Progress in Flight Dynamics, GNC, and Avionics **6** (2013) 469-486
DOI: 10.1051/eucass/201306469

PROGRAM COMPACTION FOR REAL-TIME APPLICATIONS

A. Shalimov and R. Smeliansky

M. V. Lomonosov Moscow State University
Moscow, Russia

This paper presents a method of program compaction based on the frequency characteristics of program behavior. The proposed method keeps in the compiled form only frequently executed portions of programs and stores infrequently executed portions of programs in a compacted interpreted form, and dynamically unpacks and loads them into the memory for execution only when they are requested. Determination of infrequently executed code is done by estimating the execution frequency of the program's basic blocks based on the known distribution functions of its input parameters. It allows to control the growth of the compacted program execution time. The theoretical and experimental results of the research prove the possibility of using the proposed program compaction method in real-time systems.

1 INTRODUCTION

At present, all devices varied from sensors and mobile phones to aircrafts and satellites are equipped with an embedded control system. Today, embedded control system is the largest type of computer systems. For example, the number of microprocessors used in embedded systems exceeds by several orders of magnitude the number of microprocessors used in personal computers, servers, and computer system complexes.

Memory and energy are the most critical resources in embedded systems [1,2]. Applications for these systems often require more memory than can be fitted into the embedded systems, or more energy than can be provided by the embedded systems. The software has to adapt itself to a system. Moreover, computer history shows that memory is always a critical resource, and there is never enough because of the constantly growing demands on the functionality of programs.

Adding more memory to embedded system is a hard task. For example, in aircraft systems, memory chip must be resistant to electrical interference,

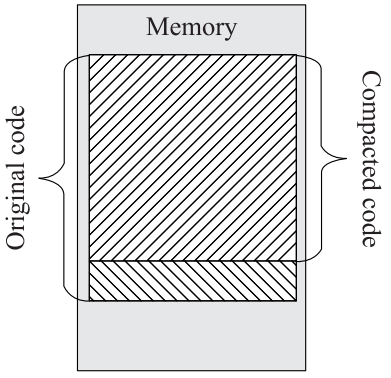

Figure 1 The scheme of reducing a program footprint

acceleration overload, and overheating, which lead to increasing its mass and weight. Even 100 KB of main memory in the aircraft has a size equaled to the PC's system block. For mobile phones, to add more memory is problematic because of limited energy and size: the user wants their mobile phone to be as small and light as possible.

In embedded systems, most of memory is used for program code, not for storing calculation results. Therefore, the actual trend is to use program compaction methods — a transformation that reduces the program code size in memory (memory footprint) while retaining the program functionality (Fig. 1). Moreover, the modern tendency of preferentially using of low-level programming languages for developing embedded systems (due to the fact that using high-level programming languages leads to large programs) may serve as an additional reason for requiring program compaction methods in embedded systems.

Almost all the embedded systems are real-time systems. Therefore, their software consists of real-time programs, whose execution time should not exceed the specified time limit (deadline). Therefore, program compaction methods have to take into account the time constraints set for the original programs.

In this paper, *soft real-time systems* are considered where programs must meet their deadlines in the average case (e. g., unlike hard real-time systems, where programs must meet their deadlines in the worst case).

Definition 1. *Program compaction for real-time systems is a program transformation C, which given the original program Π and the execution time threshold τ produces the compacted program $\Pi' = C(\Pi, \tau)$ such that*:

- *the functionality of Π' and Π coincide*;
- *the footprint of compacted program $M(\Pi')$ is smaller than the footprint of original program $M(\Pi)$: $M(\Pi') < M(\Pi)$*;
- *the execution time of the compacted program increases on average by no more than the given coefficient τ: $T(\Pi'(\bar{X})) \leq (1 + \tau)T(\Pi(\bar{X}))$.*

To the present authors' knowledge, there are no program compaction methods that focus on timing constraints on compacted program. All program compaction methods can be divided into two main groups: without decompression process (executable compressed form) [3–5] and with decompression process (decompress

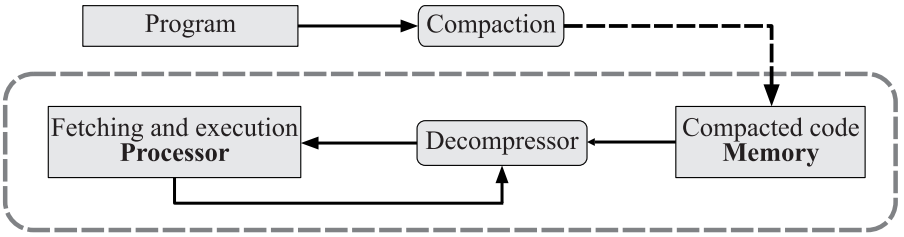

Figure 2 The main scheme of using program compaction methods

first then execute) [6–8]. Base scheme is shown in Fig. 2. Program compaction methods without decompression can be used in real-time systems, because they generally do not increase program execution time. However, their compression ratios strongly depend on the program details (for example, how often a program uses the different libraries, how much duplicated code is in the program, etc.). The average compression ratio of such methods is 0.9. Program compaction methods with decompression have a higher compression ratio. For instance, dictionary-based methods statically identify identical instruction sequences in the code and replace them by a codeword if they yield a smaller code size based on a heuristic. At run-time, the codeword is replaced by a dictionary entry storing the corresponding instruction sequence [8, 9]. The average compression ratio is 0.7. The main problem here is that these methods significantly increase the execution time. Thus, the existing methods generally cannot be used in real-time systems. So, it is important to create a program compaction method with decompression that will allow to control the program execution time vs. the compression ratio.

