Survey of Code-Size Reduction Methods

ÁRPÁD BESZÉDES, RUDOLF FERENC, AND TIBOR GYIMÓTHY

University of Szeged

AND

ANDRÉ DOLENC AND KONSTA KARSISTO

Nokia Mobile Phones

Program code compression is an emerging research activity that is having an impact in several production areas such as networking and embedded systems. This is because the reduced-sized code can have a positive impact on network traffic and embedded system costs such as memory requirements and power consumption. Although code-size reduction is a relatively new research area, numerous publications already exist on it. The methods published usually have different motivations and a variety of application contexts. They may use different principles and their publications often use diverse notations. To our knowledge, there are no publications that present a good overview of this broad range of methods and give a useful assessment. This article surveys twelve methods and several related works appearing in some 50 papers published up to now. We provide extensive assessment criteria for evaluating the methods and offer a basis for comparison. We conclude that it is fairly hard to make any fair comparisons of the methods or draw conclusions about their applicability.

Categories and Subject Descriptors: A.1 [**General Literature**]—*introductory and Survey*; C.4 [**Performance of Systems**]—*performance attributes*; E.4 [**Coding and Information Theory**]—*data compaction and compression*

General Terms: Algorithms, Measurement

Additional Key Words and Phrases: Code compaction, code compression, method assessment, method evaluation

1. INTRODUCTION

Data compression in general is a field almost as old as information technology itself. There are many reasons for the importance of this issue that probably need not much further explanation. Still, one of the motivations is that the storage media has always been limited, although the capacity and its relative costs are constantly improving. However, the requirements for storing large amounts of data

Authors' addresses: Á. Beszédes, R. Ferenc, and T. Gyimóthy, Research Group on Artificial Intelligence, University of Szeged, Aradi vértanúk tere 1., H-6720 Szeged, Hungary; email: {beszedes,ferenc,gyimi}@inf.u-szeged.hu; A. Dolenc and K. Karsisto, Nokia Mobile Phones, P.O. Box 407, FIN-00045 Nokia Group, Finland; email: {andre.dolenc,konsta.karsisto}@nokia.com.

are probably increasing even more significantly. Hence, if the same amount of information can be stored using less space (i.e., in a compressed form), this—among other things—saves money. Another issue (especially in the "wired-up" world of today) is the limited bandwidth of communication channels. In this case, a compressed data set with the same information content is also preferable.

A special case of data compression is the compression (or more generally, size-reduction) of program code.[1] There are many reasons for making a distinction between data and code compression. One is the fact that if we have a certain amount of knowledge in advance about the structure of the code, it may be more compressible. Since the compression method is more specialized, it can include more specifics such as hardware implementation and specialized encoding strategies.

Code compression is very important in some areas where minimizing the storage requirements is even more emphasized. This primarily includes embedded (mobile, control) applications. In these areas another issue has recently become quite important too, namely the issue of energy-saving in these systems. Interestingly, this issue appeared to be related in many ways to code compression. The two most obvious examples for this are the observations that executing fewer instructions and accessing external memory devices less frequently can both reduce the energy needs of a system. Thus, several different approaches have been proposed in the literature concerning the correlation between energy saving and code compression. Hence, we will pay special attention to these issues here as well.

Over the years, many different theories and, more importantly, practical implementations have been elaborated and described as compression methods in the literature. It is not our goal to describe all of these. As a matter of fact, many good textbooks exist on this topic, for example, Nelson and Gailly [1995], Hankerson et al. [1997], and Bell et al. [1990]. However, we did not find any good articles that would survey the various code compression methods and offer some comparisons or, at least, some evaluations of these. (A good and extensive bibliography of code compression methods has, however, already been prepared by van de Wiel [2001].)

This article surveys some 50 articles devoted to code-size reduction methods published from 1984 to date, which are grouped into 12 principal methods along with several related works. We provide a general classification and assessment of various methods and evaluate various properties of theirs.

One of the objectives of this article is to help the interested reader in choosing the most appropriate methods (or their combination) for a specific need. This may seem a rather challenging task, but we feel that a survey like this can be of great help in such situations.

Since code-size reduction covers many techniques and application specifics that could be investigated from several aspects, we tried to emphasize the *space of applicability* of the methods. In particular, the need for hardware-based modification is discussed.

Naturally a survey cannot hope to cover every possible issue of the subject. Therefore, we tried to concentrate on the above-described objectives and evaluation aspects. So, let us summarize what this article is not about. First of all, this survey is not meant to be complete. It is meant to be diverse in a sense to incorporate as many different kinds of methods as possible in order to give an overall picture of the recent results and the state-of-the-art of the subject. This article is not a textbook for providing an introduction to code compression and an overview of commonly accepted methods. We do not wish to make direct confrontations of competing methods; we only give direct comparisons where it is absolutely feasible. Finally, we did not include those methods to the survey that are partly related to some of the

[1]The notion of *program code* should be taken in a broadest sense, including source code, various types of intermediate representation and machine code as well.

methods discussed, but which do not directly deal with code-size reduction (e.g., other energy saving methods).

During the preparation of this survey, we encountered several problems concerning the survey methodology itself and the assessment and evaluation of the methods. We feel that it is important for the reader to bear these in mind when evaluating the methods discussed. Namely:

—A comparison of methods is not always representative and is rarely feasible anyway. We make direct comparisons only where applicable, which means that the measurement and other assessment results must be "compatible," meaning that the environment of the measurement was the same.

—Naturally, we could not examine every available method because the list of related publications is rather long (see, for example, van de Wiel [2001]). Besides, some important publications may have escaped our attention. Nevertheless, we tried to make this survey as representative as possible.

—In Section 2, we provide definitions of terms and basic algorithms/techniques that we will use or refer to throughout the article. This is required because the terminology of the subject literature is currently far from being consistent regarding the terms and notions. Hence, we tried to gather together the most widely accepted terminology.

—The assessment of the methods along classification lines is, of course, a difficult issue. In many cases, it is not possible to make a strict distinction of whether the method is hardware- or software-based (a traditional classification model), say. Thus, we chose a classification which represents the properties of methods in several dimensions as described in Section 3.

—One published article does not always cover one distinct method, which means that a set of related papers may describe the same method (or its derivatives) or that a certain publication may contain a description of more than one distinct method. Therefore, our assessment (in Section 4) is organized around distinct methods and not according to individual articles.

—We would also like the reader to be aware of the following. We present some quantitative and/or qualitative measurement results that were given in the publications using different numerical data (e.g., compression ratios, execution times). At certain places (mostly in Section 5), we apply these pieces of data to evaluate the various methods (and sometimes to compare them with each other). We cannot take any responsibility regarding the correctness of this data since we could not test and evaluate every implementation of the methods investigated.

This article is organized as follows: Section 2 provides some necessary background information to make the survey and the articles easier to follow. In Section 3, we discuss in detail our criteria that will be used for the assessment and classification of the code-size reduction methods—the theme of Section 4. The results of the assessment, comparison, measurements and their evaluation are given in Section 5. We round off the article with a summary, an evaluation of the survey methodology and list some conclusions as well.

2. BACKGROUND

This chapter furnishes some background knowledge in order to be able to discuss the methods presented in this survey in a consistent manner. First, we briefly overview some basic definitions related to compression.[2] Then we describe how these general concepts can be applied to compressing program code, emphasizing the important differences. Finally, we overview some of the most important basic algorithms for compression, which are referred in many of the publications in this field.

[2]Note that we will use a slightly narrower definition of compression throughout this article. This is described later in this section.

2.1. Definition of Terms

This section contains information that defines and clarifies concepts utilized in the rest of the article. It will be of particular benefit to those unfamiliar with compression techniques.

2.1.1. Compression. The most general and straightforward definition of the term "compression" (or the process of "compressing") could be given as: *"storing data in a format that requires less space than usual."* By *storing*, we mean any kind of representation of the data including its transmission (this is usually referred to as the encoded data). *Data* in the previous definition should be read in a most general form (including program code). Finally, by *usual*, we mean the unaltered input of the compression process.

In this survey, we use the term "code compression" for reducing the size of *program code* (either source-, machine- or other) by its equivalent (lossless) representation in another form. Usually, this means the application of various compression methods based on some statistical information that reduce the entropy of the input (see the next paragraph).

Analogously to compression, the term *decompression* is used for the process of applying it on a compressed data set in order to obtain the original input sequence.

2.1.2. Theoretical Background for Compression. The theory behind compression is based on results of *information theory* (see, e.g., Hankerson et al. [1997]). In this section, we will then review some terms from information theory that are needed in our further discussion. Note that these definitions are not mathematically precise, but they should be sufficient for grasping the basic processes involved in compression.

We define the input for the size-reduction method (in our case program code) as a *sequence* of input symbols. This sequence might be simply the bits or bytes of the input file or it might be bigger entities like tokens for compression. The input sequence may contain values from a fixed set of symbols. The basic idea behind most compression algorithms is to represent each input symbol with a *code* that will, with the other codes, produce an overall smaller encoded sequence.

Each symbol in the input has a certain *probability* value, which is in the simplest case the frequency of occurrence of the symbols in the input. Most compression methods exploit this attribute of the input symbols to produce a smaller output sequence (by encoding the most probable symbols with short codes). More generally, the sequence X is a random variable x with a set of possible outcomes (input symbol values), $A_X = \{a_1, a_2, \ldots, a_i, \ldots, a_N\}$, having probabilities $\{p_1, p_2, \ldots, p_i, \ldots, p_N\}$, with $P(x = a_i) = p_i,\ p_i \geq 0$ and $\sum_{x \in A_X} P(x) = 1$.

The size-reduction of a sequence X is obtained by utilizing its *information content*, that is, if there is less information stored by the sequence we can represent it in a shorter encoded sequence. In other words, the information content is the *uncertainty* of the sequence: the more information is "squeezed into" the sequence the more "uncertain" we are about predicting the symbols, that is, coding them efficiently. The information content can be measured by the *entropy* of X using the following formula:

$$H(X) = -\sum_{x \in A_X} P(x) \log_2 P(x)$$

Basically, this formula gives the minimum average number of bits required to encode the symbols in the given sequence. If $H(X)$ is multiplied by the number of symbols in the input sequence, we get the theoretically minimal size (in bits) of the encoded sequence that can be obtained.

2.1.3. Compression Model. As remarked previously, most of the various compression methods are based on exploiting the information content of the input sequence that is to be compressed. This information content is modeled by various statistical computations on the input sequence like the probabilities of the symbols.

To utilize the knowledge about the amount of redundancy (and entropy) in the input and to obtain the best results (best compression ratios), two tasks must be cleverly performed: create such *models* (1), which—by utilizing the statistics of the input—can "guide" the coding process most productively by assigning suitable *codes* to the symbols (2) Nelson and Gailly [1995].

Hence, most compression methods contain a separate modeler and coder. These two can be treated as separate topics for research, but they are usually fine-tuned for each other. The decoder, on the other hand, implements the inverse operation while using the same data structures. It is possible to use the same coding method with different models that yield different results.

Modeling can mean anything from the simplest probability by counting the frequencies of the symbols to sophisticated model creation methods. (These are described when we dealing with the corresponding methods, but some discussion about the most important modeling techniques is presented in Section 2.3.)

Similarly, there are many different ways of coding the symbols, such as Huffman and arithmetic (see Section 2.3) coding.

Theoretically, for any compression algorithm there must exist an input that produces a larger result (if all inputs generated smaller outputs, there would then exist an output corresponding to two distinct inputs). The trick is to use a model that maximizes the probability of detecting redundancies in the input data, and one should know beforehand that inputs that lead to poor compression ratios would not happen in practice, that is, inputs with no or little redundancy of the kind we can detect should not occur in practice.

2.1.4. Compression Ratio. We use the following definition of the *compression ratio*. The compression ratio R is defined as c/C where c is the size of the compressed output and C is the size of the input. When expressed as a percentage P then $P = (1 - R) \cdot 100$. For example, a compression ratio of 40% means that the result now occupies 60% of the original space, so percentages indicate the amount of space saved. This survey will employ this convention, but some articles use the definition $P = R \cdot 100$. In addition, it is not always clear in the literature if the ratios include the data needed for the decompression like code tables. Therefore, the comparison of different compression ratios is not always possible.

In the literature, the ratios are presented and interpreted in many ways. The given number can represent the worst, average, or best case. This is one of the problems that we face that makes a unified evaluation of the methods difficult.

2.1.5. Compaction. Compacting the program code (or other type of sequential data) differs from the method of "compression" described above in that it creates an (equivalent) transformation of the input using the same form. The most significant difference with compression is that there is no need to decompress the code since it is directly interpretable by the client (there is no difference between the type of the uncompacted and compacted data). Bell et al. [1990] describe compaction as "irreversible compression" meaning that once the data is compacted its original version cannot be restored (since there is no decompression at all).

The basic methods that are involved in code compaction are some of the traditional compiler optimization methods (such as the elimination of useless code) and code factoring (by which repeating fragments are factored out). These are described in detail by the corresponding methods.

In the field of code-size reduction, these compaction techniques are typically applied to machine-code programs.

2.2. Compressing Code

Many compression algorithms can be applied to whatever type of input data—including various types of program code (source-, intermediate-, machine-, etc.). However, compressing program code

differs from general data compression techniques in several important respects.

First of all, we must distinguish the two basic principles by which the code-size reduction is achieved: code compression and code compaction (see the corresponding sections above).

Next, all types of program code have some special features that are not common to other types of input data or to general data. These features arise from the nature of program code like a hierarchical structure or repeating fragments. If the compression algorithm is aware of these things it can produce much better results. Furthermore, it is not uncommon that some types of program code, such as machine code, contain many redundancies. Of course, the compaction methods can also take advantage of various semantic characteristics of the input code.

A serious constraint with code compression methods is the fact that some data compression techniques (referred to as "lossy" as opposed to "lossless") trade-off accuracy with compression ratios so the output after decompression is not identical to the original input. Evidently, code compression does not permit such trade-offs.

A further drawback of applying special code-size reduction methods can be that, in many cases, a certain method is applicable only to a fixed class of code types.

Probably the most important differences between general and code compression methods, however, come from the fact that program code (especially machine code) is always stored, manipulated and executed on hardware devices. Hence, many techniques are implemented by hardware means. Most commonly, however, the software- and hardware-based solutions are intermixed in order to produce optimal results. Hence, keeping this in mind, there are some important aspects of which one must always be aware when implementing or investigating a code-size reduction method. These are:

—The amount of RAM needed for decompression is extremely important because it is usually restricted. There is no point in applying compression if too much RAM is then needed at runtime for decompression.
—Because of jumps and function calls, random access is required, whereas traditional data compression techniques permit only sequential access.
—Though the compression can use as much number crunching as needed because it is done "off-line," decompression must be very fast.

Although our focus is on code compression, some basic concepts and algorithms utilized in data compression as well are presented next.

2.3. Short Description of Basic Algorithms

In this section, we will give an introduction to some of the most important algorithms that have always been used for compression. The purpose of this section is to include the necessary background knowledge so that the survey and the articles can be understood without needing a reference textbook. Very good textbooks already exist on this topic, for example, Bell et al. [1990], Nelson and Gailly [1995], and Hankerson et al. [1997].

As we mentioned earlier in Section 2.1.3, most of the compression methods may be modeled as containing a (somewhat separate) modeler and encoder. In this section, we cannot overview both the modeling and coding issues because they are so diverse and, moreover, for several (mostly simpler) compression methods they cannot be really separated from each other. Whenever a compression scheme is presented in the literature, the authors usually compare it with another scheme. In the following paragraphs, we can only give a brief overview along with the main references to some of the most popular coding schemes. Hence, we will provide only a description of the most important coders/compressors, while modeling is only briefly mentioned here.

2.3.1. Modeling Techniques. The basic task of a modeler is to produce such *models* that can guide the encoding

process to assign the most suitable code-values to the symbols of the input sequence.

The model in its most simplest interpretation consists of statistical information obtained from the input sequence. In the simplest case, it contains probability values for the symbols based on their *frequency* of appearance. If these values are commonly used for the whole sequence, then we refer to it as a *static model*. On the other hand, if this model is constantly updated based on the portion of the input seen so far, it is known as an *adaptive model*.

It is also common to classify modelers in accordance with the number of symbols that are taken into account for computing the probability values. Based on this, we can talk about order-0, order-1, . . . , order-n models.

There are, of course, also very sophisticated modelers that compute the probabilistic distributions in more complex ways. For example, using different machine-learning methods to predict the input symbols and based on it the probability distributions. An example for the modelers is the Markov modeler.

Note that the best models could be produced based on the whole sequence and computing, for example, the minimal entropies. However, solutions of this kind are computationally very demanding.

2.3.2. Huffman Coding. Huffman coding is one of the oldest and most important coding techniques, due to its simplicity and effectiveness. Huffman published his algorithm as early as 1952. Since then, many variants have been proposed for a wide range or purposes. A good description with examples can be found in Nelson and Gailly [1995].

The idea behind this method is to use the shortest sequence of bits for symbols that occur most frequently, and the longest sequence for those symbols that appear the least. The first problem then is to find the frequency in which each symbol appears. Next, a method for assigning codes to symbols must be devised. Finally, a suitable data structure must be found that is economical and efficient.

