The method of programs compression based on the frequency characteristics of programs behaviour

Alexander Shalimov
Lomonosov Moscow State University, Russian Federation
Email: ashalimov@lvk.cs.msu.su

***Abstract*—This article presents a method of programs compression based on the frequency characteristics of programs behaviour. The proposed method allows us to keep in the compiled form only frequently executed portions of programs and to store infrequently executed portions of programs in the compacted interpreted form, and to dynamically unpack and load them into the memory for execution only when they are requested. The method also allows us 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 method for programs compression in embedded real-time controlling systems.**

***Index Terms*—code compression; code compaction; program compaction; decompression; program analysis; execution frequency; embedded systems.**

I. Introduction

Embedded real-time controlling systems is one of the main areas of the computer engineering and software development [1].

Let's list the characteristics of such systems:

1) The necessity to fulfill the requirements of high reliability and safety of an embedded system operation.
2) The programs must be executed in their deadlines, i.e. program's execution time must not exceed a given time limit.
3) Memory limitation. Such systems have small amount of main memory (for example, in modern aircraft available approximately 10MB of RAM [2], [3]).
4) Systems space and weight limitations.

The complexity of tasks for embedded real-time controlling systems is increasing, which results in consuming more memory resources. Therefore the most important characteristic of a program for such systems is the program's runtime memory size.

Programs compression methods reduce the program's runtime memory size (memory footprint). Hence it will enable to create major amount of services configured the system functionality. Besides, the modern tendency of preferential using of low-level programming languages for designing embedded systems (due to the fact that using of high-level programming languages leads to generation of exceeding program code) may serve as an additional argument for necessity of using programs compression methods in embedded systems.

In fact compressed programs run slower than original programs (overhead on execution speed). But, as mentioned above, for embedded real-time controlling systems it is important that the programs should be executed in its deadlines. This fact leads to the following main requirement for the compression methods: *the compressed program must not exceed the execution time limit set for the original program*. As a matter of fact there is always some gap between the time limit set for a program and its real time execution. This gap can be used as a time resource for compression.

Generally all program compression methods can be divided into two main groups: without decompression process [4], [5], [6] and with decompression process [7], [8], [9]. Programs compression methods without decompression can be used in the embedded real-time control systems, because they practically don't increase programs execution time. But its compression ratio strongly depends on the program (for example, how often program uses the different libraries, how many duplicated code are contained in the program, etc). Programs compression methods with decompression have a high compression ratio, but methods lead to considerable increasing execution time without possibility to control this process and the exists method can't be used in real-time controlling systems.

That is why it is necessary to create a program compression method with decompression which would allow us to control the program execution time depending on the compression ratio.

The method presented in this article guaranties that the compressed program execution time will not exceed on average a given time limit (Sections 2, 3). Also the article contains the mathematical dependencies 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 for programs compression in embedded real-time controlling systems (Section 5).

II. Method Description

The idea of the proposed method rises from [10], [11], [12] and is based on the following two facts:

1) For a consequent program, execution of 15-20% of the program usually takes 80% of total program execution time [10], [11].
2) Program presentation in the interpreted form is usually smaller than it is in the compiled form [12].

These two facts served as a starting point for making a decision to research a method, that would allow us to keep in the compiled form only frequently executed portions of programs and to store infrequently executed portions of programs in the compacted interpreted form, and to dynamically unpack and load them into the memory for execution only when they are requested. The basic compression/decompression scheme was first described in [9].

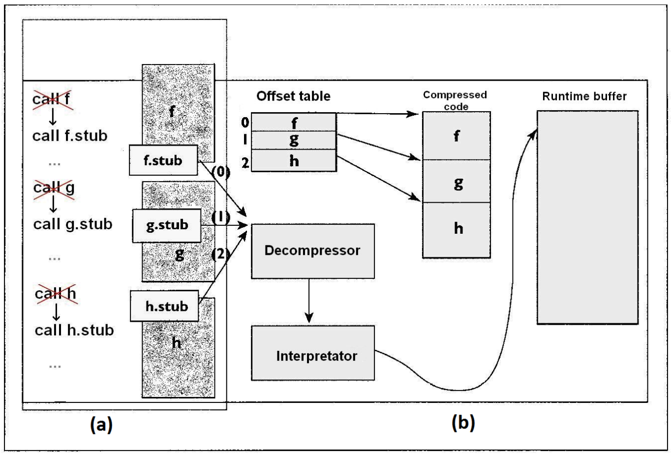

Fig. 1. Main scheme

The developed method of programs compression consists of two main parts: program compression and the compressed program execution. Figure 1 shows the basic principles of the method. Consider a program with three infrequently executed fragments of code, f, g and h, as shown in Figure 1(a). The structure of the code after compression is shown in Figure 1(b).The code for each of these fragments of code is replaced by a stub (a very short sequence of instructions) that invokes a decompressor whose job is to decompress the interpreted code for a fragment into the runtime buffer and then to transfer control to the interpreter for this 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 compressed fragment passes an argument to the decompressor that is an index into this table; this argument is indicated in Figure 1(b) by the label ((0), (1), ...) on the edge from each stub to the decompressor. The decompressor uses this argument to index into the fragment offset table, retrieves the start address of the compressed code for the appropriate fragment, and starts generating uncompressed 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 to its caller in the usual way.