The method presented in this paper guarantees by construction that the compacted program execution time will not exceed, on average, a given time limit (see sections 2 and 3). The paper also contains mathematical models for determining the possibility of using the proposed method for a given embedded system (section 4). The experimental results of the research prove the possibility of using the proposed method of program compaction in real-time systems (section 5).

2 METHOD DESCRIPTION

The proposed method is based on the following two facts:

(1) for a sequential program, execution of 15%–20% of a program's code usually takes 80% of total program execution time [10–12]; and

(2) programs in interpreted form are usually smaller than in compiled form [13].

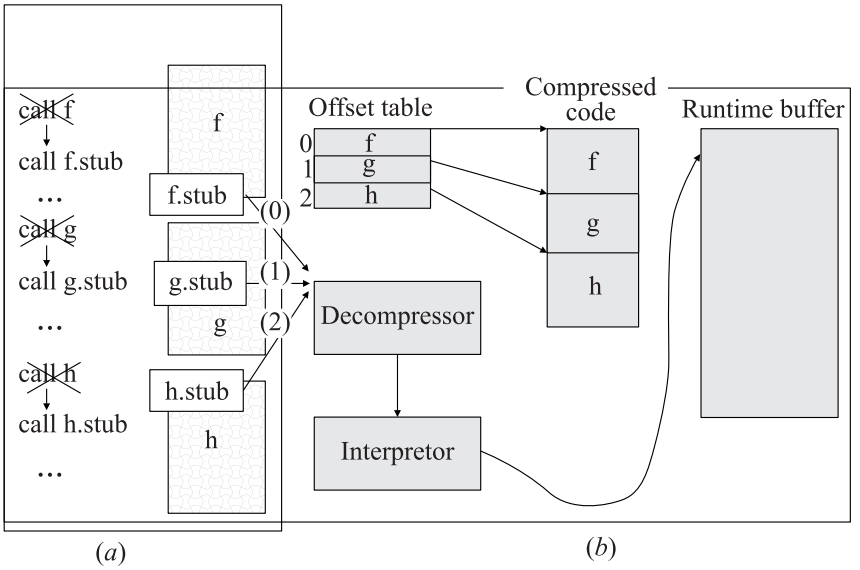

Figure 3 Diagram of the proposed program compaction method

These two facts serve as a starting point for a method that would allow to keep in compiled form only frequently executed portions of programs and to store infrequently executed portions of programs in compacted interpreted form and to dynamically unpack and load them into the memory for execution only when they are requested. This basic compression/decompression scheme was first described in [7].

Figure 3 shows the basic principles of the method. Let consider a program with three infrequently executed code fragments, f, g, and h, as shown in Fig. 3*a*. The structure of the code after compaction is shown in Fig. 3*b*. The code for each of these fragments is replaced by a stub (a very short sequence of instructions) that invokes a decompressor whose task is to decompress the interpreted code for a fragment into the runtime buffer and then transfer control to the interpreter for that decompressed code execution. A fragments offset table specifies the location within the compressed code where the code for a given fragment starts. The stub for each compacted fragment passes an argument to the decompressor that is an index into the offset table; this argument is indicated in Fig. 3*b* by the label $((0), (1), \dots)$ on the edge from each stub to the decompressor. The decompressor uses this argument as an index into the fragment offset table, retrieves the start address of the compacted code for the appropriate fragment, and starts generating uncompacted interpreted code into the runtime buffer. The decompressor then transfers control to the interpreter for the generated

interpreted code execution. When this decompressed code finishes its execution, it returns control to its caller in the usual way.

The proposed idea of program compression provides some possibilities and advantages over existing program compaction methods:

(1) using the suggested compressor/decompressor scheme allows to exploit the 80/20 aspect of programs [10–12]. For a sequential program, the strong majority of its execution time is usually spent on a small portion of the code. Therefore, compaction of the infrequently executed code will not lead to significantly increasing program execution time;

(2) this organization allows to adjust the compression ratio based on execution time deadlines and the amount of available memory;

(3) this organization also allows to use programs, which footprints are greater than the size of the available memory, by keeping infrequently executed code fragments out in the auxiliary memory (e. g., XIP*); and

(4) this method does not require any additional hardware, which would lead to increasing a real-time system's cost.

2.1 Compaction

In the proposed method, the compaction is done by translating infrequently executed code to a compressed interpreted form. First, let identify infrequently executed basic blocks and group them in order to receive more compaction gains and to decrease performance penalties. Then, each group is replaced by a stub, compacted, and stored. Finally, the offset table is updated.

Let look deeper into the steps. Assume that a set of infrequently executed basic blocks COLD $= \{b_{j_1}, \ldots, b_{j_l}\}$ has been already identified. How to actually determine b_j will be discussed in section 3.

