Images of Code: Lossy Compression for Native Instructions

Marcelino Rodriguez-Cancio
University of Rennes 1
Rennes, France
me@marcelinorc.com

Jules White
Vanderbilt University
Nashville Tennessee, USA
jules@dre.vanderbilt.edu

Benoit Baudry
KHT
Stockholm, Sweden
baudry@kth.se

ABSTRACT

Developers can use lossy compression on images and many other artifacts to reduce size and improve network transfer times. Native program instructions, however, are typically not considered candidates for lossy compression since arbitrary losses in instructions may dramatically affect program output. In this paper we show that lossy compression of compiled native instructions is possible in certain circumstances. We demonstrate that the instructions sequence of a program can be lossily translated into a separate but equivalent program with instruction-wise differences, which still produces the same output. We contribute the novel insight that it is possible to exploit such instruction differences to design lossy compression schemes for native code. We support this idea with sound and unsound program transformations that improve performance of compression techniques such as Run-Length (RLE), Huffman and LZ77. We also show that large areas of code can endure tampered instructions with no impact on the output, a result consistent with previous works from various communities.

CCS CONCEPTS

• **Theory of computation → Data compression**; • **Hardware** → *Sensor applications and deployments*;

KEYWORDS

lossy compression, software diversity, approximate computing

ACM Reference Format:
Marcelino Rodriguez-Cancio, Jules White, and Benoit Baudry. 2018. Images of Code: Lossy Compression for Native Instructions. In *ICSE-NIER'18: 40th International Conference on Software Engineering: New Ideas and Emerging Results Track, May 27-June 3 2018, Gothenburg, Sweden.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3183399.3183409

1 INTRODUCTION

Code compression techniques [5] are essential for domains such as embedded systems, the Internet of Things or Wireless Sensor Networks (IoT/WSN). Smaller code size reduces memory requirements of embedded systems. In turn, this translates into considerable savings in energy and manufacturing costs [8, 9]. Code compression also enables a reduction in network traffic [1], which accounts for enormous amount of energy consumption in IoT/WSN devices. Therefore, when remotely reprogramming IoT/WSN devices, efficient code compression schemes are a key aspect of performance.

In this paper, we show that it is possible to design lossy algorithms yielding higher compression ratios for ARM instructions than state of the art tool. These increased compression ratios enable to reduce energy consumption in over-the-air firmware update operations. Our new idea of lossy compression is founded on the areas of software diversity [3], approximate computing [10] and fault tolerance communities.

To the best of our knowledge, all existing code compression schemes are lossless. This limits the ability of a compressor to reduce data size, as the requirement to maintain all information is enforced. This constraint is due to the belief that code cannot be tampered with without modifying the program's behavior. While this intuition holds for the most part, it is not entirely true for *all* the code. Recent works in the areas of automatic software diversification and approximate computing have came to challenge this notion. These works have revealed the existence of *forgiving zones* where some instructions can be exchanged by others, still allowing the program to execute and maintain its Quality of Service (QoS) within acceptable levels. The existence of such zones allows the encoder to abandon lossless requirements when compressing code.

To understand why forgiving zones enable lossy compression of code, we think of such zones as images. In images, lossy compressors, such as chroma samplers, exploit the fact that humans cannot detect small differences in colors. This allows for the loss of information regarding the exact pixels' color, making the resulting compressed image not identical to the original, but still perceptually identical to a human. The same principle applies to instructions in forgiving zones: it is possible to lose information regarding the exact instruction at a given address, knowing that such instructions can be replaced by another similar one, still having the resulting program yield the same correct output.

We found that these forgiving zones account for an important part (22% - 55%) of the programs we inspected. This demonstrates the existence of large 'images of code' within the programs where lossy compression is possible. In other words, a compression scheme can lose information in large portions of the program.

In this paper, we support our arguments for lossy compression of code with the following evidence: (i) a set of unsound program transformations that modify a list of native ARM instructions, aiming at improving the efficiency of existing compression techniques such as Run-Length (RLE), Huffman and LZ77 encodings; (ii) results that demonstrate that such program transformations actually improve compression ratios for the programs in our dataset, while still allowing the programs to produce correct outputs; (iii) a study of the extent to which the programs in our dataset contain portions that can endure tampered instructions; (iv) a literature review

ICSE-NIER'18, May 27-June 3 2018, Gothenburg, Sweden

ACM ISBN 978-1-4503-5662-6/18/05…$15.00
https://doi.org/10.1145/3183399.3183409

showing that other authors have also found large forgiving zones, which we see as evidence of the feasibility of lossy compression beyond the field of embedded systems.

