander, P. Stenström, "A Flexible Code Compression Scheme Using Partitioned Look-Up Table", *HiPEAC Conference*, 2009.

[25] K. Watanabe, Y. Umezawa, "Optimal Triggering Of An Airbag", *Intelligent Vehicles '93 Symposium*, 1993.

[26] WCET project / Benchmarks. Mälardaalen University. `http://www.mrtc.mdh.se/projects/wcet/benchmarks.html`