Huffman coders can be quite fast, but perform badly when probabilities differ significantly from a power of 1/2, and decoders can be implemented in more than a dozen different ways, each with its own speed vs. size trade-off. Hence, Huffman coding usually refers to a family of coding schemes.

Some care is needed before using Huffman coding for code compression because the basic algorithm does not allow for random access. Consider a jump instruction. Once one determines which bit to start executing next, the next problem is to properly initialize the decoder. Normally, one would need to start decoding from the beginning of the data until one reaches the given bit. Therefore, Huffman coding is easier to use when entire functions are decompressed at a time, but it can be used effectively with blocks of code as well.

2.3.3. Arithmetic Coding. The principles of Arithmetic coding are given in this survey because this coding is used in several papers we study in the following: Moreover, it performs better than Huffman coding in some circumstances. Many descriptions of arithmetic coding can be found (e.g., Bell et al. [1990]) as well as different variations and implementations [Witten et al. 1987; Howard and Vitter 1992]. In addition, Nelson's book [Nelson and Gailly 1995] has a very good description of this method as well.

The arithmetic coding algorithm uses probabilities directly instead of frequencies because they are real numbers. The probabilities are used to split an interval, usually [0, 1], into ranges in proportional sizes, each symbol being assigned to one and only one range. The encoding itself entails keeping track of an interval.

The arithmetic needed for the implementation is very simple since it only requires trivial linear transformations such as mapping a number from one interval to another one. Thus, making it

suitable for applications where the difficulty of implementation is very important such as hardware-based implementation. Although the method is based on real numbers, most of the practical implementations can be be optimized using integers only with implied decimal points.

A drawback with the original version of the algorithm is that the entire message must be read before it can be encoded or decoded. An excellent reference that examines arithmetic coding and includes C code of different versions can be found in Witten et al. [1987].

2.3.4. Dictionary-Based Methods. Dictionary-based coding schemes cover a wide range of various coders and compressors. Their common feature is that the methods use some kind of a dictionary that contains parts of the input sequence which frequently appear. The encoded sequence in turn contains references to the dictionary elements rather than containing these over and over. There are many different variations of this approach, so it would be impossible to mention all of these here.

One example of a simple and effective dictionary method is the *Move-to-front (MTF) coding* [Bentley et al. 1986]. It maintains a list whose elements are ordered so that the first element is the most recently accessed element. This has in turn the effect that a sequence with high spatial locality tends to yield a sequence of small indices, which should compress well [Ernst et al. 1997]. On the other hand, MTF coders are reasonable solutions for dynamic data which cannot be analyzed off-line, or (parts of) data too complex to model. In other cases it may render a significant coding loss, that is they perform poorly.

Another example of dictionary methods is the Lempel—Ziv family of compression algorithms [Lempel and Ziv 1976; Ziv and Lempel 1977]. This baseline method evolved into numerous variations and different implementations, all of which cannot be mentioned here.

3. ASSESSMENT CRITERIA

The usual task of a survey article is to give an overview of the different methods available for a certain class of problems along with a possible classification and practical performance- and other measurements. In the case of code compression methods, the "usual" classification is as of a *hardware*- or *software*-based solution. In addition, the typical (and surely most important) aspect of a method's performance is its compression capability, that is, the compression ratio.

However, the properties of code compression methods are more complex, so we propose another assessment, as described in this section.

The methods that characterize themselves as "code compression methods" utilize different principles for achieving this goal. Of course, their motivations and goals may also differ, so the applicability of the methods is diverse as well. As a result of this, their effects on the efficiency of compression (expressed as the ratio) and other aspects such as compression/decompression time—that are, by the way, neglected by many methods—are different. In many cases, this difference cannot be accurately measured and compared to each other.

Based on the above, we chose to make two clear separated assessments:

(1) The first assessment is to *classify* the methods into one of the most commonly appearing groups according to certain aspects.

(2) Afterwards, the second assessment contains the actual evaluation of how the methods *perform* by measuring their effects on various things.

For the sake of simplicity and uniform treatment of the subject, we will introduce short, three-letter notations for each (classification and effect) aspect. If these aspects have predefined *values* they are represented by numbering the notation letters.

In the following, the two assessments are presented in detail.

3.1. First Assessment: Classification of the Methods

As noted earlier, classification is difficult because there may be many aspects suitable for the grouping. We found that the traditional classification as hardware- or software-based method (primarily regarding the implementation of the decompressor) is inadequate and in many cases the selection is not even possible unambiguously. However, this type of grouping is still presented in our third classification—CDE, see below. We define these six grouping methods—which are basically orthogonal *dimensions*—according to which the method can be unambiguously[3] put in the space of various method types:

(1) grouping by the basic size-reduction principle (CPR)
(2) assessment to the type of the applicable hardware (CHW)
(3) grouping by the implementation of the decompressor (CDE)
(4) grouping by the code equivalence (CCE)
(5) grouping by the type of the subject program code (CCT)
(6) grouping by the granularity of the subject program code (CGR)

In the following, these are described in detail.

3.1.1. CPR Classification (by Size-Reduction Principle). This dimension of the classification is taken as the primary aspect in our survey; the assessment of the methods in Section 4 is also ordered along this dimension. The purpose here is to classify the subject method according to the basic *principle* that is used to achieve the code-size reduction. We identified the following principles as the most typical:

CPR-1: without a decompressor (compactor)
CPR-2: decompressor is required
CPR-3: interpreter is required (or a mix of the previous two)

Methods of the first type (CPR-1) do not need a decompressor at all, the result of the size-reduction being a compacted version of the input that is of the same form (see Section 2.1.5 for details). CPR-2, the second type, is the most common and it describes methods which require some kind of a decompressor to get the original input (either software- or hardware-based and either trivial as a table or complicated as a decision tree). The third type is reserved for the methods that cannot (or only with difficulty) fit into one of the other two. It is typical of those methods that do not require a decompressor but some kind of an interpreter to get the original code (the code is not really compacted but is transmitted in another form that can be directly interpreted by the receiver).

3.1.2. CHW Classification (Applicable Hardware Type). This classification somewhat differs from the others in that it does not really groups a method into a certain class. Instead, all of the possible values of this classification are considered as showing whether the method is suitable for a specific class or not. We will also give our classification according to this with the method descriptions. The following basic hardware types are used in our assessment:

CHW-0: classification is not applicable
CHW-1: System-On-a-Chip/embedded systems
CHW-2: mainframe computers

Methods for which the application domain is not relevant/applicable (e.g., not a complete method) will get the code CHW-0 assigned to them. The other two hardware types are self-explanatory. If both are suitable, the method is probably not hardware-specific, that is, it can be applied on any hardware type.

3.1.3. CDE Classification (Implementation of the Decompressor). With this classification, the methods are grouped according

[3]The second classification—CHW—does not uniquely assign a method to a particular hardware type but, rather, states whether it is suitable for each of the types or not.

to the method of its implementation; as in software, hardware or both. The classification regards particularily the *decompressor*, since the compression phase is in most cases purely software-based and typically it is not a relevant factor for evaluating the code compression method. Of course, for methods where there is no need for a decompressor (CPR-1), this classification is irrelevant (gets a value CDE-0).

The implementation of the decompressor (if there is any) can be classified according to the *hardware* environment in which it can be applied. In some cases the methods do not explicitly define the hardware on which they operate or the method can be designed to be purely software based (CDE-0). It is, however, important when the method exploits some kind of hardware configuration (such as the CPU-cache configuration) or when it proposes a new hardware logic configuration that is combined with the compression and/or decompression process.

In our survey, we defined the entries for this classification based on the typically appearing configurations in the methods. Note that this was not so straightforward as many methods do not report clearly which hardware they operate on. On the other hand, some methods explicitly define a new approach for the hardware modification of the system for the decompression phase. We outlined the following most typical hardware architectures for implementing the decompression:

CDE-0: software-based or don't care

CDE-1: don't care, but extra memory (RAM) is needed

CDE-2: hardware decoding by the modification of the CPU

CDE-3: hardware decoding on architectures without cache (CPU is left intact)

CDE-4: hardware decoding on a pre-cache architecture (CPU is left intact)

CDE-5: hardware decoding on a post-cache architecture (CPU is left intact)

The methods that are classified as type CDE-0 are purely software-based, can be implemented using any hardware configuration, or they do not need decompression at all (they are of type CPR-1). Methods of type CDE-1 are basically purely software-based as well, which means that no hardware-related modification is required. The only difference is that extra (RAM) memory is needed in order to decompress the program in the CPU (hence, some applications—such as embedded systems—could not utilize these methods). CDE-2 classifies those methods where the decompression is done by a hardware modification to some part of the processor (i.e., its address generation logic or instruction set). Methods of the next architecture, CDE-3, require some kind of decompressor hardware module, which is placed, for example, between the CPU (which is not modified itself) and the memory, and there is no instruction cache involved, while methods of type CDE-4 and CDE-5 use a decoder between the memory and cache or between the cache and the CPU, respectively.

3.1.4. CCE Classification (by Code Equivalence). This classification is related to CPR, moreover it could be seen as part of that classification. However, it is quite crucial for certain type of applications and hence the new classification category.

The code equivalence classification groups the methods based on to what extent they preserve the layout and functionality of the original program code in the decompressed code. This attribute is, in most cases, directly dependent on the code-size reduction principle: compactors produce equivalent but different code in their layout, while most of the decompressors produce completely equivalent code. However, in many practical applications this kind of strict distinction cannot be made because, for instance, the decompressor must make use of additional computations due to the modification of the cache architecture. We classify the code equivalence in the following way:

CCE-0: the decompressed code is completely equivalent

CCE-1: the decompressed code is equivalent as seen by the CPU

CCE-2: the (decompressed) code is functionally equivalent

The first type (CCE-0) is typical of methods which use some kind of code compression to produce the compressed data and the decompression is performed before the execution of the program. This way, the original input program is gained. The methods of the second type, CCE-1, also produce equivalent code but that is seen equivalent only by the CPU. The code itself (as a memory image) is not completely identical to the input code because it may include some additional information that aids the code execution mechanism (e.g., cache refill engine). These modifications may involve some helper instructions computed on-the-fly or some address relocation information. Finally, those methods for which the decompressed[4] code is equivalent only in its *functionality* to the original code (the instructions and/or their layout is modified) we will refer to as type CCE-2. This type may also involve methods where there is really no "input stream" for compression, but the compressed representation is prepared from another representation (e.g., source code).

3.1.5. CCT Classification (Code Type). This classification addresses the *type* of input code on which the method operates. (It is more appropriate to define this as the type of the final *decompressed* code, since there are some methods where the input for compression is in another form.) Most methods are designed based on specific requirements regarding its input data, which can be classified in a simple way. However, with some methods the type of the input code is not relevant (especially if the modeling is the primary result of the method).

There are some methods that are not primarily designed for compressing program code but some other type of data like text. However, there are no obstacles to apply this kind of method to code compression as well. This is especially true when, with relatively simple modifications, it can be adapted to code compression.

Our code type classification applies the following codes:

CCT-0: non-code data (e.g., text)

CCT-1: source code

CCT-2: IR (intermediate representation)

CCT-3: assembly

CCT-4: machine

CCT-5: special type

The meaning of the codes is self-explanatory. However, when some more precise information concerning the code type is available (e.g., source code language) then it is presented with the method description. The last type describes those code types that are more specific, such as machine code for a specific family of processors or instruction sets.

3.1.6. CGR Classification (Granularity of the Code). This classification notes the *granularity* of the input code when compression (decompression) is applied. It is used to classify the size of the program code units that are compressed (decompressed) as a whole. Some methods do not explicitly define this property, but it is clear in most cases.

The following granularity values are used:

CGR-0: irrelevant

CGR-1: bit sequence

CGR-2: character

CGR-3: instruction

CGR-4: block (basic-block and/or cache-line)

- *CGR-4.1* small (e.g., cache-line)
- *CGR-4.2* large (e.g., page)

CGR-5: procedure

CGR-6: translation unit

CGR-7: program

Those methods for which the granularity is irrelevant (e.g., can be applied to any level of the granularity, or no unit is

[4]Or the compacted code, since there is no decompression, therefore the compacted code itself is the "decompressed" code as well (see 2.1.5).

used for the size reduction) are of type CGR-0. The second type addresses the smallest unit for compression, a bit sequence (which is usually smaller but it can also be bigger than a byte). The next one models a character (a byte) from the input code while methods which compress one instruction at a time get the classification CGR-3. The next type, CGR-4, corresponds to those compressors which use blocks of code as its compression units (e.g., basic-blocks or cache-blocks). Where relevant, more precise information is given about the actual size of the blocks (concrete values, if possible): CGR-4.1 stands for small blocks such as basic blocks or cache-blocks, while CGR-4.2 stands for larger blocks such as memory pages. With CGR-5, the compressor compresses one procedure at a time, which is then also served as the unit for decompression. Finally, the last two types represent the largest compression units: translation units (in the case of some code types, this could mean, e.g., classes) and the whole program, respectively.

3.2. Second Assessment: Effects of the Methods

In the previous section, we described our approach for the classification of the methods. Besides the assessment for classifying the methods in a structured way as seen before, one of the most important tasks of this survey is the investigation of the actual *effects* of the code-size reduction methods.

The first and probably most important question is how the method affects the *size* of the resulted compressed data, that is, the compression ratios it can achieve. However, there are a number of other aspects that need to be investigated and which can be vital in some applications. For example, the decompression complexity is relevant where the decompression needs to be implemented in hardware.

For the assessment and evaluation in Sections 4 and 5, we will make use of specific notations for the values of the effects. These are described at the beginning of Section 4.

We investigated the effects of the methods on the following:

(1) ECS: size of the (compressed) code
(2) EEX: number of executed instructions
(3) EES: execution speed
(4) ECT: compression time/complexity
(5) EDT: decompression time/complexity
(6) EBE: behavior safety
(7) EEN: energy consumption
(8) the mutual effect of the above

In the following these are described in more detail.

3.2.1. ECS Effect (Code Size). The effect of a code-size reduction method on the size of the resulting code is basically a measure of its size-reduction capability, which is the principal merit of a compression/compaction method. It can be expressed as the *compression ratio* like that described in Section 2.1.4.

Although the compression ratio is the most important measure of the goodness of a code-size reduction method, it is not unusual that this feature is treated with insufficient importance by the authors of a publication describing the method. This means that the precise definition is not given how to interpret the exact values given for the experimental results. It also makes it difficult to compare the effectiveness of different methods with each other in an objective way.

The following problems can arise when investigating the compression ratios:

— *What unit is used*? Relative value in the range [0, 1] or expressed as percentage or as bits-per-character, etc. (Note that these are easily convertible and a uniform representation can be given.)
— *Does the size of the compressed code incorporates somehow the information needed for decompressing*? This is important because the method can "cheat" when giving a surprisingly good compression ratio, but with enormous overheads regarding the size and/or complexity of the decompression engine (the "lost" information is incorporated

into the decompression model). In many cases, this can result in a misleading assessment of the method. Hence, it would be good to know whether the compression ratio incorporates (1) the size of the modeler (see Section 2.1.3) and (2) the size or complexity of the decompression engine. This information cannot always be accessed.

—*What was the test-case for the given values*? Is it an average, best case or worst case value? For certain kinds of applications one of these is preferable, however, the comparison of different methods the average value is the best (given that it is computed based on sufficient number of test cases).

—*Does the effect on the resulting code-size depend on the environment where the method is applied*? Or does it involve other aspects like parameterization capability of the method, type (or a certain class) of input data, dependence on other effects or other external features such as hardware architecture, implementation issues?

Based on the above, we can safely say that our survey may not fully reflect the capabilities of the methods in a real environment. However we made certain assumptions when investigating the given ratios and, where possible, we tried to assess the methods only where it was "fair to do so." In addition, we attempted to express the given numeric data in a normalized value representation.

Throughout the article, we will use the relative value notation and assume that the compression ratio does not include the model and decompressor and the given ratios are for an average case. Of course, these assumptions will make the overall assessment imprecise, but we will show how, with the corresponding methods, the given ratios have been calculated.

3.2.2. EEX Effect (Executed Instruction Number). As seen earlier, some code-size reduction methods can produce modified program code via decompression. This is especially true for code compaction methods where the compacted code itself does not actually need to be decompressed but is essentially modified in order to be smaller (see classification CCE).

Although the modified code is functionally equivalent to the original input program, the actually executed instructions will be different from those in the original code. For some applications it is important to know whether the number of these instructions is greater than with the original program. Hence, the EEX effect shows the *runtime performance* of the method. Of course, the performance is not always affected just by the number of the executed steps but may be affected by other eventual activities involved in decompression (see EES).

3.2.3. EES Effect (Execution Speed). This effect is partly related to the previous one because it mostly depends on the instructions executed at runtime. However, a method's effect on the execution speed is not always due to the modification of the code, but to some additional computations beside the decompression itself which are required by certain methods. This would include some additional activities (possibly additional hardware cycles) required to perform the decompression.