2 RELATED WORK

Code Compression. A significant number of works [4, 6, 8] have proposed code compression techniques. Lekatsas et. al. [8] realized that assigning Huffman codes to all instructions was less efficient than assigning codes to the different parts of an instruction (opcodes, registers, conditionals) separately. Benini et.al. [4] store the code compressed and then rely on a hardware unit to decompress it on the fly, prior to passing it to the CPU. The ARM Thumbs ISA [2] encodes a subset of the ARM ISA in 16-bits.

While code-size reduction is a well established field of study [5], few significant improvements have been made lately in program compression. This comes from the observation that existing methods have reached the limit in which no more compression can be attained without losing information. Also, as most works in code compression date before the discovery of forgiving zones by the approximate computing and software diversity communities, to the best of our knowledge all existing methods are lossless.

Code transformations. Compaction [6] is a technique that exploits sound transformations to reduce program size. The novelty of the ideas presented in this paper w.r.t compaction, rely on using unsound transformations that change the behavior of the program, losing information and producing a different set of instructions.

Forgiving Zones. Other authors have also found large Forgiving Zones inside programs in the areas of approximate computing [10], software diversification [3] and fault tolerance [7, 11]. This supports our belief that the results presented here are not limited to a small set of embedded systems (or ARM instructions), but instead they constitute a common feature of software that can be exploited to create lossy compression algorithms for code.

3 ARM BACKGROUND

We now provide some notions of the ARMv5 32-bit Instruction Set Architecture (ISA) required to understand the proposed lossy transformations. The complete reference for the ARM assembler language can be found elsewhere [1]. Each ARMv5 32-bits instruction is encoded in a 32-bit integer as follows:

```
opcode_<conditional> <operands>
```

ARM instructions contain one opcode field, one conditional field and one operand field. The opcode field defines what the instruction does. The conditional field defines if the instruction can be executed, which will depend on the value held by the Current Processor Status Register(CPSR). The exception to this rule is the ‘`al`’ conditional, that makes the instruction always execute. Operands can be one of the following: (i) a list of registers, (ii) a list of constant values or (iii) a list of registers and constant values. Examples of valid instructions are:

- `b_eq 0x1234`: Branches to address `0x1234` *if* bit ‘equals’ (eq) is set in the CPRS.
- `mov r0, r1`: Moves *always* the content of register `r1` to `r0`. As the conditional field is ‘`al`’, this instruction always executes, independently of the value stored on the CPRS.

4 TRANSFORMATIONS

We propose four different transformations that improve compression at the cost of losing information: (i) Lossy Huffman (ii) Lossy RLE (iii) Lossy LZ7 and (iv) Truncation [2]. This section goes into the details of these transformations and provides examples of the forgiving zones where they are applied.

(i) Lossy Huffman: This transformation consists in modifying the program to improve the compression ratio of a lossless Huffman encoding. To encode the program using a lossless Huffman, we follow the idea on [8] which consists in assigning Huffman codes to individual parts of instructions (opcodes, registers, conditionals), instead of assigning codes to the complete instructions at once.

Then, we replace low frequency tokens by high frequency tokens, effectively losing information. As frequent tokens require less bits to encode in the Huffman encoding, the size of the resulting encoding was reduced. Listings 1 and 3 are examples of forgiving zones transformed using Lossy Huffman.

(ii) Lossy RLE: RLE replaces repeated occurrences of a token by a counter and the token (i.e. {`r0`, `r0`, `r0`} becomes {3, `r0`}). The Lossy RLE consists in modifying the program to improve the compression ratio yielded by a lossless RLE encoding. To encode instructions using a lossless RLE, we store the instructions divided by parts (i.e. store all conditionals first, then store all opcodes, then store all registers). Then we modify the program to increase the run's lengths, as larger runs require less space. Listing 1 shows an example of such modifications. In the original program, destinations registers are `r0`, `r0`, `r6`, while in the modified program destination registers are `r0`, `r0`, `r0`, which is a larger run.