The proposed idea of programs compression provides some possibilities and advantage over the existing programs compression methods.

1) Using "80-20 rule". Using compressor/decompressor scheme allows us to exploit 80/20 aspect of programs [10], [11]. For a consequent program the greater part of its execution time is usually spent on a smaller program part execution. Therefore the infrequently executed code compression will not lead to significant increasing program execution time.
2) This method's organization allows us to manage the program compression level depending on requirements to a program execution time and amount of available memory. This will allow us for embedded real-time controlling systems to consider requested time limits of programs execution time.
3) Using software implementation of the compressor/decompressor scheme enable to use programs, which can not be fully loaded into the main memory, due to keeping infrequently executed code fragments in the auxiliary memory.
4) This method does not require any hardware and hence great expenses and time losses related to adding hardware to an embedded system.

III. Method Details

The proposed method is intended to solve the following tasks:

1) Determining frequency characteristics of programs behavior;
2) Determining infrequently executed portions of programs.

A. Determining execution frequency of programs basic blocks

To solve this task the author has developed the method of determining execution frequency [13].

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 presented in the form of a control flow graph where the vertices $V = \{b_j\}$ $(j = \overline{1, m})$ represent basic blocks and edges $E = \{(b_{j_1}, b_{j_2})\}$ represent possible transfer of control flow from one basic block to another.

For each input parameter $x_1, \ldots, x_p$ we know a finite set of admissible values and the distribution function for these values. The program $\Pi(x_1, \ldots, x_p)$ does not get caught in an endless loop on admissible sets of input parameters (i.e. each basic block is executed a finite number of times).

Lets use the following notations:

1) $T(x_i)$ — the set of admissible values of input parameter x_i;
2) $\hat{x}_i$ — the value of the input parameter from $T(x_i)$;
3) $M_p = T(x_1) \times \ldots \times T(x_p)$ — the set of all inputs of power $|M_p|$ and of dimension p;
4) $\Pi_j(\hat{x}_1, \ldots, \hat{x}_p)$ — the number of basic block executions while program running on the $(\hat{x}_1, \ldots, \hat{x}_p)$.

Each input parameter can be treated as a random variable with a given distribution function. Lets assume $X_1, \ldots, X_p$ random values for input parameters $(x_1, \ldots, x_p)$.

Then, the frequency of b_j we will consider as $e(b_j) = \sum\limits_{(\hat{x}_1, \ldots, \hat{x}_p) \in M_p} \Pi_j(\hat{x}_1, \ldots, \hat{x}_p) \cdot P((X_1, \ldots, X_p) = (\hat{x}_1, \ldots, \hat{x}_p))$ - the value of mathematical expectation of b_j execution count.

Calculation of $e(b_j)$ requires an enormous computational outlay comparable with an outlay for running a program on all input values. In the paper [13] it was proposed to calculate the frequency $e(b_j)$ with a given precision ε and reliability γ, i.e. to find such estimated value N_j that $P(|e(b_j) - N_j| \leq \varepsilon) = \gamma$.

The idea of proposed approach is to use the Monte Carlo method. In the beginning of each basic block we add a special counter which is incremented each time when a control flow goes into that basic block. The modified program is being iteratively re-run. On each iteration new values for input parameters are generated using their distribution functions. After n program runs we will have n values of execution counter for each basic block $\Pi_j^1, \Pi_j^2, \ldots, \Pi_j^n$. It is proved that *Law of Large Numbers*, *Central Limit Theorem* and *the Berry-Essen theorem* are applicable for analysis of these numbers.

According to the Law of Large Numbers, $N_j = 1/n \cdot \sum_{i=1}^{n} \Pi_j^i$ — 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 Central Limit Theorem and the Berry-Essen theorem allow us to estimate a number of program runs to get the execution frequency with a given precision and reliability.

See the next algorithm for evaluation basic block execution frequency.