3.2.4. ECT Effect (Compression Time/ Complexity). Different compression and compaction methods require different computations for creating the reduced-sized code. It is not unusual that in order to gain a high compression ratio, the required computations must be very complex (in many cases more complex than for the decompression). Sometimes complexity is an important issue. Moreover, this also involves the *time* required for executing the compression algorithm.

3.2.5. EDT Effect (Decompression Time/ Complexity). As for the effect of compression complexity, the complexity of the decompression may also be an important factor. In many methods it is even more relevant because the decompression needs to be performed on-the-fly or prior to the execution of the program. Moreover, the

compression is usually performed only once but the decompression process is executed many times. Hence, the decompression algorithm is designed to be more effective.

Another important issue regarding the complexity of the decompression algorithm is that in some hardware-related methods it is implemented by hardware means. In such cases, the complexity in terms of the required computations should be minimized.

3.2.6. EBE Effect (Behavior Safety). As described for the EEX effect above, some methods produce modified code as the result of the decompression (or compaction). Apart from the runtime efficiency of such a modified code, perhaps it is more important that the code should preserve an equivalent *behavior*.

This issue is very important for some kinds of codes such as embedded systems machine code owing to timing and runtime issues. This behavioral safety can be further divided into (1) external behavior (functional equivalence) and (2) internal behavior (e.g., whether the resource requirements increase or not). Thus, in applications where the modified code is safety-critical, this effect is very important.

3.2.7. EEN Effect (Energy). The effect of a method on the energy consumption of the platform executing the reduced-sized code is not a primary aspect in the majority of methods (it may even be irrelevant or unmeasurable[5]), despite the fact that in some applications it is a very relevant issue. However, there are a number of methods which were designed with energy-saving aspects in mind. This survey pays special attention to these methods regarding the energy issues.

[5]It is said that the energy saving is a "system-wide exercise," meaning that investigating only the size-reduction capabilities of a method probably does not give an overall picture of energy issues of a hardware/software system. However, producing such reduced-sized codes using a method that can influence the energy saving is of crucial importance in some cases.

There are a number of issues which may influence the energy-aspects of a method such as the number of executed instructions, the type of the executed instructions (e.g., which memory types are accessed) or the specifics of the hardware used (cache organization, etc.). However, energy mostly depends on the hardware architecture used for decompression and/or interpretation of the compressed code, which means that the compression ratio may not always be the key factor in energy issues. Generally, the effect on energy depends on many things and, more importantly, it comes in conjunction with other effects, as described in the next section.

The energy issues are discussed in more detail in Section 5.4.

3.2.8. Mutual Effects. In some circumstances, it is beneficial to separately investigate one or more of the previously described effects of a code-size reduction method. Still, the different methods are in many ways related to each other and, moreover, it is not uncommon that a specific effect is directly dependent on another.

The most important fact is, however, that some of the effects mutually influence each other in terms of trade-offs or followings between them. The first type of influence means that if a method can be adjusted to be more effective by a certain aspect, other effects may be degraded. For example, it is very common that those methods, which can produce high compression ratios usually require complex computations during compression (e.g., for creating the best model). A similar issue is trying to strike a balance between the compression ratio and decompression time/complexity, that is, if significant reductions can be achieved by simplifying the compression/decompression algorithm and inevitably losing some of the compressability it is an affordable trade-off. Another example is that some of the compaction methods can produce smaller code by code factoring methods (see the appropriate sections) but the price is that the number of executed

instructions increase due to the code factoring.

On the other hand, the benefits of some effects are positively dependent on each other, that is, their values change in parallel. For example, the energy consumption is usually proportional to the number of executed instructions (although it may depend on other factors as well). Another example could be that higher complexity of the decompression may involve increased execution times due to additional hardware cycles, say.

Based on the above, we may conclude that the effects of the methods should be investigated *jointly* in order to discover their mutual correlations. In our survey we will try to emphasize the most important mutual effects of the methods with their evaluation.

4. ASSESSMENT OF THE METHODS

In this section, we briefly describe the methods investigated. In the case when a method is described in more than one publication, we chose to elaborate these publications jointly, when it is possible. Sometimes a publication also contains descriptions of more than one method; in such cases, we separated them as different methods. At some places other related methods are briefly described as well; unfortunately not all of the available methods could get a dedicated section in this survey.

After the short description, we briefly assess the methods by giving a distinct classification according to our first assessment as described in Section 3.1 and by giving some preliminary results of the effects as described in our second assessment in Section 3.2. A more detailed assessment of the effects and comparisons with other methods (where appropriate) are given in Section 5.

The methods investigated are organized around the classification CPR (which describes the basic compression principle, see Section 3.1). Within this classification, for those methods that require a decompressor (type CPR-2) they are ordered according to the relative complexity of the decompression (if this can be determined): beginning with trivial hardware implementation to more complicated hardware logic and the finishing with primarily software based methods.

The meaning of the special notations in the *Effects* tables is the following. A dash (—) means that the specific effect is not applicable to the corresponding method and the question mark (?) indicates that the effect is unknown. $++$ means "increase", while $--$ means "decrease" (in the case of EES "speedup" and "slowdown," respectively). The empty-set sign ($\emptyset$) denotes that the method has no effect on the corresponding quantity. We use a three-level evaluation of the ECT and EDT (compression/decompression time and complexity) effects, using L as "low level," M as "middle level" and H as "high level" (note that these are rather subjective values). For EBE (behavior safety), a three-fold classification is used as well ($\emptyset$ means that there is no safety risk, M means that in some circumstances the risk may be significant, while H means that the behavior is only functionally equivalent). $\sum$ corresponds to the discussion about the mutual effects.

4.1. Cooper and McIntosh's code compaction methods

Authors	K. D. Cooper and N. McIntosh
Affiliations	Rice University, Hewlett-Packard Corporation
Year	1999
References	[Cooper and McIntosh 1999]

Cooper and McIntosh [1999] presented a code compaction method (referred to as a compression method) involving several techniques. Their compaction framework is integrated into an optimizing compiler, hence the presented techniques are elaborated for RISC-like IR code, but it could probably be applied to RISC machine code as well.

The principles of Cooper and McIntosh's method are built on the results of C. W. Fraser from 1984 [Fraser et al. 1984]. The basic approach is to find repeating fragments in the input code that are then replaced by control transfer instructions to a

reference instance of the fragment. ($K-1$ of the total K identical instances of the same fragment are replaced with references to the last remaining fragment.)

The code-size reduction is achieved by producing directly executable code using various types of transformations on IR code (aka. code compaction). They report size savings of up to 15% with an average of 5% on already optimized code by classical compiler optimizations. (Fraser achieved 7% on VAX assembly code.)

The basic techniques of the subject algorithm consist of code transformation methods commonly known as *code factoring* (see 2.1.5). More precisely, they apply *procedural abstraction* and *cross-jumping* (tail factoring) on repeating fragments of the code (repeats). (The repeats are identified by building the so-called *suffix-trees*. See Fraser et al. [1984].) Procedural abstraction introduces a new procedure for the repeating fragment and other fragments call this new procedure instead of their "inline" execution. Cross-jumping (also known as tail-merging) can be applied to fragments with identical regions that end with a jump to the same target. These are merged together by replacing the region with a direct jump to another identical region.

Selecting the repeating fragments is based on calculating the expected saving or, alternatively, by applying profiling information as well (in which case the heavily executed portions are avoided in order to decrease the execution time overhead).

In order to be able to identify not only (lexically) equivalent fragments but also similar ones (which may differ in, e.g., just register names and operand literals), Cooper and McIntosh described several enhancements to the basic techniques by which the code fragments are transformed prior to the repetition identification. These include:

—abstracting branches by rewriting the labels to PC-relative form

—abstracting registers by first renaming them to relative references of a block and then applying the live-range recoloring of registers,

—constant abstraction,

—instruction ordering,

—etc.

Another proposal for the improvement of the algorithm is to use *profiling* information in terms of dynamic instruction counts for each function in order to choose those functions, for which it is economical to compact.

Note the resemblance of this method with the method by Liao et al. [1995, 1999], who also abstracted frequently executed instructions using minisubroutine calls.[6]

Brief classification

CPR	CPR-1	No need for decompression (compactor).
CHW	CHW-1,2	Any hardware.
CDE	CDE-0	Compactor.
CCE	CCE-2	Functional equivalence.
CCT	CCT-4	Computations on IR code (CCT-2).
CGR	CGR-7	The whole program at once.

Effects

ECS	0.95	Average value. The maximum was 0.85.
EEX	++	Code factoring involves additional instructions.
EES	−−	Code factoring involves additional instructions.
ECT	M	Relatively complex compiler techniques are used.
EDT	—	No need for decompression.
EBE	H	Only the external behavior is equivalent.
EEN	?	Probably increased due to increased EEX.
$\sum$		The decrease in ECS results in the degradation of EEX and EES.

[6]Liao et al [1995, 1999] developed a dictionary-based method for reducing the code size for DSP processors. The common instruction sequences that exceed some minimal length are put into a dictionary and each of the original occurrences is replaced with a mini-subroutine call to the (abstracted) procedure. The (compressed) code is therefore a skeleton of these procedure calls and other infrequent code sequences. The original algorithm can be applied without hardware modification but, using a minor modification in the processor hardware, greater compression ratios can be achieved.

4.2. Squeeze

Authors	R. Muth, S. K. Debray et al.
Affiliations	University of Arizona, Compaq Computer Corp.
Year	1998–2000
References	[Muth et al. 1998] [Muth 1999] [Debray et al. 2000]

The code compaction method outlined in this section has been implemented in a tool called *Squeeze*. Squeeze is part of `alto` [Muth et al. 1998], a post-link-time code optimizer developed at the University of Arizona.

The results of the whole project are comprehensively described in the PhD thesis of Robert Muth [Muth 1999]. The Squeeze tool itself is a binary-rewriting tool, which produces compacted directly executable code from a linked executable (DEC Alpha platform), already optimized by the compiler. The most comprehensive description of the Squeeze approach can be found in Debray et al. [2000].

The authors report savings as much as 30%, so this method can be regarded as practically the state-of-the-art in code compaction. Moreover, instead of running slower, the resulting code runs faster than the code produced by general code compaction techniques (e.g., Method 4.1). This is mostly due to the removal of code containing useless computations.

The main idea behind it is based on the work by Cooper and McIntosh (Method 4.1), but with several improvements. The most important one is probably that Squeeze operates on the *linked* code, which enables many opportunities for optimization, which the compiler could not exploit, as it generally works only at the level of compilation units and not at the level of the whole linked program.

Another important thing is that the Squeeze approach basically combines the classical compiler optimization concepts with general code compaction methods (such as enhanced code factoring, including procedural abstractions and others). One of the reasons why high compression ratios can be achieved with this approach is that the techniques used have been combined in a clever way by taking advantage of the merits of each.

The basis for all of the techniques is a good control flow graph (CFG), which must be computed from the linked executable code. The building of a precise CFG involves several complex techniques such as interprocedural constant propagation and register liveness analysis [Aho et al. 1985]. The performance of the optimization greatly depends on the precision of this graph.

Classical compiler optimizations. The classical optimizations for reducing the code size include redundant-code elimination, unreachable-code elimination, dead-code elimination and strength reduction. Most of these algorithms are based (although adapted to machine code) on the foundations of compiler design of the classical compiler book [Aho et al. 1985].

Code factoring. The code factoring is performed over the classical optimizations in terms of different kinds of code transformations in order to produce functionally equivalent, but smaller code. Code factoring implemented in Squeeze includes local factoring, procedural abstractions and various architecture-specific techniques.

Brief classification

CPR	CPR-1	No need for decompression (compactor).
CHW	CHW-1,2	Any hardware.
CDE	CDE-0	Adaptation of some architecture-specific techniques.
CCE	CCE-2	Functional equivalence.
CCT	CCT-4	Machine (for DEC Alpha, but can be adapted).
CGR	CGR-7	The whole program at once.

Effects

ECS	0.7	Best result.
EEX	$--/++$	An overall reduction is reported for most of the tests, however, the code factoring alone can increase the EEX.
EES	$++/--$	The same applies as for the EEX, even speedup is possible.
ECT	H	Compression involves complex compiler-techniques.
EDT	—	No need for decompression.
EBE	H	External behavior is equivalent, but there could be

		problems with for example, timing and real-time issues.
EEN	?	May be decreased due to the reduced EEX.
$\sum$		Smaller code and fewer executed instructions can positively influence the energy need. EBE may be problematic because the compacted code has been greatly altered.

4.3. Narrow-Word Instructions for Energy Reduction from Italy

Authors	L. Benini, A. Macii, E. Macii, and M. Poncino
Affiliations	Università di Bologna, Politecnico di Torino, Italy
Year	1999
References	[Benini et al. 1999]

In their paper Benini et al. [1999] describe a method for compressing code in order to decrease the energy requirements of code memory accesses (firmware code) in embedded systems. This is achieved by reducing the energy consumption originating from accessing the (external) code memory chip (ROM). The reduction is carried out by reducing the width of the instruction bus to a narrow word width, through which the most commonly executed instructions will be transferred in a compressed form.

They propose modifications to the architecture of CPU-to-memory accesses in order to achieve this compression. However, the technique is transparent to the processor core, so no modifications are needed to the CPU itself. (Note that this can be contrasted to some usual solutions by RISC MCU and CPU manufacturers to reduce energy costs in terms of instruction memory bandwidth minimization, e.g., ARM Thumb and MIPS16, where the instructions themselves are shortened.)

The transparency of the CPU is achieved by placing a specific hardware module between the program memory and the CPU (placed into an ASIC with the CPU), which uses the same communication protocol. This module is responsible for decoding the compressed instructions and to produce the original instructions for the CPU. This decoding is done on-the-fly during instruction fetch of the processor.

The idea is based on a previous paper by Yoshida et al. [1997]. The approach is the following. The most commonly executed instructions are compressed and coded with a narrow word (8 bits, in the implementation of Benini et al. [1999]). This means that only 255 instructions are encoded, while the rest is left unchanged (one specific code is reserved as an escape code). The codes assigned are *minimum Hamming distance binary codes*. The top instructions are selected based on simple statistical information, which counts the relative execution frequencies of each instruction. This is done by profiling, that is, executing the program on relevant inputs and counting the dynamic information. The coding uses $\lceil \log -2N \rceil$ bits, where N is the number of distinct instructions appearing in the code. This implies that the external code-memory bus is reduced to only 8 bits, by which the energy saving is achieved.

Based on the above, the storage in the memory consists of storing the uncompressed instructions—preceded by an escape code (256)—and the compressed instructions continuously.

The decompression is done by a hardware module between the processor core and the external memory that utilizes an IDT (Instruction Decompression Table), which stores the actual instructions corresponding to the 8-bit codes.

The most significant improvement of Benini et al. over the previous approach is that the size of the IDT is fixed and limited due to the fact that not all instructions are encoded, but only a limited number (255). This also implies a fixed compression ratio for the most frequently used instructions of $8/k$, where k is the width of the uncompressed instructions. However, the final compression ratio can be measured by incorporating the ratio of the number of compressed and uncompressed instructions. This ratio is not directly provided by the paper, however, it can be easily derived from the method itself by the above speculation. Concrete values are not presented, only the ratio of the actually executed

most commonly used instructions (static frequency ratios are not given).

Four different hardware architectural schemes are proposed of varying complexity for the implementation of the decompressor logic. They differ in the way memory is organized and accessed. The authors also present detailed simulation data for the estimated energy requirements of the four architectures. They conclude that the choice among these architectures can be made based on the available memory organization, the affordable complexity, which must be traded off against the achievable energy reduction.

The proposed approach is good for handling random access control transfers as well, since the decompression logic incorporates an address generation logic.

The final result of their experiments is that the dynamic memory utilization (this is most important for energy consumption) and memory switching is cut by about 50%, which suggests that significant memory reductions could be achieved in real implementations. This must include, however, the overhead of the decompression logic. However, the simulations suggest that its overhead is not significant with respect to the achievable savings coming from the method (an estimate of slightly more than 8-kbit memory space overhead is presented, which is mainly due to the size of the IDT). They argue that program execution time may also become shorter, although this was not demonstrated. The authors performed their experiments on the DLX processor architecture, but they claim that the method can be adapted to other processor architectures as well (primarily RISC). On the other hand, they present several assumptions regarding the architecture which may be an obstacle to the method's application.

Brief classification

CPR	CPR-2	Decompression by hardware.
CHW	CHW-1,2	Primarily RISC.
CDE	CDE-3	Several modifications.
CCE	CCE-0	The code is equivalent.
CCT	CCT-4	Machine code.
CGR	CGR-7	The whole program at once.

Effects

ECS	−−	Decreased, but concrete numbers are not given.
EEX	−−	Decrease is reported.
EES	∅	No effect.
ECT	L	Simple, but profiling is needed.
EDT	∅	Constant EDT time, but hardware complexity may differ.
EBE	∅	No effect.
EEN	−50%	Based on simulating memory utilization.
$\sum$		More complex decompression may increase overall energy need.