It is necessary to combine the infrequently executed basic blocks in to groups $R_1, R_2, \ldots$ that are good for compaction. It is necessary to take into account the following requirements for groups:

- the interpreted form of a group must fit into the execution buffer. Note that the buffer contains only one group at a time;
- the benefit of compaction should be more than the overhead of adding stubs to the code and entries to the offset table; and

*__Execute in place (XIP)__ is a method of executing programs directly from long-term storage rather than copying it into RAM (random access memory). It is an extension of using shared memory to reduce the total amount of memory required.

– the number of control transfers between the groups must be small, since they lead to additional calls to the decompressor.

This task is proved to be NP-hard (nondeterministic polynomial-time hard) [7]. Therefore, the following heuristic algorithm is used:

(1) fix K, the upper limit of the size of the execution buffer;

(2) starting from each basic block, use a depth-first search on the control flow graph in order to construct a subgraph consisting of the infrequently executed basic blocks with a total size not exceeding K after translating to an interpreted form;

(3) if the benefit has been got after compaction of the resulting set of infrequently executed basic blocks, then this subgraph will form the region R_i. All used basic blocks are excluded from further consideration. Otherwise, the current basic block is marked so that region formation will not start from that block again; and

(4) the process continues until all unmarked infrequently used basic blocks are examined.

The constructed regions are replaced by stubs. Each stub has two arguments: a region number and a basic block number. The stub returns a control value that indicates which outgoing control flow edge should be taken. The control flow graph is modified in the following way: a basic block with a stub is added as a new vertex; all edges incoming to the current region now go to the new vertex; and all outgoing edges now go from the new vertex. Examples of transformation are shown in Fig. 4.

In practice, the regions are often smaller than the size of the execution buffer K. Therefore, to increase the efficiency of compaction, several small regions are combined into one big region that still fits into the execution buffer.

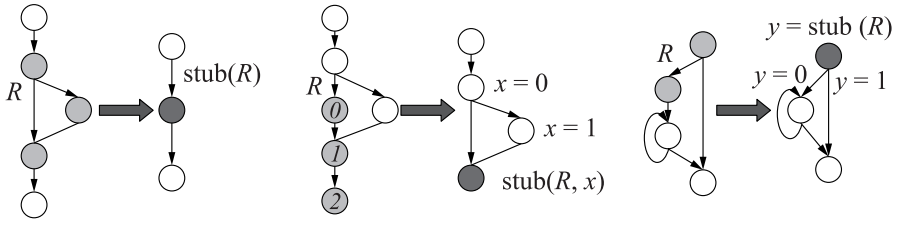

Figure 4 Examples of using stubs

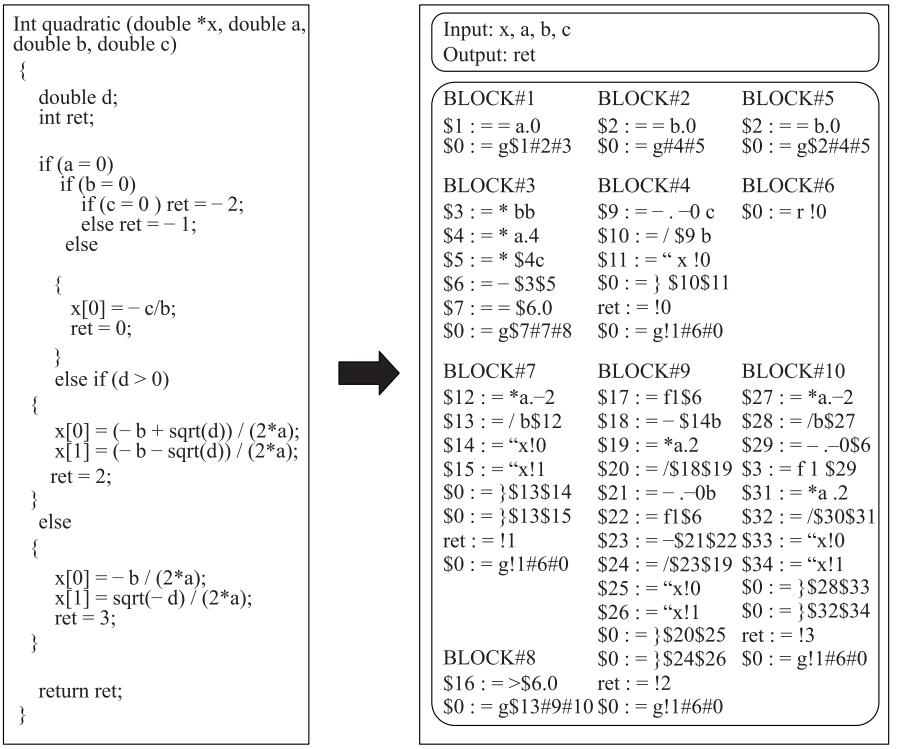

Figure 5 An example of translating a program to interpreted code

The regions use stubs with the same region number, but with different basic block numbers. Thus, the memory is saved by reducing the number of entries of the offset table. This process is called *packing*.