(iii) Lossy LZ77: The LZ77 algorithm replaces common substrings by references. The lossy transformation consists in replacing instructions by others to artificially create new substrings. Listing 3 presents an example of this, where instruction ‘`mov r1, #0`’ is replaced by ‘`sub r2, r2, #0`’ to create the ‘`sub r2`’ substring.

(iv) Truncation: This transformation is inspired by the Chroma Sampler used in images, which truncates the least significant bits, losing information in the process. It does not try to improve any existing lossless encoding. We truncate instructions to 28 bits, losing the information in the 4 least significant bits of the instruction. When decoding truncated instructions, the missing bits are filled with zeros. Listing 2 exemplifies the effect of this, showing that the resulting program branches to an instruction two addresses earlier than the original.

5 FORGIVING ZONES

This section provides examples of forgiving zones we have found. These examples came from the `basicmath` program of our dataset. All zones found in the programs of our dataset can be found on the website of this paper [3]. We present both the original and transformed programs, then we expand on why the lossy transformation

[1] http://infocenter.arm.com

[2] The transformations' source code can be found here: https://github.com/lossycomp

[3] https://github.com/lossycomp

```
1    eors_mi r0, r6, r0
2    eors_gt r0, pc, r0
3    strbt_vs r6,{r6},-r6,ror#12
```

```
1    eors_lt  r0, r6, r0
2    eors_gt  r0, pc, r0
3    strbt_vs r0,{r0},-r0,ror#12
```

Listing 1: Forgiving Zone 1. Original program on top, modified below

```
1    str r3, {fp, #-0x34}
2    str r4, {fp, #-0x30}
3    mov r2, #0
4    ldr r3, {pc, #0x148}
5    sub r1, fp, #0x34
```

Listing 2: Forgiving Zone 2, example of truncation

```
1    mov r1, #0
2    ldr r3, [pc, #0xf4]
3    sub r2, fp, #0x34
```

```
1    sub r2, r2, #0
2    ldr r3, [pc, #0xf4]
3    sub r2, fp, #0x34
```

Listing 3: Forgiving Zone 3. The modified program (bottom) exchages instruction at line 1 by other one.

produces a program which still exhibits an acceptable behavior (i.e. produces the same output as the original, given the same input).

Forgiving Zone 1 is shown in Listing 1. The zone has been processed with two transformations: the registers used by the instruction in line 3 (`r6` to `r0`) and the conditional of instruction at line 1 (`mi` to `lt`). We have found that some instructions are conditioned to situations that never occur, like overflow (conditional `vs`) in operations with bounded values. This allows to modify the rest of the instruction, like in line 3, where registers are modified to increase token frequency and run length. Also, two conditionals might be exchangeable in certain contexts, such as line 1 where 'minus/negative' (`mi`) and 'signed less than' '`lt`) are the same if the overflow bit is clear in the CPSR register. In our dataset, conditional '`lt`' was more frequent that '`mi`', hence the change removed an uncommon token.

Forgiving Zone 2 is shown in Listing 2. The example shows the effect of truncation in branching instructions. Truncation results in the loss of information in the branching address, therefore when decoding the instruction, the loss of information causes the branching address to be approximate. Listing 2 shows a part of the program where one of such modified branches jumps to. The original instruction jumps to Line 3 while the modified one jumps to Line 1. The two extra executed instructions store the value of registers `r3` and `r4` to memory without consequences to the program.

Forgiving Zone 3 is shown in Listing 3. Here, the value assigned to `r1` in line 1 is never used, allowing to replace the instruction by another one. This change creates an artificial substring.

6 EXPERIMENTAL RESULTS

In this section we provide evidence to support our claim that lossy compression of code is possible. Initially we provide a study of the size of the forgiving zones in our dataset. Later we compare the compression ratios achieved with well known compression schemes with the ratio achieved with those same methods after the information loss caused by the transformations exposed in Sec. 4.

6.1 Dataset

We study seven programs from the MiBench [4] suite, a collection of representative embedded applications and libraries.Table 1 shows the programs of our dataset. All sources for the programs in our dataset were obtained from the MiBench website and then compiled for the ARMv5 32-bit ISA. The compiled native instructions were used as input to our experiments.