4.4. Code Compression for Cache-Based RISC by Wolfe et al.

Authors	A. Wolfe, A. Chanin, M. Kozuch, M. Beneš and S. M. Nowick
Affiliations	Princeton Univ., U. C. Berkeley, Motorola, S3 Inc. and others
Year	1992–1998
References	[Wolfe and Chanin 1992] [Kozuch and Wolfe 1994] [Beneš et al. 1997, 1998]

Wolfe and Chanin presented their first paper on code compression in 1992. Since then, the basic method has been improved in several other publications, but the idea remained the same: compressing code off-line using a good compressor and executing the code on a cache-based architecture by decompressing the code into the cache-line by a hardware module. They propose this approach for RISC processors.

Wolfe et al. call this method CCRP (Compressed Code RISC Processor). The method is transparent to the processor core, the only required hardware modifications are concerning the cache refill engine. The basic idea behind the method is that blocks of the program code (cache-lines) are encoded off-line using a good statistical compressor (whose decompressor part is relatively simple) and stored in the code memory. The instructions are decompressed via the (modified) cache refill engine when an instruction of the corresponding cache-line is referenced and the fetch is initiated during runtime. Hence, the decoded instructions are placed into the instruction cache in their original form. The decompression is triggered by a cache-miss, so relatively rarely.

Since the instruction addresses in the compressed code are invalid in the code memory (when cache-miss occurs), an additional table, called the Line Address Table (LAT), is used to map the addresses. The LAT is created during the compression phase and is stored along with the code in the code memory. The translation using the LAT is necessary only if the corresponding instruction is not in the cache. Searching the LAT is enhanced using a so-called Cache Line-address Lookaside Buffer (CLB), which improves the performance of the cache refill engine.

In their experiments, Wolfe et al. used different compression schemes. In theory, any good compressor could be used, but an important requirement is that it must be capable of block-based compression, since the cache-lines are compressed and decompressed individually rather than the whole program in one decompression phase. The encoders they investigated include traditional Huffman and bounded Huffman coders. Moreover, the scheme employed must offer the possibility of a reasonable hardware implementation. The decoding has a certain overhead in the execution time, but this greatly depends on the implementation hardware.

The authors gave an outline of the implementation of the decompressing engine, including the LAT and the CLB. They also presented some experimental results concerning the performance of the approach. They used a simulator to simulate the behavior of the cache logic. The results suggest that the runtime- and complexity-overheads induced by the decompressor logic is highly traded off by the compression ratios that can be achieved and the reduced memory-to-instruction cache traffic, which can also strongly affect the overall energy requirements in a beneficial way.

In the subsequent articles, Wolfe et al. suggested several improvements to the basic approach, as well as further experimental results regarding the size of the LAT and use of various compression schemes. In Kozuch and Wolfe [1994], the authors elaborated on the complexity and theoretical (zeroth-order, first-order and aggregate entropies), as well as practical (Huffman, `gzip`, variable symbol length) compression ratios. Their final conclusion was that an asynchronous Huffman coder is suitable for most applications [Beneš et al. 1998].

The group of authors also discussed concrete hardware implementations for their compression method, and describe the Huffman decoder in noticeable detail [Beneš et al. 1998, 1997].

Notice the resemblance of this method to the procedure-based compression method by Kirovski et al. [1997], which is briefly described in Section 4.7. The real difference lies in the granularity of the compression (cache-block vs. procedure).

Brief classification

CPR	CPR-2	Decompression by hardware.
CHW	CHW-1,2	RISC architectures with cache.
CDE	CDE-4	Using LAT and CLB.
CCE	CCE-0	The code is equivalent.
CCT	CCT-4	Machine code.
CGR	CGR-4.1	The blocks are the cache-lines. The experiments were performed with a cache-line of 32 bytes.

Effects

ECS	0.73	For Huffman, other compression schemes may differ.
EEX	$\emptyset$	No noticeable effect observed.
EES	$--$	Only a slight decrease was reported.
ECT	M	Depends on the compression scheme.
EDT	M	Depends on the compression scheme, mustn't be too complex.
EBE	M	EES could cause problems in time critical applications.
EEN	$--$	No data given.
$\sum$		More effective compressor schemes achieve higher ratios, but are more complicated to implement, so the EES and EEN may be degraded.

4.5. Improved Compression for Cache-Based RISC by Motorola

Authors	M. Breternitz Jr. and R. Smith
Affiliations	Motorola, Inc.
Year	1997
References	[Breternitz and Smith 1997]

Breternitz and Smith [1997] investigated the approach by Wolfe (Method 4.4)

and proposed a method, which eliminates most of the problems of the former. Using a clever technique, the necessity for the LAT and CLB is totally eliminated, and this way the implementation of the hardware decompressor is significantly simplified (see 4.4).

The need for the LAT is eliminated by retaining the mapping between runtime addresses (as seen by the CPU) and the addresses in the compressed code. In other words, when an instruction is decoded with a certain address reference, the address is simply mapped to the correct location in the compressed code.

The compression is also done by compressing blocks of code (i.e., cache-lines) and storing these compressed blocks. The size of the uncompressed blocks is fixed, and hence their base address is known. When the blocks are being compressed, the beginning address of each compressed block will, of course, depend on the compression ratio for each individual block.

When a compressed block is being decompressed and put into the cache (note, that this approach is the same as in Wolfe's method), the instructions refer to addresses, which have been changed to a special value in the compression phase. These address values contain the necessary information about the beginnings of the compressed blocks that need to be accessed next and put into the cache. In other words, the input program is preprocessed and compressed in a single step.

The original addresses are modified in the compression phase to a *base.offset* form, where *base* is a base address where the referred compressed block will take place, and the *offset* is an offset value in the newly decoded block. However, to compute the new base addresses, an estimate of the size of the compressed block is needed, which in turn depends on the modified address. So, in this sense, this is a chicken-and-egg problem, however the authors proposed an iterative solution, by which the final compressed blocks can be created.

Decompression is straightforward (and simple to implement), since all of the work regarding the mapping of the uncompressed and compressed addresses is done during compression. The decompressor only needs to properly interpret the special form of the addresses and to control the cache-refill mechanism.

The only problem with the method is the problem of fall-through blocks, that is, the subsequent execution of instructions that cross block boundaries but which do not contain direct jumps. The authors present several solutions to this problem. One is to insert an additional unconditional jump instruction at the end of each block, which transfers to the beginning of the next. This solution, however, may introduce some serious problems for applications with strict timing issues and also the code size increase may not be beneficial. Another solution is to generate the jump instruction on-the-fly by the cache refill unit of the hardware. The third solution is to detect the fall-through in the assembly of the program or by the compiler and to correct the labels, so that they do not land on cache line boundaries. In these solutions, an additional work needs to be done by the decompressor, namely, to keep track of the fall-through situations and to make necessary actions.

Notice that the problem of fall-through blocks may be a serious problem for some architectures. Another problem is that the algorithm needs to know all addresses during the compression. However, there are several practical situations where this is not possible: most of the processors have some instructions that use relative addressing for control transfers, that is, the address of the target is computed *at run-time* based on some computed values. This may be a result of the translation of some high-level source code structures, such as switch statements that are translated into jump tables. Although the authors claim that these are easily recognizable by the compressor and that it can be transformed to absolute targets, in some situations for some architectures, it is simply not possible to transform relative targets to absolute values. In these cases, however, the method may not be applicable in all situations.

Although the authors present some experimental data, the article leaves open many important questions regarding the implementation of the method (e.g., the fall-through problem and relative address problem). And this may be a serious hindrance to its applicability on arbitrary architectures.

Breternitz and Smith made their experiments on PowerPC firmware code. They compared the applications of different cache block sizes, and they present a compression ratio of 0.56 as the best.

Brief classification

CPR	CPR-2	Decompression by hardware.
CHW	CHW-1,2	RISC architectures with cache.
CDE	CDE-4	Jumps with relative targets may cause problems.
CCE	CCE-1	The addresses are modified.
CCT	CCT-4	Machine code.
CGR	CGR-4.1	Cache blocks sizes of 8 and 16 instructions were tried.

Effects

ECS	0.56	Best case, insufficient data was given.
EEX	$\emptyset/++$	May increase if additional jumps are added.
EES	$\emptyset/--$	May slow down if additional jumps are added.
ECT	H	Depends on the compression scheme; additional preprocessing is needed (relatively complex).
EDT	M	Depends on the implementation.
EBE	M	EES may cause problems in time critical applications.
EEN	?	No data reported, but may decrease due to ECS.
$\sum$		The modification of the program because of the fall-through and relative jumps may not be beneficial for some applications (see EES, EBE).

4.6. Improved LAT-Based Compression at Princeton

Authors	H. Lekatsas, W. Wolf and J. Henkel
Affiliations	Princeton University and NEC USA, Princeton
Year	1998–2000
References	[Lekatsas and Wolf, 1998, 1999a; 1999b]
	[Lekatsas et al. 2000a, 2000b, 2000c]

Lekatsas, Wolf and Henkel published a series of papers which all describe variations of the same approach, and were described in Lekatsas and Wolf [1998]. Their subsequent articles improve the basic method and provide additional results on other aspects like hardware issues and energy saving.

4.6.1. The Basic Methods. The approach of Lekatsas and Wolf [1998] is a significant improvement on the method first proposed by Wolfe et al. (see Method 4.4). They noticed that the applied Huffman compression of the LAT-based approach has several shortcomings, for example, all 4 bytes of a 32-bit RISC instruction are compressed as 8-bit symbols using the same table. Hence, they proposed that it would be much better to compress the various instruction fields separately (such as operands, immediates, etc.), rather than treat the program sequence as a single stream of symbols.

Another improvement was the application of arithmetic coding instead of Huffman because it produces better results and is easier to implement by hardware means. Otherwise, the approach is the same, including the decompression with the LAT and CLB.

As mentioned in Section 4.4, the code (actually, the cache blocks for it) is stored in the main memory in a compressed form. The decompression is triggered by a cache miss of the CPU. The code is then decompressed (by the hardware cache refill engine) and this decompressed code is stored in the cache memory for the CPU. The authors cogently remind us that the compression cannot be a traditional file-based scheme since the individual cache blocks must be accessed randomly.

Two methods are proposed, one which is independent of the instruction set and another which depends on the instruction set. The first method is called *SAMC* (Semiadaptive Markov Compression) and uses arithmetic coding. The second one is named *SADC* (Semiadaptive Dictionary Compression) and is dictionary-based. Since SAMC is more general and discussed more in subsequent publications, we will describe it in greater detail, while SADC will be described only briefly

(although it can actually compress more efficiently).

SAMC. This method uses a semiadaptive Markov model (semiadaptive means that the model is statically built for each subject program) to drive the arithmetic coder. The compression is performed as follows. Instructions are divided into k streams of bits, each containing k_i bits (note that these bits need not be adjacent). Our CGR-1 classification can be used here to describe the granularity as bit sequences. The stream division makes use of statistical profiling information to get the minimal joint entropy. The opcode division idea could be further improved by machine-learning techniques for determining (and predicting) the best opcode-division strategies. However, the authors conclude that the simple division into 4 streams with 8 bits each gives reasonable results without requiring excessive storage and complex decoding. A binary Markov tree is then generated for each stream with the appropriate probabilities. This is why the division is required. The reader may recall that the storage requirement of a k-bit long stream is $(2^{k+1} - 2)/2$! For the compression, a classic arithmetic coding is used based on the Markov tree for the streams, which are coded independently without taking into account any correlation between the streams themselves. The decompression is performed in the way described by Wolfe et al., but it is simplified due to arithmetic coding (for hardware implementation).

SADC. The second method differs significantly from SAMC in the way the instructions are divided into streams. This is done in a predetermined way taking into account the particulars of the instruction set for which it is applied. This means that a dictionary has to be created which stores opcodes or opcode combinations. The dictionary generator uses a greedy method to decide which values to put into the dictionary based on the estimated gain in savings using entropy information. The streams are finally encoded using Huffman.

Lekatsas and Wolf give some experimental results in their paper for both methods. They described their method with machine code of typical RISC (MIPS) and CISC (x86–Pentium Pro) architectures and compare the results measured on the SPEC95 benchmark programs. Their results show that SADC can produce significantly better results since it uses specialized information. However, it is much harder to implement in hardware and is less general. They also report that MIPS can be better compressed than x86. They finally conclude that the results are slightly worse compared to those using `gzip`, which is a file-based technique that can take advantage of finding long repeating sequences. The average compression ratios are about 0.5 for MIPS and 0.7 for x86. Overall, SADC performs about 5–10% better than SAMC.

4.6.2. Improvements. In Lekatsas and Wolf [1999a], the authors went in more detail their Markov model-based method (SAMC). In their paper, they focused on the coding scheme (arithmetic), especially the decoding part. They presented an arithmetic coding with reduced precision, which uses look-up tables instead of costly arithmetic operations. An implementation of the decoder and its Markov model is also briefly discussed there in terms of state-machines.

The SAMC method is described again with more detail in article [Lekatsas and Wolf 1999b]. The most important contribution of this paper is, however, that the authors provide experimental results on two architectures, Analog Devices Sharc and ARM's ARM and Thumb instruction sets. They also compare the results to the implicit size-reduction that can be gained by applying the reduced instruction set of ARM Thumb over its standard ARM instruction set (they present some interesting data according to which the Thumb code is already smaller than ARM by about 30%). It is also interesting to see that the already compact Thumb code is significantly harder to compress (0.6 for ARM vs. 0.9 for Thumb). The SAMC generally performs fairly well, only 15%

worse than gzip, one of the best file-based compressors.

In addition to compression performance, Lekatsas and Wolf give some results regarding the effect of the block sizes of the different architectures. They suggest that the bigger the block, the better the compression. The last experiment they made dealt with the decoding speed performance, where the conclusion is that on average about 2–7 bits can be decoded during one cycle, depending on the number of bits the encoder is looking ahead. Finally, the authors compare their method to similar related results like IBM's CodePack (Section 4.7), Lekatsas and Wolf's SADC, and Wolfe and Chanin's original method (Section 4.4). The overall result is that the SAMC method is better than all the others, except SADC, which is clearly the best (see above). In this evaluation, the authors also comment on the speed and storage of the different methods.

The overall conclusion is that using the SAMC method programs can often be reduced by more than 50%.

In Lekatsas et al. [2000a], the SAMC method is explained in terms of energy saving in embedded systems. They exploit an approximate arithmetic coding and a fast table-based decoding relying on Lekatsas and Wolf [1999a]. This coding scheme minimizes the number of bit transitions in external buses, thus reducing power consumption. Lekatsas and Wolf even provide some formulas for estimating the energy costs of the system based on bus capacitances and voltage differences. They present results of the analysis of instruction traces of program executions by counting the bus-related toggle count. If we apply this kind of estimate to the energy requirements, the requirements can be reduced by as much as 35%. They also conclude that the precision of the applied arithmetic coding has a negative effect on bit-toggling, which can be important because the energy consumption is estimated by the number of bit toggles in subsequent code values. This implies that there is a trade-off between huge compression and lower possibility for low energy and vice-versa.

4.6.3. The New Post-Cache Approach for Low-Power. Lekatsas and Wolf in cooperation with Jörg Henkel outlined a novel approach to system energy saving. In Lekatsas et al. [2000c], they argue that code compression is an efficient method for reducing power on embedded systems with complete SOCs (SOC means System-On-a-Chip, which consists of a CPU, separate instruction- and data-caches, a main memory along with data- and address-buses). The main contribution of the method is a new architecture for the instruction decompressor. Thus, they measure the energy savings in terms of the *complete system* rather than a single effect like compression ratio (they suggest that a high compression ratio does not necessarily produce the lowest energy consumption).

This new architecture employs the (same) decompression engine between the instruction cache and the CPU, rather than between the memory and cache—the usual approach from Wolfe et al. onwards (Method 4.4). The main advantage of this new *post-cache-architecture* is that both data-buses, before and after the cache, profit from the compressed instruction code, since the instructions are only decompressed before they are fed into the CPU. Moreover, they employ bus compaction by packing the compressed instructions into a single bit-stream utilizing the width of the bus (32 bits). However, they conclude that bus compaction should work best for small blocks. They carried out experiments with different cache sizes, and they concluded that the bit toggles increase with cache size, but this is saturated at some point (ca. 512 bytes).

The method described was simulated in their framework, where the energy consumption was estimated and also different implementation costs of the decompressor were measured by its synthesis (e.g., they reported additional power consumption of the decompressor engine of around 1 mW). The energy savings were measured in two ways: without and with the adjustment of the system to maximum energy saving with trade-offs to other performance gains. Without adjustments savings in

the range of 16–54% were measured, while with the adjustments they were 22–82%.

In Lekatsas et al. [2000b], the authors discuss further the post-cache architecture and present additional experimental results. They also present a *design methodology* that allows the hardware/software co-designer to control parameters of architectural trade-offs involving speed, power and chip-area. They concluded that the improvement/degradation of these parameters are mutually dependent. Different parameters are involved in the evaluation including cache size. The results presented suggest that increasing the cache size has an overall benefit on execution time and energy consumption.

Brief classification

CPR	CPR-2	Decompression by hardware.
CHW	CHW-1,2	RISC architectures (CISC has also been investigated).
CDE	CDE-4,5	Both approaches are presented in different articles.
CCE	CCE-0	The code is equivalent.
CCT	CCT-4	Machine code.
CGR	CGR-4.1	Different cache sizes, a typical one being 512 bytes. In terms of the stream division in SAMC and SADC, CGR-1 is also meaningful here.