The next task is to generate the interpreted form of the infrequently executed code. The basis of the interpreted code is prefix notation (also known as polish notation), where operators are placed to the left of their operands. For example, $5 + 6$ can be written in prefix form as $+ \ 5 \ 6$. An example of the translation to interpreted code is shown in Fig. 5. The representation used consists of a finite number of unique characters — about 30. The number of unique characters does not exceed 64. So, no more than 5 bits per character ($2^5 = 32$) is used. Compared to full ASCII, this interpreted form provides a compression ratio of 40%.

To compress the interpreted code, the Huffman encoding was used [14]. The algorithm represents the most frequently used symbols of the input stream by bit sets of a shorter length, and infrequently used symbols are represented by longer sets of bits. Important features of this coding method are rapid decompression

with a small memory footprint. Also, it is easy to predict the time needed for code decompression. In practice, the compressed text is approximately 66% of its original size.

Note that any other text compression scheme could be used. Remember that compression ratio is not the primary goal. The aim is to provide the ability to control the execution time of compacted program.

2.2 Execution

The main subtasks for executing the compacted program are:

(1) decompression of the infrequently executed code;

(2) interpreted code execution; and

(3) control of the overall process of program execution.

The basic ideas of compacted program execution have been described above. Now let consider some details. If during code interpretation a function call is met, the interpreter's context is stored: the region number, the basic block, the instruction number, and the interpreter's state (the values of all variables of the interpreted code). If the function call leads to control transfer to another infrequently executed code, the decompressor puts that new code into the buffer. After return from the called function, the decompressor unpacks the region and transfers the control to the interpreter. The interpreter loads the saved context and continues to execute code from the desired position.

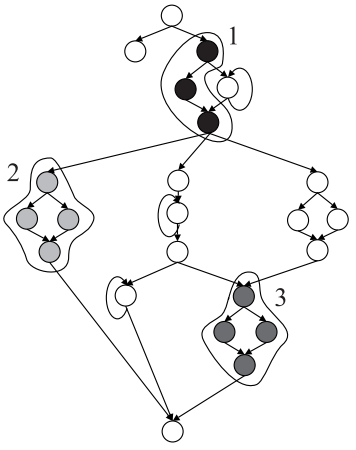

Figure 6 The example of program packing

Note that the execution buffer is not emptied when the control returns to the program. The current region is still in the buffer until the execution of a new region. In some situations, it reduces the overhead of reunpacking the compressed interpreted code. For example, execution of a called function may not lead to control transfer to the infrequently executed code. Then, after returning to the interpreted code, the buffer already contains the desired region and reunpacking is not required. The next example relates to packing (see the previous subsection). If the next code is also in the current region, decompressing is not required as well. Moreover, in order to increase the

probability of such coincidences, during the packing process, the code from the same path of a program execution is combined. For example, in Fig. 6, it is more profitable to pack the first and the third regions together, because there are two paths between them, whereas it is only one path between the first and the second regions. The most probable edges are also considered and the code from the appropriate branches of the program is combined.

3 METHOD DETAILS

A major challenge of the proposed approach is how to choose infrequently executed basic blocks. In this paper, a code is classified as infrequently executed if its execution time is less than the execution time of other code. Determining what basic blocks are infrequently executed is based on execution frequencies.

3.1 Determining Execution Frequency of a Program's Basic Blocks

To solve this task, the authors have developed a method of determining the execution frequency of a program's basic blocks [15].

There are given $\Pi(x_1, \ldots, x_p) = \{V, E\}$ — the original sequential program with p input parameters $(x_1, \ldots, x_p)$. The program is represented as a control flow graph where the vertices $V = \{b_j\}$ $(j = 1, \ldots, m)$ are the basic blocks and edges $E = \{(b_{j_1}, b_{j_2})\}$ are the possible transfers of control flow from one basic block to another.

For each input parameter $x_1, \ldots, x_p$, the finite set of admissible values and the distribution function for these values are known. Note that it does not matter if the parameters are countable-valued or real-valued because only finite sets of those values are picked. For example, the cruising speed of a Boeing 737 is 780 km/h, but it can vary by 20 km/h. Thus, the values of the variable V are [760, 800], and are normally distributed with parameters 780 and 20. However, an open question is how to determine distribution function of a variable. One way is recording user input and statistically constructing a distribution function based on these values.

Let assume the program $\Pi(x_1, \ldots, x_p)$ does not get caught in an endless loop on admissible inputs (i. e., each basic block is executed a finite number of times).

Let use the following notations:

- $T(x_i)$ is the finite set of admissible values of input parameter x_i;
- $M_p = T(x_1) \cdots T(x_p)$ is the set of all inputs of size $|M_p|$ and dimension p;
- $\hat{x}_i$ is the actual value of input parameter x_i; and

– $\Pi_j(\hat{x}_1, \ldots, \hat{x}_p)$ is the number of b_j basic block executions of the program running on inputs $(\hat{x}_1, \ldots, \hat{x}_p)$ (i. e., the value for the counter of a given basic block).

Each input parameter can be treated as a random variable with a given distribution function $X_1, \ldots, X_p$. Therefore, the counter for the basic block $\Pi_j(X_1, \ldots, X_p)$ is a random value too.