Table 1: Case studies for our experiments

Program	Code Size in Bytes	Purpose
`sha`	3096 bytes	Set of Hash Algorithm SHA-0
`qsort_small`	1076 bytes	Tests the `qsort` function
`fft`	8356 bytes	Fast Fourier Transform
`dijkstra_small`	2292 bytes	Dijkstra's Algorithm
`crc32`	1140 bytes	CRC error-detecting codes
`bitcount`	5880 bytes	Bit counting functions
`basicmath`	7244 bytes	Mathematical computations

6.2 Size of Forgiving Zones

Figure 1 shows which parts of the programs in our dataset were forgiving. The figure shows one stacked bar per program. Each bar is divided in four segments 'Forgiving', 'QoS', 'Crash' and 'Froze'. In turn, each segment represents a percentage of the program's instructions that were forgiving, crashed or froze the program or that produced a different output.

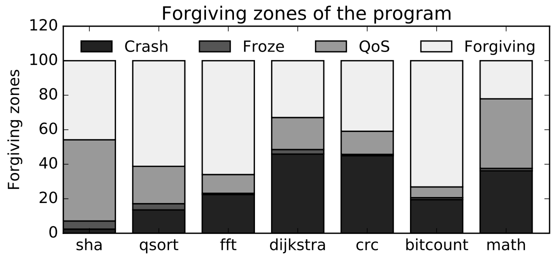

Figure 1: Size of forgiving zones of dataset's programs

Instructions in Forgiving Zones are those instructions that could be exchanged by at least another instruction while maintaining an acceptable behavior. The non-forgiving instructions were divided in three groups 'Crash', 'Froze' and 'QoS'. Crash instructions were those that crashed the program in most attempts of being exchanged. Similarly, Froze instructions where those for which a replacement usually prevented the program from exiting at all. Finally, QoS instructions were those that produced an incorrect output in most attempts of replacement. If an instruction caused more than one type of error, we selected the most common one. For example, if an instruction was replaced five times and three replacements caused the program to crash, we placed the instruction in the Crash group.

[4] http://vhosts.eecs.umich.edu/mibench/

Figure 1 shows that a large portion of the program could be exchanged by at least another candidate. In three cases (qsort, fft and bitcount) the forgiving zones accounted for more than 50% of the instructions. The programs with larger forgiving zones are fft and bitcount. Not surprisingly, those are also the programs for which the lossy schemes improve the compression ratio the most.

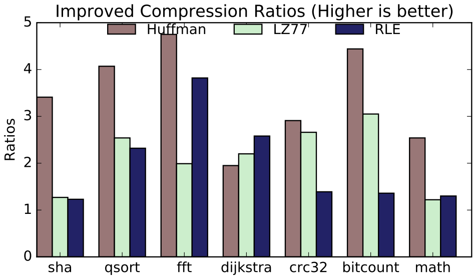

Figure 2: Improvement in compression ratio obtained with Lossy Huffman, Lossy RLE and Lossy LZ77.

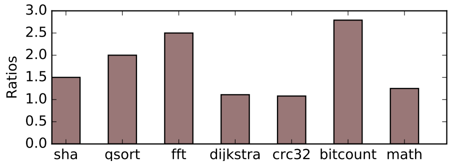

Figure 3: Compression ratio using truncation only (percent)

6.3 Compression ratio improvement

To measure the changes in compression ratios, we encode each program using a single encoder (Huffman, RLE and LZ77) each time (i.e. no combination of encoders are used) and we measure the lossless compression ratio obtained. In a second step, a set of instructions is transformed using the lossy transformations trying to maximize improvement in compression. If the program crashes or produces an incorrect output, a different set of instructions is selected until the transformed program yields the same output as the original one. The resulting transformed program is compressed again using each encoding to obtain the lossy compression ratio. Figure 2 shows the percent to which the lossy compression ratio is higher than the lossless ratio. Each program in our dataset is represented by three bars. Each bar in the figure represents the following metric's value: $(lossy/lossless) * 100$, for each encoding we studied (Huffman, LZ77 and RLE).

