Archiving Programs for the Future

Alexander Binun, Shlomi Dolev and Yin Li
Department of Computer Science, Ben-Gurion University of the Negev, Beer Sheva, Israel

Keywords: Data Archiving, Long-term Bit Preservation, OISC, Subleq.

Abstract: This paper presents a novel approach for long-term software archiving which is based on preserving programs as bit blocks. A simple machine that is able to execute a single command is used to interpret these bit blocks. We suggest to compile the existing programs into the bit representation of the One-Instruction-Set computer (OISC) command "SUBtract and Branch if Less than or EQual to zero", shortly Subleq. This is in order to keep the resulting bit stream using error correcting code in a reliable storage unit. At any moment, this bit stream can be executed by a simple interpreter that possesses the functionality of a basic Random Access Machine. Furthermore, a compiler prototype based on an existing compiler and interpreter is also proposed to convert a program written by a procedural language (e.g., C) into the Subleq assembly language, and then translates it into a binary executable format. Error correcting is achieved by supplementing bit streams with Hamming codes. Our scheme nullifies the need to preserve legacy hardware in order to support/operate preserved software systems thus serving as a program "time capsule" for the future.

1 INTRODUCTION

Information preservation is one of the most crucial challenges facing scholarly communities today. We have large amounts of digital data, such as e-journals, e-books, emails and blogs that will be preserved for generations to come. Nowadays, there are certain programs (National Digital Information Infrastructure, 2015) and organizations (Research Information Info, 2015) that are dealing with digital asset preservation, before technological obsolescence, or data loss creep in.

Attempts to preserve information for future generations have been known since the ancient times, for example ancient Egyptian or Cretan hieroglyphs and Babylonian cuneiform scripts. Organized efforts to conserve information for future generations are observed during the recent centuries. These efforts manifest in placing and tracking time capsules (Time Capsule, 2015), i.e. containers of such information. However, languages do not become dead and time capsules do not disintegrate during a short period. Modern works on computerized long-term data preservation such as (Campisi et al., 2009) focus mainly on supplementing data with auxiliary information needed for proper interpretation (metadata) and synchronizing metadata.

In this paper, our work is devoted to finding a uniform solution for the preservation of code, in an architectural and compiler independent form, in the extreme, to ensure preservation of codes for a long time – for decades, centuries, or even for thousands of years. To put it another way, we suggest to store program code in a time capsule that should be opened in the future for technology preservation and can still be executed in the future. For example, imagine that we need to run an important program written in PL/I with DOS on PDP-11, with procedure written in Pascal for MAC. Obviously, backward compatibility issues become impassable barriers for executing legacy programs as time goes by. Thus, a new paradigm for software repository is extremely important.

Our approach focuses on different aspects:

- We suggest using a minimalistic and simple command, namely, OISC machine commands that can be easily interpreted by any future machine, or even manually.
- In the longer range future (thousands of years), people may cease to understand the rationale behind code, its effects and even the language used to express the code ideas. We emphasize the need to develop a *universal writing system* in

Binun, A., Dolev, S. and Li, Y..
Archiving Programs for the Future.
In *Proceedings of the 6th International Workshop on Software Knowledge (SKY 2015)*, pages 53-57
ISBN: 978-989-758-162-5
Copyright © 2015 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

order to keep our code today understandable for future generations.

We assume that the design concepts behind the code will become less relevant in a certain amount of years. Our assumption is based on the fast pace of software evolution. In the 1970s efficiency was amongst the major requirements for C programs which were often enriched by assembly fragments. Nowadays, thanks to modern and powerful computers, (that can endure huge libraries) readability and maintainability have become "affordable" and actual issues. The design rationale behind modern OO libraries (such as Java Beans) may become less relevant in the upcoming 30-40 years but sometimes we might be willing to run programs based on these libraries. We need to be able to understand the *execution effects* rather than the *design details* of old programs.