Effects

ECS	0.5	Various ratios are given ranging from 0.5 for the MIPS processor to 0.9 for the ARM Thumb. The performance greatly depends on the method applied, target architecture and coding scheme.
EEX	$\emptyset$	No effect.
EES	$--$	May decrease because of the decompression.
ECT	M	Depends on the coding scheme.
EDT	M	Depends on the implementation.
EBE	M	EES may cause problems in time critical applications.
EEN	-35%	Using the post-cache approach even 82% can be gained.
$\sum$		The final results of the authors suggest that ENE depends on many factors not just on ECS.

4.7. Charles Lefurgy and IBM CodePack

Authors	C. Lefurgy, P. Bird, T. Mudge, T. M. Kemp et al.
Affiliations	University of Michigan, IBM
Year	1996–2000
References	[Bird and Mudge 1996] [Chen et al. 1998] [Lefurgy et al. 1997] [Lefurgy and Mudge 1998] [Kemp et al. 1998] [IBM 1998] [Lefurgy et al. 1999, 2000] [Lefurgy 2000]

Trevor Mudge, Charles Lefurgy et al. conducted research at The University of Michigan in the area of code compression for embedded systems. They published several papers on their results and proposed a dictionary-based compression method with hardware decompression. Their work is very similar to the LAT-based approach (CCRP) by Wolfe et al. outlined in Section 4.4. This similarity is because both use statistical compression and the decompression is done into the instruction cache and is triggered on cache-miss. This means that the granularity of the compression is a cache-line.

All of the methods are based on earlier work in technical reports [Bird and Mudge 1996; Chen et al. 1997] and the paper of Lefurgy et al. [1997]. IBM adopted the dictionary-based compression and implemented it for their PowerPC processor [Kemp et al. 1998; IBM 1998]. Although there are a number of differences between their implementation and the method described in the original articles, the basic principles are the same (but no clear connection can be found in the literature).

Further modifications, adaptations and improvements can be found in various publications. These include the experiments for DSP processors [Lefurgy and Mudge 1998] and software-managed decompression (see below). The whole methodology is summarized in Lefurgy's PhD thesis [2000], where a good survey of related methods can also be found.

With each of the methods described below, a system configuration with a good cache-miss ratio (optimal cache size and refill engine) is needed in order to minimize any slowdown due to decompression overheads.

4.7.1. The Baseline Dictionary Method. The original algorithm was first described in Lefurgy et al. [1997], which is in turn based on earlier work of Bird and Mudge [1996]; Chen et al. [1997] combining the results by Liao et al. (this method is outlined in Section 4.1) and Wolfe et al. (Section 4.4). The authors described a (statically) repeating pattern replacement method using a dictionary and investigated the behavior of the instruction cache (I-cache) in an experimental simulation environment. The motivation for the method is to be able to use smaller program memories (usually ROM), in which the program code is stored in a compressed way. This requirement is traded-off for performance loss that is needed for extra decompression activity during the execution.

The compressed code is created as follows: The machine code of the program generated by the compiler is analyzed to find the frequency of occurences of instruction sequences. The most frequent sequences (just one instruction or up to a whole basic block) get a codeword value that is stored in the program memory instead of the original sequence. The original sequence is mapped to the codeword value using the dictionary. The program, therefore, consists of intermixed uncompressed and compressed codeword values (this can be referred to as selective compression, since not all of the data is compressed). The codeword-values are of fixed length for fast decoding (8 or 16 bit) rather than using variable length codes as is done for Huffman codes, say.

The decompression is done on-the-fly by hardware means in the CPU logic during the execution of the program: if a compressed codeword is fetched, the decompressor logic is invoked, which produces the native instruction that is transferred to the execution pipeline. The uncompressed instructions are transferred directly.

In order to make it suitable for practical implementations, several constraints must be satisfied with this compression method. The number of dictionary entries and their sizes are limited to a practical value. Another restriction is that the branch instructions with offset-values (relative jumps) are not compressed as the branch targets are altered by the compression method. In other cases the modified control unit handles the branch targets.

Various experimental results are presented including the compression ratios (0.61 for PowerPC including the size of the dictionary) and the measurements about the inevitable overheads in terms of execution performance loss.

4.7.2. CodePack. IBM presented CodePack, their code compression method, which was implemented in their PowerPC processor in 1998 [IBM 1998]. The compression method applied is not unlike the work of the researchers at the University of Michigan.

There are, however, many differences to the (mainly experimental) algorithm by Lefurgy et al. The decompressor of CodePack is surely more complicated because IBM implemented several improvements to the technique. CodePack is designed primarily for small embedded applications, where the code memory size is an important factor. As is the case with Lefurgy's and the previous approaches by Wolfe et al., the CodePack compression is transparent to the processor core, that is, the decompression is achieved using a special control logic in the ASIC.

Lefurgy et al. published several articles dealing with the investigation of the CodePack compression method and its possible improvements [Lefurgy et al. 1999, 2000; Lefurgy 2000]. One interesting comparison is to the ARM Thumb and MIPS16 reduced instruction sets (viewed as compression). An obvious advantage of CodePack (and related methods) is that it does not need switching to and from the compressed/uncompressed instruction set, while having access to the full resources of the processor core (all registers, full immediate ranges, etc.). The achievable compression ratio is comparable to the architectures investigated and there is no need for new compilers since the native instruction set remains the same.

The decompression implemented in CodePack consists of a decompression unit on the ASIC chip between the core and the main memory. The core is unchanged since it sees the native instructions through the L1-cache, which is filled with the uncompressed native instructions (on cache-miss). The cache-miss address is mapped to the corresponding compressed address. The compressed instruction is fetched, decompressed and returned to the core. This is achieved using the dictionary and the index table.

In the concrete implementation, the 32-bit codewords are divided into two 16-bit sections that are encoded separately. Sixteen instructions are then compressed into a group of unaligned variable-length codewords. The compressed program, the dictionary and the index table are all located in the main (ROM) memory. However, for efficient operation the index-table is cached and the dictionaries are stored in a 2Kb on-chip buffer. With this efficient implementation the decompression is achieved at 1 instruction per cycle. There are several further implementation improvements, such as instruction prefetching and instruction forwarding. Besides this, Lefurgy et al. [1999] proposes further improvements for decreasing the cache-miss cycles.

The achieved compression ratios are comparable to those of the original methods by Lefurgy et al., which is roughly 0.6. At the same time not much performance decrease is reported; even speedup is possible because CodePack implements prefetching behavior, which the underlying processor does not have.

4.7.3. Software-Managed Decompression. A recent study of the authors of the method [Lefurgy et al. 2000] is the investigation of the possibility to replace the hardware-based decompression by software means. This would allow more independence on on various hardware and compression methods. On the other hand, serious modifications are required to the core and the instruction set. Yet the approach is very interesting and promising if added to some future processors.

The software decompression is similar to Kirovski's method[7] (with the significant difference in the granularity of the compression, i.e., block vs. procedure) and is based on CodePack. The method uses the instruction cache as a decompression buffer, into which the decompressed instructions are placed on a cache-miss. This whole procedure is performed by the processor itself execution a special cache-refill routine. To implement this, two modifications to the instruction set architecture are needed: exception raising on a cache-miss and an instruction to modify the contents of the cache.

The authors validated their approach on a simulator. Further experiments are presented for different selective compressions: based on cache-miss frequency and execution frequency (the frequent sequences are not encoded to reduce the decompression overheads). The achievable compression ratios based on simulations are similar (or slightly worse) than those for CodePack, but the execution slowdown is less significant due to the simpler decoding.

Brief classification

CPR	CPR-2	Decompression by hardware (or software).
CHW	CHW-1,2	Mainly for mainframe computers.
CDE	CDE-4	RISC architectures with different cache setups, PowerPC for CodePack; CDE-2 in the case of the software decompressor, because new instructions are involved.
CCE	CCE-0	CCE-1 in the case of using a software decompressor.
CCT	CCT-4	Machine code.
CGR	CGR-3,4.1	An instruction is the smallest, while a basic block is the largest unit that gets a codeword. However, the whole program is compressed.

[7]In Kirovski et al. [1997], the authors introduced a method for the compression of machine code at the granularity of the procedures (CGR-5) using an adaptive Ziv–Lempel model. Little or no hardware support is needed for the decompression. A procedure-cache is used to hold the decompressed procedures, which need to be large enough to hold the largest procedure. Frequently appearing code fragments are abstracted using mini-subroutine calls. A significant portion of their work deals with the handling of the defragmentation of this procedure-cache.

Effects

ECS	0.6	Includes the size of the dictionary and index table.
EEX	∅	In the case of the software decompressor EEX increases due to the decompressor code.
EES	−15%	Sometimes even speedup is possible in an optimized implementation.
ECT	L	A simple dictionary-based algorithm.
EDT	M	Many optimizations are possible in hardware decompression.
EBE	M	The unpredictible change of EES may be a problem in time critical applications.
EEN	−−	No data reported.
$\sum$		ECS and EES vary according to the cache-setup used. EEN is not reported but is probably improved due to smaller EES and decreased code-size.

4.8. Code Compression with Hardware Decompression from Brazil

Authors	G. Araújo, P. Centoducatte, M. Côrtes and R. Pannain
Affiliations	University of Campinas, PUC Campinas, Brazil
Year	1998–1999
References	[Araújo et al. 1998] [Centoducatte et al. 1999] [Araújo et al. 2000a, 2000b]

Araújo, Centoducatte et al. elaborated variants of their code compression techniques, whose basic characteristic is that the decompression is performed by a special hardware logic located in front of the CPU's instruction fetch unit. The decompression is done in real-time during program execution (i.e., the original instructions are generated during instruction fetch). A big drawback with their approach is that the decompression engine needs to be specific for the program it is decompressing.

In the original paper of Araújo et al. [1998], the authors propose a method for code compression that is based on *operand factorization*. Similar approaches have been proposed by others (see, e.g., Methods 4.12, 4.11). Their work differs in the decompression technique since it is designed to be applicable in *real-time* environments through a hardware logic. Another difference is that they compress expression trees, as produced by the compiler from the internal representation (based on Aho et al. [1985]). These trees contain the machine code patterns representing the expressions.

The basic idea is that the input program (more precisely, the expression trees with machine code instructions) is separated into two components: (1) the *tree-patterns*, which contain the opcode fragments of the expression (they do not contain branch instructions nor cross basic block boundaries) and (2) the *operand-patterns*, which contain the registers and immediates. The tree-patterns are identified using the algorithm in Aho et al. [1985]. They present some results, in which it is stated that the tree-patterns can occupy only a slight portion of all expression trees in the program (about 1%) and that the same is true for the operand-patterns as well. Hence, their separate encoding is justified.

These patterns are then composed into a sequence of codeword pairs, containing the encoded patterns. They experimented with several different (variable-length) encoding schemes, such as Huffman, bounded Huffman and VLC encoding [Haskell et al. 1997]. Different encodings require decompression engines of various complexity. Using the approach of Araújo et al., a silicon part gained by high compression ratio can be traded for an improved design of the decompression engine.

Probably the most important contribution of this method is the description of a novel decompression hardware logic that reassembles the original instructions from the encoded codewords. It must be used as a modification in the CPU instruction fetch mechanism. It has some limitations though, such as that branch instruction must be aligned to codeword boundaries and that the fixed base address width (21 bits) can necessitate an extra jump table in some cases. They claim that the decompression engine overheads are relatively small, on average 8%.

An overall size-reduction of 0.43 and 0.48 can be produced by the method for

different encoding techniques, and these ratios include an estimate of the decompression engine size (the best pure ratio is 0.35). The authors performed the experiments for a typical RISC processor.

In the article [Centoducatte et al. 1999], the authors adapted their previous method to DSP architectures (they applied it to a common DSP processor). The major difference with this method is that the expression trees are not decomposed, but they are encoded as a whole. The reason for this is that for the DSP processors the operand factorization has more redundancy (the DSP instructions are more compact).

Another feature of the author's new method is that it uses a simple encoder based on simple statistics about the frequencies of individual expression trees, rather than using Huffman or VLC.

Centoducatte et al. describe an outline of the decompression hardware as well in the second paper which translates the compressed codewords into the original instructions for the CPU. The compressed codewords are stored in a variable-length form. They even prepared simulations of the decoder logic with which estimated complexity overheads could be determined. They report that the average compression ratio was 0.28, but taking into account the size of the decompression engine the final ratio is on average 0.75.

The whole method is described in more detail and with additional experimental results in the articles [Araújo et al. 2000a, 2000b].

Brief classification

CPR	CPR-2	Decompression by hardware.
CHW	CHW-1,2	Primarily RISC; special application to DSP.
CDE	CDE-2	Special logic for the instruction fetch of the CPU.
CCE	CCE-0	Equivalent after decompression.
CCT	CCT-5	Machine code in the form of expression trees
CGR	CGR-7	The whole program at once.

Effects

ECS	0.43	0.75 for the DSP application. The data includes an estimate of the decompression engine size.
EEX	∅	No effect.
EES	∅	No effect.
ECT	M	Standard Huffman and VLC encodings.
EDT	+8%	Sophisticated decompression hardware has to implement the decompression (+47% for the DSP).
EBE	∅	No effect.
EEN	?	Complex decompression may not be good for energy, however if it is inside the same ASIC the reduced code-size may be beneficial because of the fewer accesses to external ROM.
$\sum$		The achievable compression ratio is reciprocally proportional to the complexity (and size) of the decompression engine.

4.9. Philips ThumbScrews

Authors	R. van de Wiel et al.
Affiliations	Philips Research
Year	2001
References	[Hoogerbrugge et al. 1999] [van de Wiel et al. 2001] [van de Wiel and Hoogendijk 2001]

Philips investigated various solutions for reducing the size of the embedded software in their electronic products. Unfortunately very little information is publicly available on the current research and results of their approach. In a Philips Research magazine, van de Wiel and Hoogendijk [2001] summarize their approach for software compaction in general (not just code-size reduction is discussed but other data compression techniques as well).

Regarding the code compression method utilized by Philips, the white paper [van de Wiel et al. 2001] provides an overview of the basic idea of the so-called "ThumbScrews" approach. ThumbScrews is a general solution for reducing the required memory size of an embedded program[8] for all kinds of standard microprocessors and controllers (although their primary implementation is for ARM Thumb code).

[8]The authors refer to the method as *code compaction*; however, using the terminology of this survey, we rather treat it as a *code compression* method.

The basic idea behind the ThumbScrews approach is to extend the compiler with a special target for code generation. The target architecture is a *virtual processor* with a special instruction set. A program compiled for this architecture is basically the compressed program that is stored in the program memory and is translated (i.e., decompressed) before the execution on the native processor.

This approach is basically a dictionary-based method for storing the program code in a compact way. A dictionary (memory table) is used to store the translation scheme (i.e., the way compressed instructions should be translated into native instructions). The compressed instruction set consists of (1) basic instructions and (2) macro instructions. The basic instructions are simply the encoded versions of basic instructions of the native processor. The macro instructions on the other hand encode *sequences* of native instructions that occur repeatedly in the code (i.e., they are entries to the dictionary which contains the sequences themselves). The instructions are encoded using variable length encoding (Huffman). Depending on the required performance vs. flexibility tradeoffs, the dictionary itself can be implemented via RAM, ROM, or hardware logic.

Once the compressed program is stored in the program memory, it must be decompressed (translated) to native instructions before the execution of the program. This task is done by a hardware decompressor module located between the program memory and the CPU. The decompressor translates each basic instruction into one native instruction and each macro instruction into a series of native instructions. The decompressor must keep track of the address of the next compressed instruction in the program memory (this issue is not elaborated in detail in the white paper, although the correct handling of runtime addresses is generally not simple, see, e.g., method 4.4).

The method is suitable for different hardware architectures also involving an instruction cache. In this case, the authors suggest to put the decompressor between the cache and CPU core in order to decrease the bus load by translating the compact code as late as possible. Of course, the cache line must be correctly handled (i.e., flushed and refilled) in the case of branch instructions. This introduces additional delays in execution.

There is not enough information provided on how the compression process and preparation of the dictionaries are carried out (in the article below [Hoogerbrugge et al. 1999] just the description is given of how the CFG-based internal representation is processed to find repeating expression trees). In general, ThumbScrews consist of a compilation and compression tool chain that produce the final program in the compressed instruction set along with the required dictionary (the decompressor and the tool chain contains the possibility for debugging as well). The tools analyze the source program and determine the best set of macro instructions for the program. Basically, the macro-instruction set is fixed for a subject program, but if the the table is to be implemented in ROM or hardware logic it can be specific for an application area (this way the software may be exchanged within this area without much degrading the compression capability). The tools support the production of mixed executables where the code may contain intermixed native and compressed instructions. Since the execution of compressed code involves some performance decrease (see below) this feature can be useful where high speed performance (or other critical issue) is needed by allowing it to be in native code. Of course, this must be supported by the execution engine as well, which simply means that if a non-compressed instruction is fetched the decompressor may be bypassed.

The reported compression ratios are quite typical for dictionary-based methods with variable length encoding. For instance, in an ARM environment, the ThumbScrews code is about 30% smaller than the same program compiled for the ARM Thumb instruction set. With other architectures with less compact instruction sets, this can be higher.