Definition 2. *The execution frequency of basic block b_j is the value of the mathematical expectation of the basic block counter Π_j:*

$$e(b_j) = \sum_{(\hat{x}_1,\ldots,\hat{x}_p)\in M_p} \Pi_j(\hat{x}_1, \ldots, \hat{x}_p) P((X_1, \ldots, X_p) = (\hat{x}_1, \ldots, \hat{x}_p)) .$$

Calculation of $e(b_j)$ requires an enormous computational outlay, comparable with running a program on all input values. Let calculate the frequency $e(b_j)$ with a given precision ε and reliability γ (i. e., find an estimated value N_j such that $P(|e(b_j) - N_j| \leq \varepsilon) \geq \gamma$).

The idea of the proposed approach is to use the Monte Carlo method [16]. In the beginning of each basic block, let add a special counter which is incremented each time control flow goes into that basic block. The modified program is rerun iteratively. On each iteration, new values for the input parameters are generated using their distribution functions. After n program runs, there will be n values of the execution counter for each basic block $\Pi_j^1, \Pi_j^2, \ldots, \Pi_j^n$. The Law of Large Numbers, the Central Limit Theorem, and the Berry–Essen theorem are applicable to the analysis of these numbers as a sample of the random variable Π_j.

According to the Law of Large Numbers, $N_j = (1/n) \sum_{i=1}^{n} \Pi_j^i$ is the average of the values of basic block execution counter obtained from a large number of program runs should be close to the mathematical expectation $e(b_j)$ and will tend to become closer as more program runs are performed ($N_j \to e(b_j)$ when $n \to \infty$). Both the Central Limit Theorem and the Berry–Essen theorem allow to estimate the number of program runs necessary to get the execution frequency with a given precision and reliability.

The following algorithm evaluates basic block execution frequency.

1. Set ε, γ.

2. Initialize the counter of program runs, $n = 0$.

3. Run the modified program on a generated set of input data; Π_j^n is the value of b_j execution counter. Increment the number of program runs, $n = n+1$.

4. If $n > 30$, then calculate the following values (let assume that after 30 iterations, the sample characteristics can be trusted); else, go to step 3:

(a) $N_j = (1/n) \sum_{i=1}^{n} \Pi_j^i$ is the average;

(b) $s_j^2 = (1/(n-1)) \sum_{i=1}^{n} (\Pi_j^i - N_j)^2$ is the sample variance; and

(c) $m_j^3 = (1/(n-1)) \sum_{i=1}^{n} (\Pi_j^i - N_j)^3$ is the sample third central moment.

5. If $0.5 m_j^3/(s_j^3\sqrt{n}) \le (1-\gamma)/10$ and $n > \left(u_{(1+\gamma)/2}/\varepsilon\right)^2 s_j^2$, then N_j evaluates $e(b_j)$ with the given precision ε and reliability γ ($u_{(1+\gamma)/2}$ is the quantile of order $(1+\gamma)/2$ of the standard normal law); else, go to step 3.

Note that this algorithm is not applicable to basic blocks with constant execution frequency (i. e., if a basic block execution counter does not depend on input data, or the probability of basic block execution is close to zero). Therefore, if during the program runs the execution counter of a basic block remains the same, then the final decision about the execution frequency of such basic block should be taken by a programmer, i. e., a programmer should decide to continue programs reruns or to stop.

3.2 Determining Infrequently Executed Code

The suggested approach to determine infrequently executed code is to define a threshold θ as the portion of a program's execution time that infrequently code can account for (i. e., if execution time of any program's code is less than θ, then this code is called infrequently executed). How to choose threshold is described in section 4.

Let use the following notations:

- $t(b_j)$ is the execution time of the basic block b_j during a single run on a given processor. The assessment uses the CPI (cycles per instruction), the number of clock cycles that happen when an instruction is being executed [17];
- $t(b_j)e(b_j)$ is the average execution time of the basic block b_j during program execution. It is called the basic block's "weight;" and
- $T = \sum_{j=1}^{m} t(b_j)e(b_j)$ is the average execution time of the whole program.

Definition 3. *Infrequently executed code is a set of a program's basic blocks with cumulative execution time not exceeding θ of the average execution time of the program*:

$$\mathrm{COLD} = \left\{ b_{j_1}, \cdots, b_{j_l} \,\middle|\, \sum_{k=1}^{l} t(b_{j_k})e(b_{j_k}) \le \theta T \right\}.$$

Let now consider the task of compaction of infrequently executed code as an optimization problem.

Let $C(\Pi, \theta)$ be a program compaction method based on infrequently executed code compaction. For each basic block b_j, its original size $s(b_j)$, compaction size $s'(b_j)$, execution time $t(b_j)$, and execution frequency $e(b_j)$ are known. Then, it is needed to find a vector $Z = (z_1, \cdots, z_m)$, $z_j \in \{0, 1\}$, where if $z_j = 0$, then b_j will not be compacted, and if $z_j = 1$, then b_j will be compacted:

$$\sum_{j=1}^{m} (z_j s'(b_j) + (1 - z_j) s(b_j)) \to \min;$$
$$\sum_{j=1}^{m} z_j t(b_j) e(b_j) \leq \theta T.$$

The following suboptimal algorithm is used to choose the infrequently executed basic blocks.