We therefore focus on *bit preservation*, i.e., the ability to restore the bits of a program stored for a long period of time and obtain their effect, namely, run the bytecode of the restored program. The bit-level interpreter should thus embody some fundamental computing ideas based on bit manipulation (e.g., Random Access Machine, Goldstine et al., 1947)). This concept is powerful, simple to simulate, efficient (with relation to Turing machine) and has not be changed (and will probably not change) since the beginning of the computer science era.

However, there are several challenges for long-term code preservation as in (Factor et al., 2009). These are storage media degradation, hardware obsolescence and hardware failures. These drawbacks are aggravated by targeted attacks against media systems staged by malicious parties. As a result, corrupted data may be fetched from the storage. To address this challenge, we use the Subleq machine as in (Mazonka et al., 2011) as an appropriate format for keeping and running code. Redundancy is also employed when storing the data, augmenting data with error correcting information. The use of error correcting codes enables integrity checks and correction during data access. One possible error correcting code is Hamming code that can be utilized for the preservation of every bit of a command.

RAM based Subleq is proven to be Turing-equivalent and therefore capable of expressing code for any feasible computation. In our scheme any code is converted into the Subleq assembly format and then into a bit stream. Such bit streams enriched with the Hamming error-correcting codes (Arazi, 1988) are placed into media storage units (e.g., CD-ROMS, or even into a plate with Braille style marks on a surface).

The second challenge is to ensure that our descendants in future will want to learn more about our lives and *will be able to do so.* Even if we focus solely on execution effects, what is the input/output format? Textual input/output is usually expressed in some modern language (today mostly in English). Images and sound might be based on modern cultural references (like modern computers, cars). With that being said, modern languages and other cultural references may be forgotten in the future, sharing the destiny of undeciphered ancient scripts. A possible solution would be to propose a set of visual or tactile signs allowing any language group to express their own utterances; that is, using a universal writing system (Universal Writing System, 2015). Possible inputs/outputs can be translated into this system and deciphered in the future. There is a clear need to bring together computer scientists, linguists, psychologists etc. into the joint interdisciplinary project aimed at developing a universal writing system.

Paper Organization. Section 2 outlines the principles behind preserving code excerpts as bit streams. It includes compiling any procedural language into the Subleq assembly language and further into bytecode as well as integrating error-correcting capabilities. Section 3 outlines the main idea behind the usage of a universal writing system in our context and provides an example of a simple C program converted into the bit stream format. Finally, some conclusions are drawn in Section 4.

2 PRESERVATION OF CODE EXCERPTS

2.1 Keeping Code as Bit Stream

A Subleq program represents an infinite array of memory cells, where each cell holds an integer number. This number can be an address of another memory cell. The Subleq interpreter considers this array as a sequence of instructions; each instruction has 3 operands: `A,B,C`. An operand may be either an integer number or a symbolic label describing a memory address. Execution of a Subleq instruction subtracts the value in the memory cell at the address stored in `A` from the content of a memory cell at the address stored in `B` and then writes the result back into the cell with the address in `B`. If the value after subtraction in `B` is less than or equal to zero, the

execution jumps to the address specified in C. Otherwise, the execution continues to the next instruction, which is the address of the memory cell next to the current cell.

We use the compiler that accepts a simplified version of C language (mentioned as *Higher Subleq* in (Mazonka et al., 2011)) that is referenced throughout the article as **the compiler**. Higher Subleq does not have a preprocessor, does not support structures, bit fields, abstract declarations and bit operations. It has only one underlying type **int**. Depending on the configuration the compiler can produce:

- A Subleq assembly language module.
- An executable Subleq module. It is assumed that the address of the first instruction in a Subleq assembly program is zero so every Subleq instruction is converted into a triple of integer numbers.

We configure the compiler to transform a program written in a high-level language into an executable Subleq module. Then we convert the integer numbers of this module into the binary format. Thus, eventually a program written in a high-level language is converted and stored as an array of bits.

There are a number of high-level languages that possess similar syntax and semantics for basic operations. Based on the similarities and the modularity of the compiler, it can seamlessly adapt to many procedural languages.