The decompression process before the execution adds some additional cycles to

the execution. This additional decompression activity results in an average decrease of 40% in execution time if the CPU is capable of fetching the next instruction in one cycle. The authors report, however, that if the processor has to wait more than one cycle the execution time converges to the uncompressed program execution.

Some of the basic techniques applied in ThumbScrews seem to be based on a previous work of the authors [Hoogerbrugge et al. 1999]. Here a compression method is described for a proprietary VLIW processor architecture. The compressed program is also in a form of a compact virtual instruction set. In contrast to ThumbScrews, this program is *interpreted* in software (classification CPR-3) using pipelining in the CPU so as to reduce the interpretation overheads. No special encoding is used either such as Huffman. One of the similarities with ThumbScrews is the fact that using this method the program is also partitioned on a per-function basis into critical and noncritical code to allow mixed executables comprising of compressed and noncompressed code. Furthermore, one of the main contributions of this article is the more detailed description of how the compressed code is produced.

Brief classification

CPR	CPR-2	Decompression by hardware (interpretation is also involved in some respect).
CHW	CHW-1,2	Mainly for but not limited to embedded systems.
CDE	CDE-5	Post-cache is proposed, although the method is not dependent on the intended architecture.
CCE	CCE-2	Since there is no such thing as the uncompressed code (the IR is used instead) the translated native code is only functionally equivalent (to the IR).
CCT	CCT-2,4	In some respects the internal representation is used for compression rather than in machine code form, but the final decompressed code is the machine code.
CGR	CGR-3	A native instruction is gained by translating one compressed basic instruction or macro.

Effects

ECS	0.7	Does not include the space needed for the dictionary.
EEX	$\emptyset$	No effect.
EES	-40%	This is due to decompression by hardware; may improve depending on the processor's fetching capability.
ECT	H	Compiler techniques are probably used to find and translate the macro instructions but the concrete complexity of dictionary creation is unknown.
EDT	M	The translation is relatively simple but the handling of addresses may be difficult.
EBE	L	The risk in timing-critical code (see EES) can be eliminated by leaving it in native form.
EEN	$--$	The decompression consumes extra energy but the overall effect may be beneficial because there are fewer external program memory accesses.
$\sum$		EBE mainly depends on EES and the selection of the noncompressable code. The dictionary may be program-specific (high ECS and low flexibility of the system) or application area-specific (lower ECS but higher flexibility of the system). The ECS also depends on how clever the compression algorithm is in finding the repeats (ECT).

4.10. Automatic Inference of Models by Fraser

Authors	Christopher W. Fraser
Affiliations	Microsoft Research
Year	1999
References	[Fraser 1999]

Fraser [1999] proposes a method for applying machine learning to code compression. He describes the basic approach for inferring the models for the compression and presents some experiments. However, other issues, such as encoding and application of the decompressor, are not the subject of the article.

There are several related approaches to code compression that utilize various artifacts in program code and their different (statistical) characteristics, primarily the opcodes and operands of machine language programs. These methods separate

the program into streams, trees or other representation, which are encoded separately. This separation is typically done by humans in a more ad hoc manner. See, for example, Methods 4.11, 4.12 and 4.8. On the other hand, in Fraser's method, this partition is done *automatically* by the means of machine learning technique that tries to produce the best possible streams for compression.

The method accepts a large amount of training data in the form of intermediate program representation and automatically infers a *decision tree*, which can be later used to separate the IR code into streams that compress much better than the undifferentiated whole. The decision tree produces statistical models that contain the necessary probabilistic distribution that can be used by arbitrary statistical coding methods (e.g., arithmetic). The nodes of the decision tree (the decision points) contain *predictors*, using which the tree can be later traversed and used. The predictors are, in a sense, binary functions that take some information based on the current input. The predictors (P) all have two children: one for which the predictor equals a value (V_P) and one for all others.

The basic idea behind the method is the observation that to encode the undifferentiated input using some statistical data (characterized by the entropy of the stream) is less effective than coding the separated streams using the local statistical data (with the local entropies). The learning of the model (i.e., the creation of the decision tree) is therefore based on *reducing the entropy* of the divided streams. This means in practice that the tree is decomposed into lower levels if the minimal sum of the differentiated entropies is less than that of the undifferentiated parent. More precisely, the minimal sum of entropies in each partition is computed for every predictor. If the minimal sum is less than the entropy of the whole sample (or subtree), do the partition recursively, otherwise use the undifferentiated distribution. This way, the leafs of the decision tree contain nearly optimal probabilistic distributions for the given input.

The tree itself is a kind of a classic decision tree with the difference that in this case it is binary in a sense that each node has two children: one for the positive answer of the actual predictor and one for the negative.

Once the tree is inferred (i.e., learnt) from the large amount of training data (which must be representative in order that it can be used for unknown inputs as well), the tree can be used to compute a nearly optimal model for the encoder from unknown inputs. (The whole procedure is necessary because the computation of the best entropy-based distributions and the learning of the tree is a very computationally demanding task, while the decision making based on the ready tree is very fast.)

The predictors are used to identify different context of the actual program point, based on which the decisions are made. These are typically used to investigate the type of the last tokens in the input (the so-called "Markov" predictors using the last 10–20 tokens) or some other computed values like the actual stack height.

Based on this, the usage of the tree is simple. The raw input (the intermediate representation taken from the compiler front-end) is separated into tokens like opcodes and operands and a table of predictor values is prepared. Then the tree is traversed for the input tokens (beginning from the root and simply going down to one of the leafs) using the actual predictor values to get the distributions present at the leafs, which are then used to encode the streams.

The article presents some impressive experimental results. Comparisons of this method are made with traditional (general purpose) compressors like `gzip` and code compressors like Method 4.11. Driving a typical arithmetic coder, this method produced a compression ratio of 0.19, which is significantly better than the other related methods (e.g., 30% better than Method 4.11 and 29% better than `gzip`). They also report that the achievable compression is 24% smaller than by using an output of an optimizing compiler with a good data-compressor.

An obvious disadvantage of this approach is that the "lost" information which is missing from the reduced-sized code is encoded in the decision tree. The decision tree, however, can be quite large in general. The authors propose some efficient techniques to aid this problem and reduce the size of the tree. Since the decoder must implement the decision tree in order to be able to reproduce the uncompressed input, the actual compression ratios should include this overhead. The given compression ratios for this method do not include this value but, as noted earlier, in many cases the size of the tree can be greatly reduced.

Another drawback can be that, without adaptation, it is not possible to use the method in situations where random jumps are present because the code needs to be decompressed completely before it can be used.

Brief classification

CPR	CPR-2	Only modeling is described.
CHW	CHW-0	An incomplete compression method.
CDE	CDE-0	An incomplete compression method.
CCE	CCE-0	The code is equivalent.
CCT	CCT-2	IR, but could be applied to CCT-4, too.
CGR	CGR-7	Could be adapted to others as well.

Effects

ECS	0.19	(20–30% better than a general compressor.)
EEX	Ø	
EES	?	No effect on the instructions, but decompression would be needed before execution.
ECT	L/H	Once the decision tree is ready ECT is not too complex, but the tree inference is hard.
EDT	M	Depends just on the applied coder and structure of the tree.
EBE	—	
EEN	?	The high compression ratio can be beneficial.
$\sum$		The achievable compression ratio is reciprocally proportional to the size of the decision tree (and therefore to the decompressor).

4.11. University of Arizona with Microsoft

Authors	C. W. Fraser, T. A. Proebsting, J. Ernst, W. Evans et al.
Affiliations	University of Arizona, Microsoft Research, AT&T Bell Labs.
Year	1995–1997
References	[Fraser and Proebsting 1995] [Ernst et al. 1997]

This section presents a set of related publications involving a group of authors whose most important results were published Ernst et al. [1997]. The common property of the methods is that a runtime system including an interpreter and decompressor is assumed. This is because the result of the decompression is not native machine code, but a low-level intermediate code (e.g., IR tree) that must be either interpreted or just-in-time (JIT) translated. The motivation for this method is that it is faster to send compressed code that is then interpreted or decompressed and executed at the client side. Two important bottlenecks are considered: transmission and memory. In the first case, a *wire code* compression is used, while in the second one the compression for *interpretable code*.

The approach is that the IR code is patternized in order to create separate streams of literal operands and repeating fragments of code instructions. These are then compressed in various suitable ways and transmitted. Using this method remarkable compression ratios as good as 0.2 can be attained, but the decompression may involve very sophisticated technologies such as client-side compilation and interpretation.

Ernst et al. [1997] present two techniques for compressing IR code (but it could possibly be applied to machine code as well). They present two bottlenecks as their motivation: transmission and memory, thus they suggest using the algorithms as a means of transferring and distributing executable code. This may be a serious hindrance to its use in certain class of applications, such as embedded systems.

The transmitted data is, in fact, a set of streams of compressed data using standard general compression methods such

as gzip and move-to-front (MTF) coding. Before compression and transmission, the input code is manipulated using various techniques.

In their paper, Ernst et al. present two basic techniques that can be applied in different scenarios of the above-mentioned motivations: *wire code* and *interpretable code*.

Wire code. This technique is used to gain high compression ratios by the application of different sophisticated compression methods. With this method separately compressed streams are produced, which are then compressed using suitable compressors. On the client side, the decompression consists of the decompression of the various streams and their translation (interpretation and/or JIT compilation) into the native code. The full compression algorithm involves several distinct steps, which we will not describe in detail here. However, the techniques involved in the preparation of the compressed streams include literal patternization (operand literals are placed into a separate stream), MTF coding, Huffman coding and gzip-ing.

Interpretable code. With this technique an encoded instruction-stream is produced, which is of a special form called BRISC (Byte-coded RISC), an interpretable virtual machine (VM) code. This is transferred to the client along with a dictionary containing the information needed to construct the native code. This one is probably more suitable for random access and code execution than the previous approach. The algorithm for preparing this special code includes sophisticated techniques for the manipulation of the IR code such as *operand specialization* and *opcode combination*. The operand specialization patternizes the operands of an instruction, thus producing several combinations of patterns for the same instruction. Opcode combination means finding instruction pairs (or N-tuplets) which are frequently used together.

The algorithm uses the techniques outlined above together to produce a heap of candidate instructions. This heap is iteratively updated using different combinations of operand specialization and making opcode pairs, triplets, N-tuplets, etc. The candidates are selected as described below and inserted into the dictionary (decompressing table). A greedy algorithm is used to select the best dictionary candidates, which will produce the smallest output data containing both the dictionary and the code stream. It uses the best K candidates based on a benefit value estimated for a concrete operand specialization and the opcode combination taking into account the program size reduction and the inevitable overheads introduced by the dictionary. Consequently, it does not always produce the best result.

The decoding means on-the-fly interpretation of the BRISC code or JIT compilation by the client.

Brief classification

CPR	CPR-3	Interpreter (and/or JIT) is required.
CHW	CHW-2	Mainframe computers.
CDE	CDE-1	Complicated decompression mechanism.
CCE	CCE-2	Functional equivalence (the input is in another form).
CCT	CCT-2	IR.
CGR	CGR-7	The whole program at once.

Effects

ECS	0.2	Best case for the wire code, BRISC performs around 0.6.
EEX	—	The input is in another form.
EES	−−	Compiler/interpreter on the client.
ECT	H	Many different techniques are involved.
EDT	H	Compiler/interpreter on the client.
EBE	—	Depends on the compiler.
EEN	?	Complex decompression may not be good for energy.
$\sum$		The remarkably high compression ratios have a trade-off in complex compression/decompression algorithms.

4.12. Slim Binaries

Authors	M. Franz and T. Kistler
Affiliations	University of California at Irvine, ETH Zürich
Year	1993–1997
References	[Franz 1994, 1997] [Franz and Kistler 1997]

"Slim Binaries" is the result of a project done over several years at Franz's original university, ETH Zürich. The fruits of his labours were then published in his PhD dissertation [Franz 1994]. Since then the method has appeared in several publications (e.g., Franz [1997]) of which the *ACM Communications* article probably gives the best overview with some basic technical details [Franz and Kistler 1997].

The notion of Slim Binaries is—as the authors call it—an infrastructure incorporating different techniques in a unified environment. Its purpose is to achieve flexible software portability and efficient program execution while at the same time preserving cross-platform compatibility in the *binary form* of the programs.

Two basic observations motivated the method. The first one is that the input and output speed of computer memories and peripheral devices do not grow at the same rate as the raw computational power of microprocessors. This means that in recent years it is increasingly more efficient to recalculate some intermediate results than to off-load them to secondary storage and read them back later [Franz and Kistler 1997]. The second observation was that cross-platform software releases have recently evolved and that for software vendors, to provide their products in different binary formats, they needed to do this using the so-called *fat binaries*, where multiple versions of the same program within a single object file were presented.

In sharp contrast, slim binaries can provide efficient storage of the program in a *platform-independent intermediate representation* format, which is, in addition, in a compressed form. The execution of the program is performed by fast decoding and *simultaneous code-generation*.

This kind of representation of the software binaries means that the object file of the program contains the "slim binary" plus the decoding engines that will provide the on-the-fly code generation.

Franz et al. argue that their implementation significantly reduced the overall storage requirements of the binaries. They present an example where an application in a "fat binary" form with three platforms had the overall size of 8.8 Mbytes, while the slim binary form including the on-the-fly compiler contained only 2.7 Mbytes. Taken the target binaries independently they could achieve compression ratios of about 0.35 with respect to the slim binary.

The method has the apparent drawback that the execution speed of the program is decreased because of the code generation tasks. However, based on the initial observations, the extra computations are far more economic than the secondary storage costs.

Slim binaries are created using the intermediate representation of the compiled program. More precisely, the *abstract syntax tree* generated by the compiler front-end is stored in a compact way (this means different kinds of *generalizations* on the recurring subexpressions). This tree is encoded by applying a predictive compression scheme, based on adaptive methods like LZW [Welch 1984]. The tree is traversed and encoded into a stream of symbols from a continually evolving vocabulary. This vocabulary is updated using adaptation and prediction heuristics. It contains variations of common subexpressions in some generalized form (e.g., the expression `i + 1` can be encoded by *"i-plus-something"* and *"literal one"* and later when a similar expression is found with a different constant, it will only refer to the previous subexpression code with a different literal). The authors claim that encoding syntax trees has many advantages over compressing character streams. One is that recurring common subexpressions can be efficiently encoded this way.

The application of the slim binaries technique includes:

(1) to preserve the above-mentioned effective cross-platform compatibility of the binaries,

(2) dynamic linking of software modules,

(3) extensible systems such as active documents containing program code,

(4) use with intelligent agents.

The authors initially integrated their method into an Oberon operating environment on the Macintosh platform, the *MacOberon*. This environment uses applications written in the Oberon language (although they claim that the method can be easily adapted to other languages as well). The applications are stored in slim binaries and upon their execution, the on-the-fly code generators create the native Mac binary code.

Apart from the obvious delay in the execution (they report about double startup times for a typical application), another drawback of the method can be that the code generation may be too complex for certain platforms like RISC processors (and the more complex processors would also require more complex code generation strategies). The implemented code generation is therefore chosen to be fast but not so efficient comparing to the modern optimizing compilers. However, they propose a solution to this problem in terms of *background code-generation*, where a central thread continually optimizes all of the already executing code in the background (which can use aggressive, albeit slow, optimization techniques). This opens the possibility of optimizing the code according to the profiling data collected so far.

An obvious similarity of this approach is the one with the Java byte-code and its virtual machine technology. However, there are obvious advantages of the slim binaries over the other. Namely, the Java byte-codes are at a much lower semantic level (the slim binaries store syntax trees while the byte-codes are only low-level instructions) and therefore their compaction is not so efficient (the authors report that their binaries are twice as dense as Java byte-codes). Another advantage is that the code-generation of the syntax trees of slim binaries can be easier to implement (on the other hand, their byte-to-byte interpretation is not possible). Talking about Java, we should mention that Java byte-codes are compressed with CGR-6 granularity with respect to a Java class.

Brief classification

CPR	CPR-3	On-the-fly code generation is required.
CHW	CHW-2	Mainframe computers.
CDE	CDE-1	On-the-fly code generation into RAM.
CCE	CCE-2	Functional equivalence (the input is in another form).
CCT	CCT-2	IR (abstract syntax tree).
CGR	CGR-7	(Java byte-codes are CGR-6.)

Effects

ECS	0.35	Typical value.
EEX	—	The input is in a different form.
EES	÷2	Execution time doubles because code generation is required before execution.
ECT	H	Source code analysis and adaptive compression is involved.
EDT	H	Decompressor and (optimizing) code generator on the client.
EBE	—	Depends on the compiler.
EEN	?	Decreased code-size is good but code generation needs extra space and computation.
$\sum$		The better code is generated, the more complex decompression is involved, therefore EES is decreased.

5. EVALUATION OF THE METHODS

We evaluate the twelve methods of the survey using our assessment criteria defined in Section 3 and the results of the assessment in the previous section. For the tables and diagrams, we use the data described in the tables at the end of each section corresponding to the methods.