1) Set ε, γ.
2) Set counter of program runs to zero, $n = 0$.
3) Run modified program on a generated set of input data. Π_j^n – the value of b_j execution counter. Increase the number of program runs, $n = n + 1$.
4) If $n > 30$ then calculate the following values (we assume that after 30 iterations we can trust to sample characteristics). Else go to step 2.
   a) $N_j = 1/n \cdot \sum_{i=1}^{n} \Pi_j^i$ — the average,
   b) $s_j^2 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (\Pi_j^i - N_j)^2$ — the sample variance,
   c) $m_j^3 = \frac{1}{n-1} \cdot \sum_{i=1}^{n} (\Pi_j^i - N_j)^3$ — the sample third central moment.
5) If the $\frac{0.5 m_j^3}{s_j^3 \sqrt{n}} \leq \frac{1-\gamma}{10}$ and $n > \left(\frac{u_{\frac{1+\gamma}{2}}}{\varepsilon}\right)^2 \cdot s_j^2$ then N_j evaluates $e(b_j)$ with a given precision and reliability ($u_{\frac{1+\gamma}{2}}$ - quantile of order $\frac{1+\gamma}{2}$ of the standard normal law). Else go to step 2.

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 probability of its execution is closer to zero). Therefore if during the program runs the execution counter of some 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.

B. Determining infrequently executed code

Lets use the following notations:

1) threshold θ – the part (quota) of a total program 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 are described in Section 4). Note further we assume that execution times of instructions are the same and take one unit of time (program execution time is measured in the number of instruction executed at program runtime).
2) $weight(b_j) = e(b_j) \cdot |b_j|$ — the weight of a basic block be the number of instructions in the block multiplied by its execution frequency.
3) $T_{av} = \sum_{j=1}^{m} weight(b_j)$ — the average number of instructions executed at program runtime.

We consider all basic blocks in the program in increasing order of execution frequency until the sum of their weights will not exceed $\theta \cdot T_{av}$. All selected basic blocks are considered to be infrequently executed.

IV. Method Application

This section is about the mathematical dependencies for determining the possibility of using the proposed method for a given embedded system. Use the following notations:

1) $\tau \geq 1$ – the coefficient of admissible increasing of a program execution time. This coefficient is necessary for using a gap between actual and requested execution time.
2) $\lambda(\theta)$ — the compression ratio achieved by using the proposed method. It is calculated empirically for a given implementation of the method.
3) I – the number of instructions used for execution of a single interpreted command.
4) M – the amount of additional memory for using the proposed program compression method.

It is important to note that last three parameters are the characteristics of a given implementation of the method (see next section).

The proposed method stores infrequently executed portions of programs in compacted interpreted form and dynamically executes them. According with infrequently code definitions, no more than $\theta \cdot T_{av}$ executed instructions are transformed. So, the count of executed instruction growths on $\theta \cdot T_{av} \cdot I$ instructions and the total execution time of compacted program is equal to $(1-\theta) \cdot T_{av} + \theta \cdot T_{av} \cdot I = (1+\theta \cdot (I-1)) \cdot T_{av}$. This time should be not more than $\tau \cdot T_{av}$ and $(1+\theta \cdot (I-1)) \cdot T_{av} \leq \tau \cdot T_{av}$. Therefore, in order to compacted program execution time not to exceed a given time limit, the threshold θ should be no more than $\frac{\tau-1}{I-1}$. Note that it is only guaranteed on average, because determining of infrequently executed code based on average execution frequencies of basic blocks.

The memory overhead resultant from a programs compression method use must not exceed the amount of memory saved due to code compression. Therefore it is necessary to select programs with total memory footprint exceeded $\frac{M}{1-\lambda(\theta)}$.

Then for using the proposed method it is necessary:

1) Choose parameter $\theta \leq \frac{\tau-1}{I-1}$.
2) Choose for compression programs whose total size more than $\frac{M}{1-\lambda(\theta)}$.
3) If the above conditions (1, 2) can not be concurrently met or a greater compression ratio should be gained, then it is possible to add the requirement to increase the system performance in $\frac{1+\theta\cdot(I-1)}{\tau}$ times.

If to follow the above recommendations on use of the proposed code compression method, it is guaranteed, that programs execution time will increase on average no more than in τ factor with the compression ratio $\lambda(\theta)$.

V. Method Implementation

The system of programs compression implemented the proposed method is written on the C++ language. It consists of two parts: program compression and compressed program execution (see Figure 2). The input for the system is a program for compression (written on C language), a distribution functions of program's input parameters, and maximum quota the program total execution time. The output is the compressed program and files with an offset table and compressed interpreted code.

The compressor transforms infrequently executed program code into the interpreted presentation and compresses it as a text. Based on distribution functions of input parameters the compressor determines average execution frequency of program basic blocks. The threshold and average frequencies calculated for a given program are used to determine fragments of infrequently executed code. The infrequently executed code are grouped that the memory overhead due to their compression would not exceed the code size reduction that can be achieved.

The decompressor consists of two main parts: the interpreter and a service of decompression of compressed interpreted code.