2.2 Using Error-correcting Codes

Long-term preservation of bit streams may be challenged by storage media degradation, hardware obsolescence and hardware failures (Factor et al., 2009). Media systems may be also attacked by malicious users. As a result, the stored bits that need to be interpreted may be corrupted.

To preserve the integrity of bit streams we stored those streams using Hamming error-correcting codes as in (Arazi, 1988). We omit the details of Hamming codes as it is a classical basic error-correcting code; in fact we can use any other code. The storage system recognizes code excerpts by their IDs. Periodic storage scans of stored bit streams can exploit its redundancy, preserving the data integrity.

When a user wants to retrieve a code excerpt, the corresponding excerpt ID is sent to the storage system. The required excerpt with error correction codes is fetched. When a bit stream arrives at the interpreter site, Hamming codes are stripped off, necessary correction is performed and the “clean” bit stream is executed.

3 A UNIVERSAL WRITING SYSTEM

All natural languages carry a lot of irregularities in grammar which make them more difficult to learn. They are also associated with the national and cultural dominance of the nation that speaks it as its mother tongue. Therefore creating an artificial or constructing language lies at the core of an approach of universal language where people from different nations can communicate, just like numbers and calculations became universal.

The notion of “phoneme“, that is a single “unit” of sound that has meaning in any language is at the heart of the ongoing efforts to devise a universal writing system. This is because pronouncing of individual letters may differ in various languages and depend on their context; English is known for many irregularities and exceptional cases.

It should be noted that writers of ancient manuscripts did not invest decent efforts in making their writing understandable for future generations. As a result, some dead languages (e.g., Cretan hieroglyphs) still remain undeciphered. The ancient Egyptian hieroglyphs were not deciphered until several stones with bilingual writings in two languages (Egyptian and ancient Greek) were found (the so-called Rosetta Stones). Ancient Greek was known to archaeologists and served as a universal writing system when deciphering Egyptian.

As a first attempt to instruct our descendants how to use our storage system, we suggest to use the pictogram approach, to draw several pictures describing an operator executing a program. We thus will be following ancient Egyptians whose hieroglyphs carried intuitive sense (e.g., the hieroglyph “king” resembles a king, one can even recognize the crown, see Figure 1).

Figure 1: The picture of crown represents ‘the King”.

3.1 A Simple Program Written as a Bit Stream

In this subsection, we give a very simple example to show the whole procedure of our scheme. Consider the following simple program written in *Higher Subleq*:

```
int main(void){
    int x=0;
    x=x-2;
}
```

The Subleq assembly code for this fragment is.

```
main:
    c1 x ?
    x=x-2;
. x:0 c1:2
```

- The semantics behind the Subleq code is: The row starting from "." denotes the data section where variables are initialized. Thus `x` gets the value `0`, the constant `2` is placed into the variable "`c1`".
- The instruction "`c1 x ?`" performs subtraction of the value stored at the address denoted by "`c1`" from the value stored at the address denoted by "`x`". Afterwards the execution proceeds to the next instruction (whose address is denoted by "`?`").

The bytecode (and the bit stream resulting from it) are obtained by setting the address of the first instruction to 0. We assume that 2-byte integer numbers hold actual addresses.

The actual address of "`x`" is resolved to `402`, the actual address of "`c1`" is `404`, the instruction following "`c1 x ?`" has the address 350. The instruction bytecode has the form "`404 402 350`". The corresponding bit stream is:

```
0000000110010100
0000000110010010
0000000101011110
```

If we use the Hamming (7,4)-code principle to add redundant bit for error correcting, the above bit stream is extended as follows:

```
0000000 1101001 0011001 0101010
0000000 1101001 0011001 1001100
0000000 1101001 0100101 0010110.
```

In above codes, every 4 bits are extended into 7 bits, the bits underlined are additional parity bits that we added.

3.2 Visual Puzzle Styled Manual Guide

As mentioned previously, we use a pictogram to explain our operations and ideas. More explicitly, a "user manual" is needed to explain the process and should be stored as a part of the record.