5.1. Summary of the Methods

In this section, we summarize our classifications given at the end of the discussion of each method. The table below contains the results on the classification assessment. The methods are listed according to their section number.

Method	CPR	CHW	CDE	CCE	CCT	CGR
4.1 Cooper & McIntosh	1	1,2	0	2	4	7
4.2 Squeeze	1	1,2	0	2	4	7
4.3 Narrow-Word	2	1,2	3	0	4	7
4.4 Wolfe (LAT-based)	2	1,2	4	0	4	4.1
4.5 Breternitz	2	1,2	4	1	4	4.1
4.6 Lekatsas & Wolf	2	1,2	4,5	0	4	4.1,1

4.7	CodePack	2	1,2	4	0	4	3,4.1
4.8	Brazil	2	1,2	2	0	5	7
4.9	ThumbScrews	2	1,2	5	2	2,4	3
4.10	Model inference	2	0	0	0	2	7
4.11	Arizona-Microsoft	3	2	1	2	2	7
4.12	Slim Binaries	3	2	1	2	2	7

The next table offers a quick reference of the methods according to their classification in Section 4 (the numbers correspond to the subsection numbers of Section 4). The table could be useful when a suitable method is sought based on requirements related to its application, which may be assessed according to our classification. The reader should bear in mind that the CHW classification is interpreted differently from the others and is described in Section 3.1.

CPR-1	compactors	1, 2
CPR-2	compressors	3, 4, 5, 6, 7, 8, 9, 10
CPR-3	interpreter-based	(9), 11, 12
CHW-0	n/a	10
CHW-1	SOC/embedded	1, 2, 3, 4, 5, 6, 7, 8, 9
CHW-2	mainframe	1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12
CDE-0	n/a, software-based	1, 2, 10
CDE-1	sw-based with memory	11, 12
CDE-2	modified CPU	8
CDE-3	without cache	3
CDE-4	pre-cache	4, 5, 6, 7
CDE-5	post-cache	6, 9
CCE-0	complete	3, 4, 6, 7, 8, 10
CCE-1	as seen by the CPU	5, (7)
CCE-2	only functional	1, 2, 9, 11, 12
CCT-0	non-code	
CCT-1	source code	
CCT-2	IR	(1), 9, 10, 11, 12
CCT-3	assembly	
CCT-4	machine	1, 2, 3, 4, 5, 6, 7, 8, 9
CCT-5	special	8
CGR-0	irrelevant	(10)
CGR-1	bit sequence	(6)
CGR-2	character	
CGR-3	instruction	7, 9
CGR-4.1	small block	4, 5, 6, 7
CGR-4.2	large block	
CGR-5	procedure	see Kirovski et al. [1997]
CGR-6	translation unit	
CGR-7	program	1, 2, 3, 8, 10, 11, 12

Another interesting issue regarding the compressor-type methods (CPR-2) is their choice of the applied modeler/coder combination (see Section 2 for background details). In some cases, it would be beneficial if the modeler or coder could be adapted to a certain application. The table below might help one choose the appropriate method by listing the applied coding scheme (compactors do not use any coding).

	Method	Coder
4.3	Narrow-Word	Dictionary
4.4	Wolfe (LAT-based)	Huffman
4.5	Breternitz	Huffman
4.6	Lekatsas & Wolf	Arithmetic
4.7	CodePack	Dictionary
4.8	Brazil	Huffman
4.9	ThumbScrews	Dictionary
4.10	Model inference	(Arithmetic)
4.11	Arizona-Microsoft	Dictionary
4.12	Slim Binaries	Dictionary

5.2. Summary of the Effects

In this section, we provide an overview of all of the effects looked at in the methods investigated. The table below summarizes the effects found for each method (the methods marked with a * have important remarks concerning their compression ratio, which are given below).

Method	ECS	EEX	EES	ECT	EDT	EBE	EEN
4.1 Cooper & McIntosh	0.95	++	−−	M	—	H	?
4.2 Squeeze	0.70	−−	++	H	—	H	?
4.3 Narrow-Word	−−	−−	∅	L	∅	∅	−50%
4.4 Wolfe (LAT-based)	0.73	∅	−−	M	M	M	−−
4.5 Breternitz	0.56	++	−−	H	M	M	?
4.6 Lekatsas & Wolf	0.50	∅	−−	M	M	M	−35%
4.7 CodePack*	0.60	∅	−15%	L	M	M	−−
4.8 Brazil*	0.43	∅	∅	M	+8%	∅	?
4.9 ThumbScrews	0.70	∅	−40%	H	M	L	−−
4.10 Model inference*	0.19	∅	?	L	M	—	?
4.11 Arizona-Microsoft*	0.20	—	−−	H	H	—	?
4.12 Slim Binaries	0.35	—	−50%	H	H	—	?

In the following, separate diagrams are provided for each effect presenting the detailed data, which provide a basis for comparison for each method.

Notice that many of the effects could not be determined completely for every method investigated. Another problem was that, in some cases, no numeric data was available. Hence, we employ some special notations in the diagrams to allow for

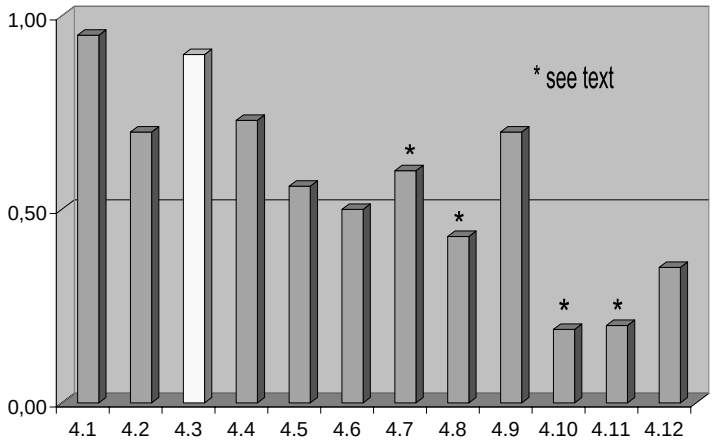

Fig. 1. Effect on the size of the compressed code (ECS).

comparisons between the various methods. In general, **the estimated values are denoted by light bars, while the darker bars correspond to concrete values**. For those effects where a decrease or an increase could be measured, we used the ratio *modified/original*, where 1 stands for no change. Values greater than 1 mean an increase, and smaller than 1 mean a decrease. At those places where no concrete values are given (light bars) and only an increase or decrease is listed (e.g., EEX), we assume an increase or decrease of 10% (which is a very imprecise estimate, but there is no other way of including these values in the comparison). For example, the ECS effect for Method 4.3 is assigned a value of 0.9. For results with three-value estimates denoted by L, M and H, we used 0 to indicate an unknown, no-impact or not-applicable value.

The most important effect is ECS, the effect on the size of the (compressed) code. In other words, this gives the achievable compression ratio. As mentioned earlier in Section 3.2 and elsewhere, the main difficulty with this ratio is that the methods do not always describe whether the measurements contain the size of the modeler and/or decoder. In fact, only methods 4.7 and 4.8 explicitly state that their results contain these overheads. Another example is method 4.10, where it is clear that the given (rather high – 81%) ratio does not include the size of the model, which can be quite high in general (see the corresponding section for details). The other method with the highest ratio is 4.11. The given numbers (80%) are best case and should be also viewed with caution because it is not clear whether this ratio can be achieved if all of the overheads are incorporated with all the variations of the method.

In Figure 1, we can see the compression ratios given by the methods. Note again that the value for Method 4.3 is inaccurate, it is only an estimate of 0.9, since the publication fails to give precise results on the compression ratio of this method.

We also prepared similar diagrams to demonstrate the other effects for the methods investigated, which are shown in Figures 5 to 10 in the Appendix.

In Figure 2, we can see a summary of three important effects, which often have mutual impact on each other: the compression ratio, the number of executed instructions and energy consumption. It may be concluded here that there is a correlation between EEX and EEN in the sense that more executed instructions require a higher energy consumption. Another observation is that where a significantly higher compression ratio is obtained (e.g., Method 4.11), the other two effects are degraded.

In a combined diagram shown in Figure 3, the compression ratio can be compared with the compression and decompression time/complexity measures.

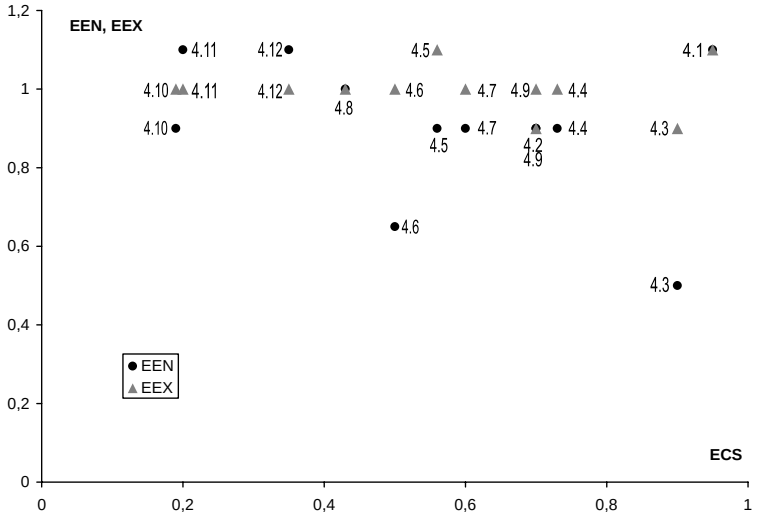

Fig. 2. Summary of effects for EEN and EEX. Many of the values are estimates (see text).

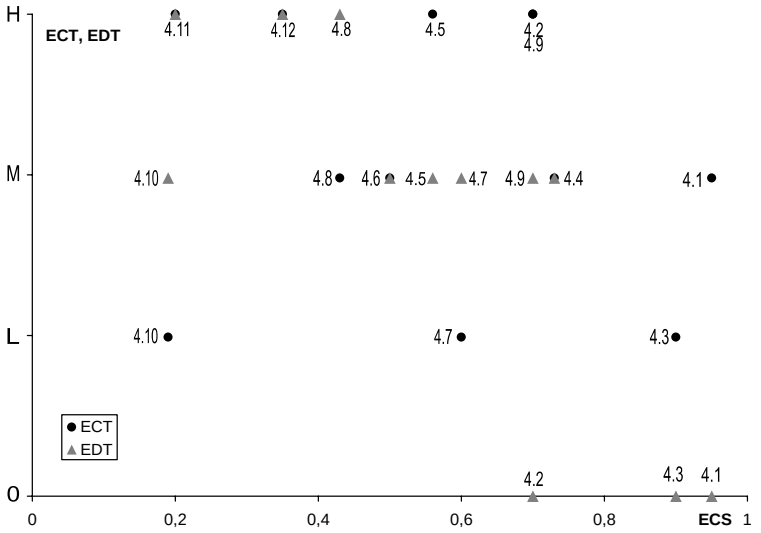

Fig. 3. Summary of effects for ECT and EDT. Many of the values are estimates (see text).

Not surprisingly, the simpler methods produce worse ratios (EDT in particular).

5.3. Effects on Each CPR

In this section, we summarize how the effects depend on the basic classification of the method, namely the CPR—size-reduction principle. For each effect, we computed the average values for each of the three CPR groups. It may be seen in Figure 4 that the interpreter-based methods (CPR-3) produce the best compression ratios, but they are also more involved as well. With these methods the effect on the execution speed (EES) is also significant, since the interpretation (compilation) of the compressed code also involves runtime overheads.

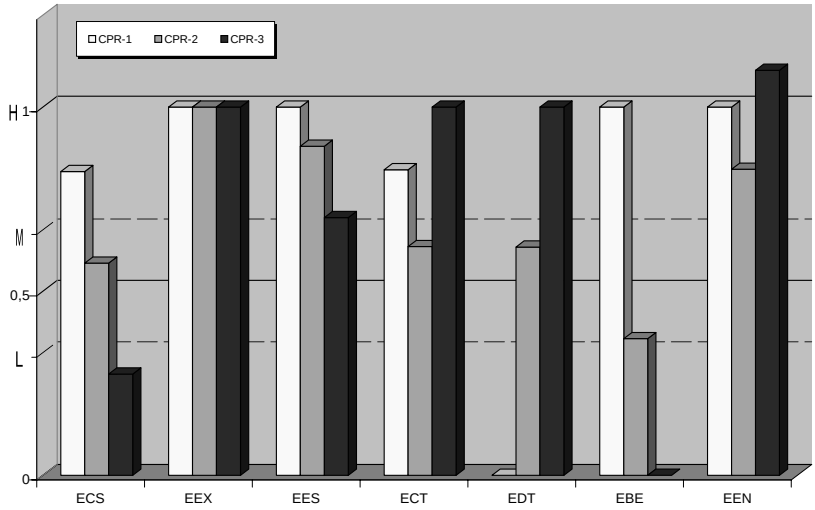

Fig. 4. Which CPR is best for a specific effect?

On the other hand, compressors (CPR-2) produce acceptable ratios as well, but they are simpler to implement. The only type of methods, which report concrete energy measurements is also this type.

The compactors (CPR-1) can produce only modest compression ratios, but in this case there is no need for decompression at all. Another observation is that since they produce modified code, the effect on behavior safety (EBE) is rather high.

5.4. Conclusion about the Energy Aspects

As pointed out earlier, the question of energy consumption is very important for some applications, mostly embedded systems. It is also interesting to note that this issue appeared to be related in various ways to code compression. Several methods were concieved simply because of this. These mostly belong to the CPR-2 group, where the energy saving is achieved by reducing the access to program memory (in many different ways, as can be seen from the method descriptions in Section 4).

Altogether, it is difficult to form reasonable conclusions about the most appropriate methods to use for energy saving. There are several reasons for this. Firstly, energy saving is not a primary aspect of many methods. However, those methods which had energy saving as their prime motivation in every case produced concrete results. For example, Method 4.3 (narrow word coding + IDT) halves the energy need by reducing the width of the bus connected to the external code memory.

It may be generally concluded that this issue mostly depends on the hardware architecture used for decompression and/or interpretation. This suggests that the compression ratio is not always the key factor in this issue. This is also clearly seen in the improved version of Method 4.6. Here, access to external memory is decreased, hence there may be a close correlation (they could be proportional) between energy saving and compressed code-size (provided that cache activity consumes less energy).

Other code-size reduction methods such as compactors (type CPR-1) are not primarily designed for energy saving. In fact, one may have a general feeling that code factoring methods (as part of some compactors) usually produce more instructions to be executed in runtime, therefore external memory is accessed more frequently. However, if caching is used, this might not be true. Moreover, some methods from the Squeeze approach (Section 4.2) decrease the cost of the executed instructions (time- or energy-cost). This includes strength reduction techniques, where a costly code

fragment is replaced by a cheaper one. And, of course, since smaller RAM chips are needed this will have a benefit regarding cost and energy.

Interpreter- and JIT compiler-based methods (type CPR-3) can rarely be applied to energy saving. However, one issue is worth considering here: if the RAM in the ASIC is large enough (to hold larger units for decompression, e.g., granularity CGR-5) and if accessing it consumes less energy than the external ROM, then direct in-RAM decompression and execution could be employed (recall that these methods can produce good compression ratios).

Our overall conclusion here is that more methods should be combined in order to achieve the best energy saving results. In particular, we should remember that energy saving does not just depend on code compression, but other hardware/software co-design and architectural issues as well. As was once pointed out, "energy saving is a system-wide exercise."

6. SUMMARY AND CONCLUSION

In this paper, we surveyed 12 methods for code-size reduction, published in some 50 articles from 1984 to date. They differ in many ways because they often have different motivations and application areas, ranging from network traffic minimization to energy saving in embedded systems. The basic size-reduction principle is also very diverse, like code compaction and hardware-based decompression and interpretation.

We propose two sets of broad assessment criteria which (1) classify the methods and (2) evaluate their effects. This type of survey methodology was chosen because the classification of the methods is difficult due to their diversity and because the effects can be independently monitored. The effects of the methods are grouped in various ways including the most obvious one, that for the compression ratio.

Evaluation results are presented where the different assessment criteria are jointly evaluated for all of the methods investigated. A direct comparison of the various aspects of the methods (assessment criteria) is difficult because, in many cases, the publications do not provide enough information about the methods and, when they do, the data is often not directly comparable because of the different environments used for the experiments.

We may conclude that there is no such thing as "the best method for code-size reduction" since all methods examined here perform well in different usage contexts. We should only seek to provide a good basis for selecting the best candidate or, at least, offer some guidelines on the scope of applicability and achievable effects vs. affordable trade-offs.

APPENDIX

In Figures 5 to 10, the effects are summarized for the methods that we investigated.