1. All basic blocks are considered in descending order of the compaction gains divided by the weights, $(s(b_j) - s'(b_j))/(t(b_j)e(b_j))$.

2. If the sum of the weights of the basic blocks $t(b_j)e(b_j)$ exceeds time θT, then go to the next basic block.

3. All chosen basic blocks are infrequently executed (see Definition 3).

As mentioned above, a program compaction method is presented for soft real-time systems. In such systems, missing a deadline does not cause catastrophic consequences on the environment, but only a performance degradation, often evaluated through some quality of service parameter. There are two quality factors of soft real-time systems: the number of missed deadlines and the tardiness (positive difference between the completion time and the deadline). So, depending on the factor which should be minimized, an additional criteria is added to task 1:

- $\sum_{j=1}^{m} z_j e(b_j) \to \min$, when minimizing the expected number of missed deadline; and

- $\sum_{j=1}^{m} z_j (\max(b_j) - e(b_j)) \to \min$, when minimizing the total tardiness (value $\max(b_j)$ is the maximum recorded value of the execution counter during iterative program reruns in the previous step, see subsection 3.1).

To solve the obtained multicriteria problems, an approximation of the Pareto front was constructed using an algorithm based on the search for directions [18].

4 METHOD APPLICATION

This section shows that the proposed method of program compaction can be used in real-time systems. Mathematical dependencies are suggested for determining the possibility of using the proposed method in the given real-time system. The following notations are used:

(1) θ is the threshold of infrequently executed code;

(2) τ is the coefficient of admissible increase of a program's execution time;

(3) $\lambda_C(\theta)$ is the compression ratio of the proposed method,

$$s'(b_j) = \lambda_C(\theta) s(b_j)\,.$$

This depends on θ: the larger the number of program's instructions are labelled as infrequently executed, the greater the compression ration will be. The compression ratio is calculated empirically for a given implementation of the method (see section 5);

(4) α_C is the coefficient of increasing execution time of a compacted code relative to the execution time of an original code, $t'(b_j) = \alpha_C t(b_j)$; and

(5) $M(C)$ is the size of memory used by the complete, compacted implementation.

The proposed method stores infrequently executed code in compacted interpreted form and dynamically executes it. According to the infrequently executed code definition (see Definition 3), the execution time of a compacted program is equal to $\alpha_C \theta T + (1-\theta)T$. This time should be more than $(1+\tau)T$ (see Definition 1) and $(1+\theta(\alpha_C - 1))T \le (1+\tau)T$. Consequently, $\theta \le \tau/(\alpha_C - 1)$. Therefore, in the average case, in order not to exceed a given time limit for the compacted program execution time, the threshold θ should be not more than $\tau/(\alpha_C - 1)$. Note the dependency of τ on the parameter θ is $\tau(\theta) = (\alpha_C - 1)\theta$.

The memory overhead resultant from a program compaction method use must not exceed the amount of memory saved due to a program compaction. The total compaction gain is

$$\sum_{i=1}^{k} (M(\Pi_i) - \lambda_C(\theta) M(\Pi_i)) = (1 - \lambda_C(\theta)) \sum_{i=1}^{k} M(\Pi_i)\,.$$

It should be more than $M(C)$. Consequently,

$$\sum_{i=1}^{k} M(\Pi_i) > \frac{M(C)}{1 - \lambda_C(\theta)}\,.$$

So, it is necessary to select the programs with total footprint exceeding $M(C)/(1-\lambda_C(\theta))$.

Thus, for using the proposed method, it is necessary:

(1) to choose the parameter $\theta \leq \tau/(\alpha_C - 1)$;

(2) to choose the programs for compaction whose total footprint is more than $M(C)/(1-\lambda_C(\theta))$; and

(3) if the above conditions (1) and (2) cannot be concurrently met, or a greater compression ratio is required, then it is necessary to increase the system performance by $(1+\theta(\alpha_C-1))/(1+\tau)$.

If the above recommendations on the use of the proposed program compaction method are satisfied, it is guaranteed [19] that the program execution time will increase on average not more than by factor τ, with the compression ratio $\lambda_C(\theta)$. Thus, the method can be successfully used in soft real-time systems.

5 METHOD IMPLEMENTATION

A program compactor implementing the proposed method was written in the C++ language. It consists of two parts (Fig. 7): program compaction (compressor) and execution of compacted program (decompressor). The input for the system is a program for compression (written on the C language), distribution functions of the program's input parameters, and the threshold of infrequently executed code. The output is a compacted program, an offset table, and the compacted interpreted code.