Here, we can follow the rule of visual puzzles to write such a manual. For example, the following figure gives a manual for Subleq

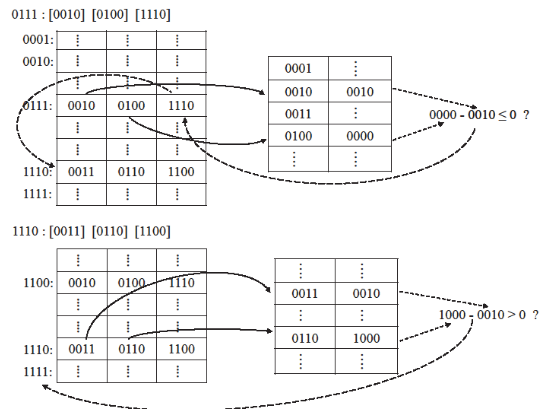

Figure 2: The visual manual guide for Subleq.

In the above figure, we use different types of arrows to show one Subleq command step by step, and describe the way it is manipulated. For the sake of simplicity, Figure 2 does not include a visual puzzle for explaining the subtractions operation and comparison. One can easily add related con- tents to make these operations comprehensible. The manual for Hamming error correcting code could be made in a similar fashion.

In the future, when people want to recover the program, they first check the integrity of the bit stream using these parity bits. Then, they delete these parity bits and recover the original bits stream. Intelligent people should be able to follow the visual puzzle manual and recover the whole program, with the input and output.

4 CONCLUSIONS

This paper proposed a new prototype for program archiving for the future. The OISC command Subleq is used to simplify the classic complex program codes and Hamming error-correcting code to protect the program itself. Meanwhile, visual puzzles are utilized to design instruction manual. All these tricks form an alternative universal writing system. This scheme allows us to archive various programs without preserving different hardware and software platforms, allowing for future execution by future generations of computers. At last our scheme can be used as an Arecibo message as in (Arecibo Message, 2015).

ACKNOWLEDGEMENTS

We thank Oleg Mazonka for providing us with the Subleq compiler prototype and for giving thorough explanations regarding its usage. We thank the support of the Rita Altura Trust Chair in Computer Science. We also indebted to the Lynne and William Frankel center for its generous support.

REFERENCES

B. Arazi, 1998. A Commonsense Approach to the Theory of Error- Correcting Codes. Computer System Series, The MIT Press, ISBN-10:0262010984.

P. Campisi, E. Maiorana, E. Ducci Teri, A. Neri, 2009. Challenges to long-term data preservation: a glimpse of the Italian experience. In DSP'09: Proceedings of the 16th international conference on Digital Signal Processing. Pages 120-127, IEEE Press, Piscataway, NJ.

M. Factor, E. Henis, D. Naor, S. Rabinovici-Cohen, P. Reshef, S. Ronen, G. Michetti and M. Guercio, 2009. Authenticity and Provenance in Long Term Digital Preservation: Modeling and Implementation in Preservation Aware Storage. In TAPP09: First Workshop on on Theory and Practice of Provenance, Pages 6:1-6:10, San Francisco, USA.

H. H. Goldstine and J. von Neumann,1947. Planning and Coding of the Problems for an Electronic Computing Instrument. Institute for Advanced Study, Princeton. McGraw-Hill Book Company, New York.

O. Mazonka and A. Kolodin,2011. A Simple Multi-Processor Computer Based on Subleq. In CoRR, Volume 1106.2593.

Time Capsule, 2015. http://en.wikipedia.org/wiki/Time_capsule

A Universal Writing System, 2015. http://www.omniglot.com/pdfs/phonbook.pdf

National Digital Information Infrastructure, 2015. http://en.wikipedia.org/wiki/National_Digital_Information_Infrastructure

Research Information Info, 2015. http://www.researchinformation.info/features/feature.php?feature_id=506

Arecibo Message, 2015. http://en.wikipedia.org/wiki/Arecibo_message