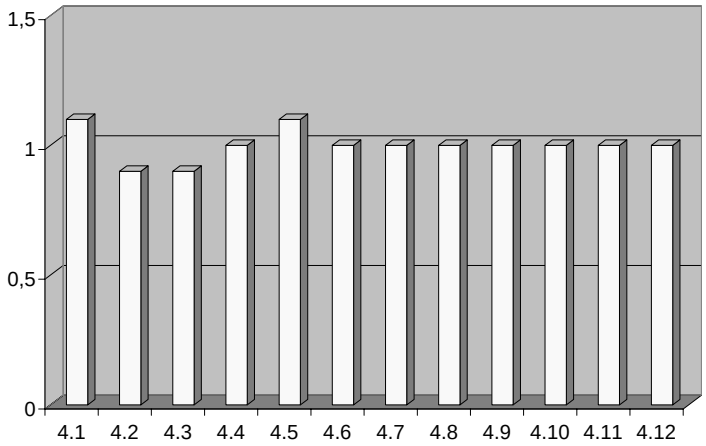

Fig. 5. Effect on the number of executed instructions (EEX).

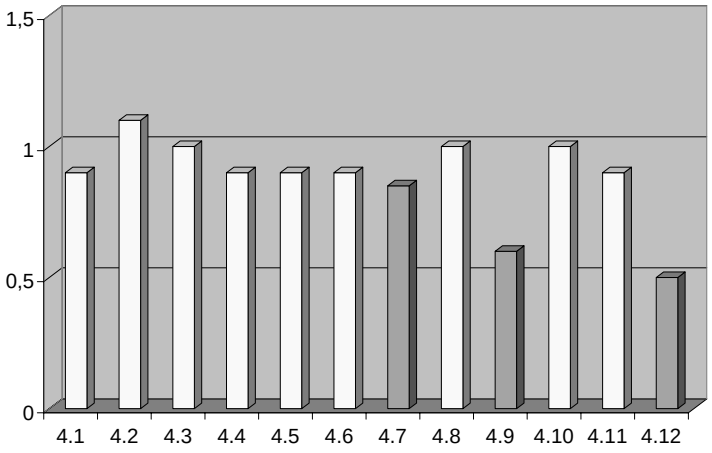

Fig. 6. Effect on the execution speed (EES).

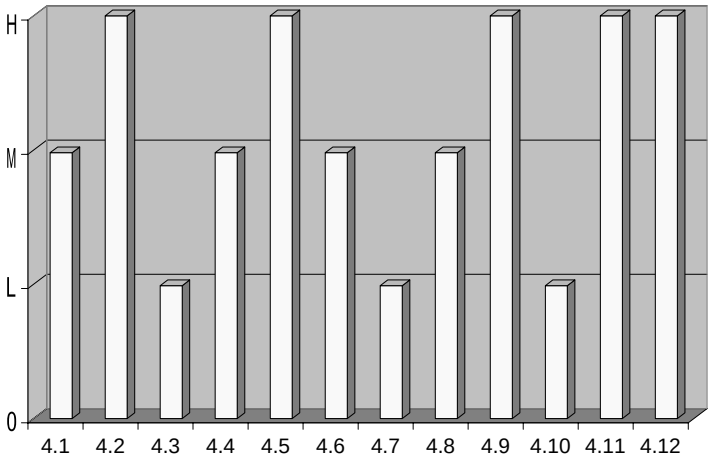

Fig. 7. Effect on the compression time/complexity (ECT).

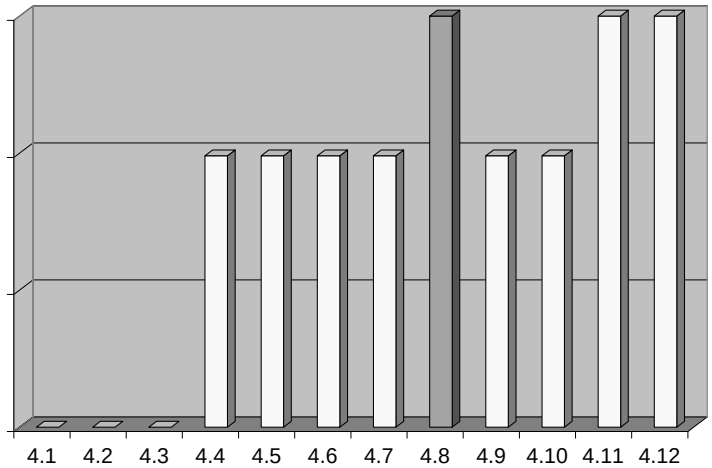

Fig. 8. Effect on the decompression time/complexity (EDT).

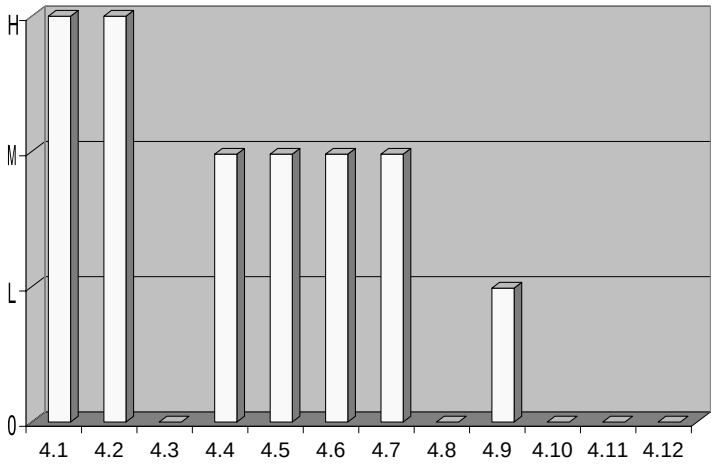

Fig. 9. Effect on the behavior safety (EBE).

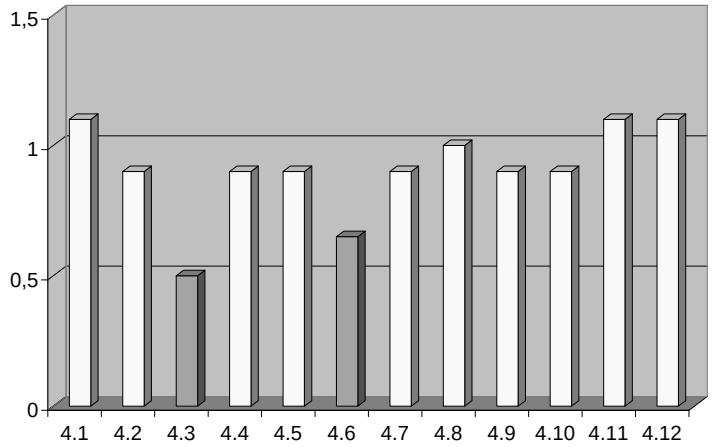

Fig. 10. Effect on the energy consumption (EEN).

REFERENCES

AHO, A. V., SETHI, R., AND ULLMAN, J. D. 1985. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass.

ARAÚJO, G., CENTODUCATTE, P., AZEVEDO, R., AND PANNAIN, R. 2000a. Expression tree based algorithms for code compression on embedded RISC architectures. Tech. Rep. IC-00-01, Instituto de Computação—UNICAMP. Jan.

ARAÚJO, G., CENTODUCATTE, P., AZEVEDO, R., AND PANNAIN, R. 2000b. Expression tree based algorithms for code compression on embedded RISC architectures. *IEEE Trans. VLSI Syst. 8*, 5 (Oct.), 530–533.

ARAÚJO, G., CENTODUCATTE, P., CÔRTES, M., AND PANNAIN, R. 1998. Code compression based on operand factorization. In *Proceedings of International Symposium on Microarchitecture (Micro-31)*. 194–201.

BELL, T. C., WITTEN, I. H., AND CLEARY, J. G. 1990. *Text Compression*. Advanced Reference Series. Prentice Hall, Englewood Cliffs, N.J.

BENEŠ, M., NOWICK, S. M., AND WOLFE, A. 1998. A fast asynchronous Huffman decoder for compressed-code embedded processors. In *Proceedings of the International Symposium on Advanced Research in Asynchronous Circuits and Systems*. IEEE Computer Society Press, Los Alamitos, Calif., 43–56.

BENEŠ, M., WOLFE, A., AND NOWICK, S. M. 1997. A high-speed asynchronous decompression circuit for embedded processors. In *Proceedings of the 17th Conference on Advanced Research in VLSI*.

IEEE Computer Society Press, Los Alamitos, Calif.

BENINI, L., MACII, A., MACII, E., AND PONCINO, M. 1999. Selective instruction compression for memory energy reduction in embedded systems. In *Proceedings of ISLPED '99*. ACM, New York, 206–211.

BENTLEY, J. L., SLEATOR, D. D., TARJAN, R. E., AND WEI, V. K. 1986. A locally adaptive data compression scheme. *Commun. ACM 29*, 4 (Apr.), 320–330.

BIRD, P. L. AND MUDGE, T. N. 1996. An instruction stream compression technique. Tech. Rep. CSE-TR-319-96, EECS Department, University of Michigan. Nov.

BRETERNITZ, M. J. AND SMITH, R. 1997. Enhanced compression techniques to simplify program decompression and execution. In *Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*. IEEE Computer Society Press, Los Alamitos, Calif., 170–176.

CENTODUCATTE, P., ARAUJO, G., AND PANNAIN, R. 1999. Compressed code execution on DSP architectures. In *Proceedings of the ACM/IEEE International Symposium on System Synthesis*. ACM, New York.

CHEN, I.-C. K., BIRD, P. L., AND MUDGE, T. N. 1997. The impact of instruction compression on I-cache performance. Tech. Rep. CSE-TR-330-97, EECS Department, University of Michigan.

COOPER, K. D. AND MCINTOSH, N. 1999. Enhanced code compression for embedded RISC processors. In *SIGPLAN Notices (PLDI '99)*, vol. 5. ACM, New York, 139–149.

DEBRAY, S. K., EVANS, W., MUTH, R., AND DE SUTTER, B. 2000. Compiler techniques for code compacion. *ACM Trans. Prog. Lang. Syst. 22*, 2, 378–415.

ERNST, J., EVANS, W., FRASER, C. W., LUCCO, S., AND PROEBSTING, T. A. 1997. Code compression. In *PLDI '97*. ACM, New York, 358–365.

FRANZ, M. 1994. Code-generation on-the-fly: A key to portable software. Ph.D. dissertation, ETH Zürich.

FRANZ, M. 1997. Adaptive compression of syntax trees and iterative dynamic code optimization: Two basic technologies for mobile-object systems. In *Mobile Object Systems: Towards the Programmable Internet*. Lecture Notes in Computer Science. Springer-Verlag, New York, 263–276.

FRANZ, M. AND KISTLER, T. 1997. Slim binaries. *Commun. ACM 40*, 12 (Dec.), 87–94.

FRASER, C. W. 1999. Automatic inference of models for statistical code compression. In *SIGPLAN Notices (PLDI '99)*, vol. 5. ACM, New York, 242–246.

FRASER, C. W., MYERS, E. W., AND WENDT, A. L. 1984. Analyzing and compressing assembly code. In *Proceedings of the ACM SIGPLAN'84 Symposium on Compiler Construction*, vol. 19. ACM, New York, 117–121.

FRASER, C. W. AND PROEBSTING, T. A. 1995. Custom instruction sets for code compression. Unpublished manuscript. `http://research.microsoft.com/~toddpro/papers/pldi2.ps`.

HANKERSON, D., HARRIS, G. A., AND JOHNSON, P. D. J. 1997. *Introduction to Information Theory and Data Compression*. CRC Press.

HASKELL, B. G., PURI, A., AND NETRAVALI, A. N. 1997. *Digital Video: an Introduction to MPEG–2*. Digital Multimedia Standards Series. Chapman & Hall, New York.

HOOGERBRUGGE, J., AUGUSTEIJN, L., TRUM, J., AND VAN DE WIEL, R. 1999. A code compression system based on pipelined interpreters. *Softw.–Pract. Exper. 29*, 11, 1005–1023.

HOWARD, P. G. AND VITTER, J. S. 1992. Practical implementations of arithmetic coding. *Image Text Compres.*, 85–112.

HUFFMAN, D. A. 1952. A method for the construction of minimum redundancy codes. *Proc. IERE 40*, 1098–1101.

IBM. 1998. *CodePack: PowerPC Code Compression Utility User's Manual Version 3.0*. International Business Machines (IBM) Corporation.

KEMP, T. M., MONTOYE, R. K., AUERBACH, D. J., HARPER, J. D., AND PALMER, J. D. 1998. A decompression core for PowerPC. *IBM J. Res. Develop. 42*, 6 (Nov.), 807–812.

KIROVSKI, D., KIN, J., AND MANGIONE-SMITH, W. H. 1997. Procedure based program compression. In *Proceedings of the International Symposium on Microarchitecture*. 204–213.

KOZUCH, M. AND WOLFE, A. 1994. Compression of embedded system programs. In *Proceedings of the International Conference on Computer Design: VLSI in Computers & Processors*. IEEE Computer Society Press, Los Alamitos, Calif.

LEFURGY, C. R. 2000. Efficient execution of compressed programs. Ph.D. dissertation. The University of Michigan.

LEFURGY, C. R., BIRD, P. L., CHEN, I.-C., AND MUDGE, T. N. 1997. Improving code density using compression techniques. In *Proceedings of the 30th International Symposium on Microarchitecture (Micro-30)*. IEEE Computer Society Press, Los Alamitos, Calif.

LEFURGY, C. R. AND MUDGE, T. N. 1998. Code compression for DSP. Technical Report CSE-TR-380-98, EECS Department, University of Michigan. Nov.

LEFURGY, C. R., PICCININNI, E., AND MUDGE, T. N. 1999. Evaluation of a high performance code compression method. In *Proceedings of Micro-32*. 93–102.

LEFURGY, C. R., PICCININNI, E., AND MUDGE, T. N. 2000. Reducing code size with run-time decompression. In *Proceedings of the 6th International Symposium on High-Performance Computer Architecture (HPCA)*.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000a. Arithmetic coding for low power embedded system design. In *Proceedings of the 2000 IEEE*

Data Compression Conference. IEEE Computer Society Press, Los Alamitos, Calif. 430–439.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000b. Code compression as a variable in hardware/software co-design. In *International Workshop on Hardware/Software Co-Design*.

LEKATSAS, H., HENKEL, J., AND WOLF, W. 2000c. Code compression for low power embedded system design. In *Proceedings of the ACM/IEEE Design Automation Conf. (DAC 2000)*. ACM, New York, 430–439.

LEKATSAS, H. AND WOLF, W. 1998. Code compression for embedded systems. In *Proceedings of DAC 98*. ACM, New York, 516–521.

LEKATSAS, H. AND WOLF, W. 1999a. Random access decompression using binary arithmetic coding. In *Proceedings of the 1999 IEEE Data Compression Conference.* IEEE Computer Society Press, Los Alamitos, Calif., 306–315.

LEKATSAS, H. AND WOLF, W. 1999b. SAMC: A code compression algorithm for embedded processors. *IEEE Trans. CAD 18*, 12 (Dec.), 1689–1701.

LEMPEL, A. AND ZIV, J. 1976. On the complexity of finite sequences. *IEEE Trans. Inf. Theory 22*, 1 (Jan.), 75–81.

LIAO, S., DEVADAS, S., AND KEUTZER, K. 1999. A text-compression-based method for code size minimization in embedded systems. *ACM Trans. Des. Autom. Elec. Syst. 4*, 1 (Jan.), 12–38.

LIAO, S. Y., DEVADAS, S., AND KEUTZER, K. 1995. Code density optimization for embedded DSP processors using data compression techniques. In *Proceedings of the Conference on Advanced Research in VLSI*.

MUTH, R. 1999. Alto: A platform for object code modification. Ph.D. dissertation, Department of Computer Science, The University of Arizona.

MUTH, R., DEBRAY, S. K., WATTERSON, S., AND BOSSCHERE, K. D. 1998. `alto`: A link-time optimizer for the DEC Alpha. Tech. Rep. 98-14, Dept. of Computer Science, The University of Arizona. Dec.

NELSON, M. AND GAILLY, J.-L. 1995. *The Data Compression Book*, 2nd ed. M&T Books.

VAN DE WIEL, R. 2001. The code compaction bibliography. `http://www.extra.research.philips.com/ccb/`

VAN DE WIEL, R., AUGUSTEIJN, L., BINK, A., AND HOOGENDIJK, P. 2001. Code compaction: Reducing memory cost of embedded software. Philips white paper. See `http://www.extra.research.philips.com/`

VAN DE WIEL, R. AND HOOGENDIJK, P. 2001. Belt-tightening in software. *Philips Res. Passw. Mag.*, 16–19.

WELCH, T. A. 1984. A technique for high-performance data compression. *Comput. Mag. Comput. Group News IEEE Comput. Group Soc. 17*, 6 (June), 8–19.

WITTEN, I. H., NEAL, R. M., AND CLEARY, J. G. 1987. Arithmetic coding for data compression. *Commun. ACM 30*, 6 (June), 520–540.

WOLFE, A. AND CHANIN, A. 1992. Executing compressed programs on an embedded RISC architecture. In *Proceedings of the International Symposium on Microarchitecture (Micro-25)*. 81–91.

YOSHIDA, Y., SONG, B.-Y., OKUHATA, H., ONOYE, T., AND SHIRAKAWA, I. 1997. An object compression approach to embedded processors. In *Proceedings of the International Symposium on Low-Power Electronics and Design (ISLPED-97)*. 265–268.

ZIV, J. AND LEMPEL, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23*, 3 (May), 337–343.

Received December 2001; accepted June 2003