The compressor transforms infrequently executed program code into the interpreted form and compresses it as text using Huffman coding [14]. Based on

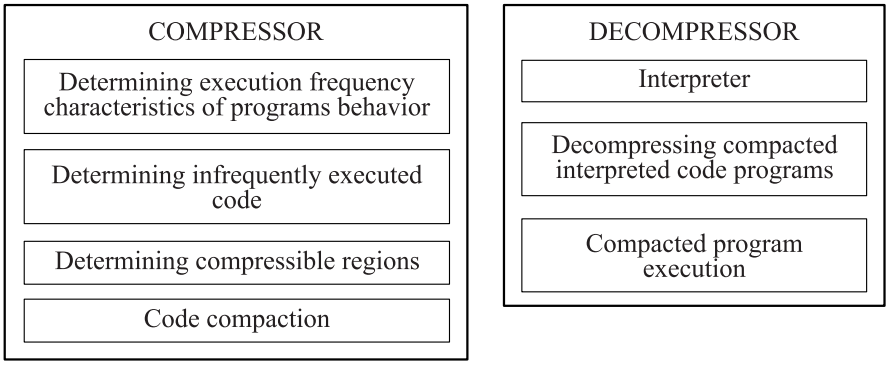

Figure 7 The structure of the proposed method implementation

the distribution functions of input parameters, the compressor determines the execution frequency of program basic blocks. The threshold and calculated execution frequencies are used to determine infrequently executed basic blocks. The infrequently executed code is grouped to get more size gains. The compressor is implemented in the LLVM (low-level virtual machine) compiler as an optimization pass.

The decompressor consists of two main parts: the interpreter and a service for decompression of compacted interpreted code. The decompressor functionality is in a shared library that is loaded dynamically at run-time on request.

The proposed method has been tested on real-time programs used in modern real-time systems. The aim of the testing was to determine the following characteristics of the proposed method:

- $\lambda(\theta)$ is the compression ratio depending on the input parameter θ (the threshold of infrequently executed code); and
- $\tau(\theta)$ is the coefficient of increasing program execution time depending on the input parameter θ (the threshold of infrequently executed code).

Note that the resulting coefficient of increasing program execution time must met the dependency $\tau(\theta)$ formulated in analytic form in the previous section: $\tau(\theta) = (\alpha_C - 1)\theta$. It will be shown that in practice, the proposed program compaction method can correctly meet the given time constraints imposed on the real-time programs.

Test programs have been used from the following sources: DrTesy* (Moscow State University, an international project on the modeling of an aircraft navigation system), SNU Real-time† (Seoul National University, a set of programs for numerical computation for DSP (digital signal processing) real-time systems), MiBench‡ (University of Michigan, a set of programs used in embedded real-time systems).

For each test program, the system was run 31 times with different values of θ: 0.01, 0.02, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1, 0.12, 0.14, 0.16, 0.18, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6, 0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, and 1. The steps of θ are not uniform: the majority of the values are near zero, since this is the most useful portion of the design space.

After each run, the compacted program size and compacted program average execution time were saved. These values were averaged across all benchmarks at a given θ.

Figure 8*a* shows the resulting relationship between compression ratio and the input parameter, $\lambda(\theta)$. The best compression ration was $\lambda(\theta) = 0.77$ for $\theta = 1$ (all executed instructions are compacted). At $\theta = 0.5$, the compression

*http://lvk.cs.msu.su/index.php/articles/65.
†http://www.cprover.org/goto-cc/examples/snu.html.
‡http://www.eecs.umich.edu/mibench/.

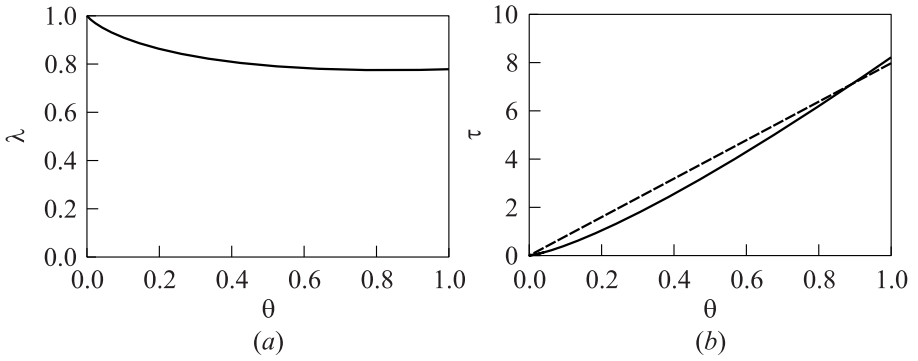

Figure 8 The dependencies between the compression ratio λ (a) and the coefficient of increasing program execution time τ (b) and the input parameter θ — execution time threshold

ratio was $\lambda(\theta) = 0.79$. Note that for small values of the threshold of infrequently executed code $0 < \theta \leq 0.25$, the highest decrease of program compacted size has been got, with compression ratio $\lambda(\theta) = 0.85$. For $0.3 \leq \theta \leq 0.45$, the grade of the compression ratio decreases: compression ratio changes from 0.83 to 0.8. Therefore, it is necessary to choose the input parameter from the range $\theta \leq 0.5$, since for the larger values of θ, the improvement in compression ratio is negligible.

Figure 8b shows the relationship between the coefficient of increasing program execution time and the input parameter, $\tau(\theta)$. The experimentally obtained values of $\tau(\theta)$ do not exceed the analytically obtained expression $\tau(\theta) = (\alpha_C - 1)\theta$ (for $\alpha_C = 9$) formulated in the previous section (dashed line). One exception is for $\theta = 1$. This happened because of higher amount of "load to buffer" operations since whole program was translated to the interpreted form. To recap, when the input parameter was chosen according to the recommendations described in the previous section, the execution time of compacted program will satisfy the given time constraints.