The results show that the presented lossy transformations improve the compression ratio over the lossless encoding between 1.2% - 4.7%. Higher improvements were achieved in programs with larger forgiving zones. Better improvements were obtained with Huffman encoding, as it sufficed to change a few infrequent tokens to obtain improvements when using this encoding. RLE and LZ77 require to change larger program portions to improve compression ratio, resulting in more rejected transformed program versions.

Figure3 shows the compression ratio obtained using the Truncation transformation. The values here represent the percent for which the program containing truncated instructions is smaller than the uncompressed original. The figure shows that truncated programs were between 1.1% and 2.8% smaller. Truncation was the least efficient transformation since many transformed programs were rejected as they produced incorrect results.

7 CONCLUSIONS AND FUTURE WORK

This work introduces the following new idea: lossy compression of native instructions is possible. We supported this claim by providing a series of lossy transformations that modify a list of instructions to improve the efficacy of existing compression algorithms. Our emerging results show the beneficial impact of such transformations in compression. We also provide evidence of large forgiving zones, where instructions can be exchanged to increase compression ratio, similarly to how pixel colors in images can be modified for the same purpose. This is consistent with results from other authors, who also have found large Forgiving Zones inside programs.

While the improvements obtained using our lossy transformations might not justify the risk of transforming a program, these proofs of concept do provide evidence that lossy compression of programs is possible and invite to further research in this direction. Also, it is important to notice that in our experiments we only consider a program correct if the output is identical to the original, not exploiting the fact that some outputs might be considered correct even if not identical to the original, something frequently exploited by the approximate computing community. Hence, our future work will consist in developing a lossy compression algorithm for native instructions, aiming to improve the compression ratio of the state-of-the-art methods specifically designed to compress code.

REFERENCES

[1] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. 2009. Energy Conservation in Wireless Sensor Networks: A Survey. *Ad Hoc Networks* 7, 3 (May 2009), 537–568. https://doi.org/10.1016/j.adhoc.2008.06.003

[2] ARM. 2018. The Thumb Instruction Set. http://infocenter.arm.com. (2018).

[3] B. Baudry and M. Monperrus. 2015. The Multiple Facets of Software Diversity: Recent Developments in Year 2000 and Beyond. *ACM Comput. Surv.* 48, 1 (2015).

[4] L. Benini, A. Macii, E. Macii, and M. Poncino. 1999. Selective Instruction Compression for Memory Energy Reduction in Embedded Systems. In *Proceedings. 1999 International Symposium on Low Power Electronics and Design (Cat. No.99TH8477).*

[5] Árpád Beszédes, Rudolf Ferenc, Tibor Gyimóthy, André Dolenc, and Konsta Karsisto. 2003. Survey of Code-Size Reduction Methods. *ACM Comput. Surv.* 35, 3 (Sept. 2003), 223–267. https://doi.org/10.1145/937503.937504

[6] H. He, J. Trimble, S. Perianayagam, S. Debray, and G. Andrews. 2007. Code Compaction of an Operating System Kernel. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '07).* IEEE Computer Society, Washington, DC, USA.

[7] A. Heinig, V. J. Mooney, F. Schmoll, P. Marwedel, K. Palem, and M. Engel. 2012. Classification-Based Improvement of Application Robustness and Quality of Service in Probabilistic Computer Systems. In *Architecture of Computing Systems – ARCS 2012.* https://doi.org/10.1007/978-3-642-28293-5_1

[8] H. Lekatsas, J. Henkel, and W. Wolf. 2000. Code Compression for Low Power Embedded System Design. In *Proceedings of the 37th Annual Design Automation Conference.* ACM, New York, NY, USA.

[9] H. Lekatsas and W. Wolf. 2006. SAMC: A Code Compression Algorithm for Embedded Processors. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 18, 12 (Nov. 2006), 1689–1701. https://doi.org/10.1109/43.811316

[10] Sparsh Mittal. 2016. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.* 48, 4 (March 2016), 62:1–62:33. https://doi.org/10.1145/2893356

[11] Darshan D. Thaker, Diana Franklin, John Oliver, Susmit Biswas, Derek Lockhart, Tzvetan S. Metodi, and Frederic T. Chong. 2006. Characterization of Error-Tolerant Applications When Protecting Control Data. In *IEEE International Symposium on Workload Characterization (IISWC).*