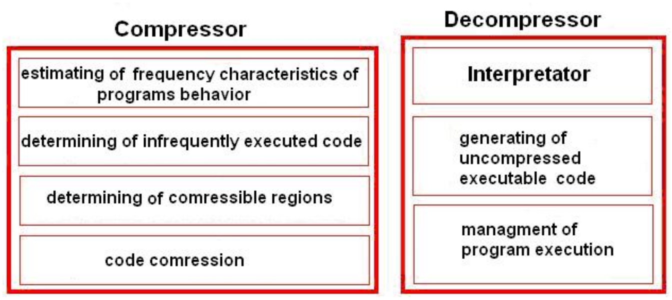
Compressor
estimating of frequency characteristics of programs behavior
determining of infrequently executed code
determining of comressible regions
code comression
Decompressor
Interpretator
generating of uncompressed executable code
managment of program execution

Fig. 2. Implemetation scheme

Programs for an on-board aircraft computer system were used for testing the proposed method implementation [14]. The aim of the work was to determine dependency between input parameter θ and the compression ratio achieved as a result of using the proposed method.

On each test program the system runs 10 times with different values of θ $(0.1, 0.2, \ldots, 1)$. The compression ratios $\lambda(\theta)$ obtained for each run were saved and averaged after finishing of all the experiments. This resulted in getting the following dependency presented in the Figure 3.

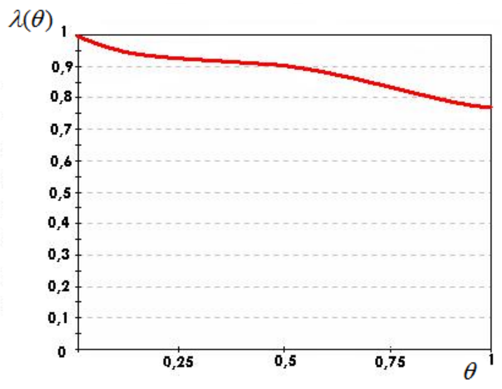

Fig. 3. Compression Ratio

So, for $\theta = 1$ the best compression ratio was achieved $\lambda(\theta) = 77\%$. For $\theta = 0.5$ the compression ratio $\lambda(\theta) = 89\%$ is achieved. For $0.2 < \theta < 0.4$ the grade of the compression ratio decreases. Therefore it is necessary to choose the input parameter from the recommended (given, specified) range.

VI. Conclusion

This article represents a program compression method based on the frequency characteristics of programs behaviour. The implementation of the proposed method has been written on the C++ language. Testing of this implementation was aimed to determine the dependency of the compression ratio on the input parameter for the system implemented the proposed method. For testing the system were used the real programs for on-board aircraft computer systems. The testing results prove the possibility of using the proposed method for on-board embedded systems.

The proposed compression method allows us to control the program execution time depending on the compression ratio. The mathematical dependencies guarantee that compressed program execution time will not exceed on average a given time limit.

It should be noted, that the proposed method is universal and can be used not only in embedded systems.

References

[1] Embedded Computing Design [HTML] (www.embedded-computing.com)

[2] *K. Kolpakov* History of onboard embedded systems in Russia // PCWeek, N32, 1999

[3] *A.M. Pavlov* Principles of organization of advanced onboard computing systems [HTML] (http://www.mka.ru/?p=41177)

[4] *B. Bus, D. Kastner, D. Chanet, L. Put, B. Sutter* POST-PASS Compaction Techniques // Communications of the ACM August 2003/Vol. 46, No.8

[5] *Sheayun Lee, Jaejin Lee* Selective code transformation for dual instruction set processors // ACM Transactions on Embedded Computing Systems (TECS), Volume 6, Issue 1, May 2007

[6] *B. Sutter, K. Bosschere* Software techniques for Program Compaction // Communications of the ACM August 2003/ Vol. 46, No.8

[7] *T.M. Kemp, R.M. Montoye* A Decompression Core for PowerPC // IBM Journal of Research and Development, Volume 42 Number 5/6, September, 1998

[8] *S. Seong, P. Mishra* Bitmask-Based Code Compression for Embedded Systems // IEEE Transactions on computer-aided design of integrated circuits and systems, 2007

[9] *S. Debray, W. Evans* Profile-Guided Code Compression. // Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, 2002

[10] *R.L. Smeliansky, D.E. Guryev, A.G. Bahmurov* About one mathematical model for calculation of programs behavior. Programming, N6, 1986

[11] *R.L. Smeliansky, T. Alanko* On the calculation of control transition probabilities in a program Inform. Processing Letters N.3, 1986

[12] *P. Brown* Macros without tears // Software: Practice and Experience. Volume 9, Issue 6, 1979

[13] *A.V. Shalimov* Method of determining execution frequency of programs basic blocks // Modeling and analysis of information systems, Volume 18, Number 2, 2010.

[14] DrTesy [HTML] (http://lvk.cs.msu.su/index.php/articles/65)