After analysis of the decompressor code, the following characteristics of the implementation of the proposed program compaction method have been obtained:

(1) the number of instructions used for execution of a single interpreted command is equal to nine ($\alpha_C = 9$); and

(2) the amount of additional memory for using the proposed program compaction method is equal to 103 KB ($M(C) = 103$ KB).

So, the recommendations for using proposed method in real-time systems can be now finalized. For use of the proposed method implementation in real-time systems, it is necessary:

(1) to choose the parameter $\theta \leq (1/8)\tau$ where τ is the coefficient of admissible increase of compacted program execution time relative to the original program; and

(2) to choose the programs for compaction whose total footprint is more than $103/(1 - \lambda(\theta))$ KB where $\lambda(\theta)$ is the compression ratio of the implemented system of program compaction for a chosen θ (see Fig. 8*a*).

If the above conditions cannot be concurrently met, or a greater compression ratio should be gained, then it is needed to add the requirement to increase the system performance in $\rho = (1 + 8\theta)/(1 + \tau)$ times.

Let consider the example of choosing an input parameter θ according to above recommendations. Assume a program's footprint is 200 KB, and the admissible increase of execution time of the compacted program is 30% ($\tau = 0.3$). Then, the input parameter has to be set to $\theta = 0.04$. This allows to achieve the compression ratio of $\lambda(\theta) = 0.95$ (the program footprint will be reduced by 10 KB). If this compression ration is not enough, it will be necessary to increase the processor's performance. For example, if $\rho = 2$, the compression ration will be equal to $\lambda(\theta) = 0.85$ (the program footprint will be reduced by 30 KB).

6 CONCLUDING REMARKS

This article presents a program compaction method based on the frequency characteristics of program behavior. An implementation of the proposed method has been written in the C++ language. Testing of this implementation was aimed at determining the dependency of the compression ratio on the input parameter and the dependency of the coefficient of increasing execution time on the input parameter. Modern real-time programs were used for testing the method implementation. The experiments show that the proposed method has a high compression ratio (compared with existing program compaction methods) and it is able to control both the program execution time and the compression ratio. The testing results prove the possibility of using the proposed method in real-time systems. The mathematical dependencies guarantee that the compressed program execution time will not exceed on average a given time limit.

REFERENCES

1. Kolpakov, K. 1999. History of onboard embedded systems in Russia. *PCWeek* 32.
2. Pavlov, A. 2001. Principles of organization of advanced onboard computing systems. *World Computer Automation: Embedded Computer Syst.* 4.

3. Bus, B., D. Kastner, D. Chanet, L. Put, and B. Sutter. 2003. Post-pass compaction techniques. *Comm. ACM* 46(8).
4. Sutter, B., and K. Bosschere. 2003. Software techniques for program compaction. *Comm. ACM* 46(8).
5. Lee, S., and J. Lee. 2007. Selective code transformation for dual instruction set processors. *ACM Trans. Embedded Computing Syst. (TECS)* 6(1).
6. Kemp, T., and R. Montoye. 1998. A decompression core for PowerPC. *IBM J. Research Development* 42(5/6).
7. Debray, S., and W. Evans. 2002. Profile-guided code compression. *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI12) Proceedings*. Berlin, Germany: ACM. 95–105.
8. Seong, S., and P. Mishra. 2007. Bitmask-based code compression for embedded systems. *IEEE Trans. Computer-Aided Design Integrated Circuits Syst.* 27(4):673–85.
9. Thuresson, M., and P. Stenstrom. 2005. Evaluation of extended dictionary based static code compression schemes. *2nd Conference on Computing Frontiers Proceedings*. Ischia, Italy: ACM. 77–86.
10. Knuth, D. 1971. An empirically study of fortran programs. In: *Software: Practice and experience*. 1.
11. Smeliansky, R., D. Guryev, and A. Bahmurov. 1986. About one mathematical model for calculation of programs behavior. *Programming Computer Software* 6.
12. Smeliansky, R., and T. Alanko. 1986. On the calculation of control transition probabilities in a program. *Information Processing Lett.* 22.
13. Brown, P. 1979. Macros without tears. *Software: Practice and experience*. John Willey. 9:433–37.
14. Huffman, D. 1952. A method for the construction of minimum-redundancy codes. *Institute of Radio Engineers*. 40.
15. Shalimov, A. 2010. Method of determining execution frequency of programs basic blocks. *Modeling and Analysis of Information Systems* 17(2).
16. Robert, C., and G. Casella. 2010. Monte Carlo statistical methods. *Springer texts in statistics*.
17. Shaaban, M. 2009. Lectures on computer organization. Rochester, NY: RIT. http://meseec.ce.rit.edu/eecc550-winter2009/.
18. Deb, K. 2009. *Multi-objective optimization using evolutionary algorithms*. Wiley Paperback.
19. Shalimov, A. 2011. The research of program compaction methods for real-time applications. PhD Thesis. Lomonosov Moscow State University.