Archiving programs for the future (Extended Abstract)

Alexander Binun, Shlomo Dolev, Yin Li
Dept. of Computer Science,
Ben-Gurion University of the Negev, Israel
{dolev,binun}@cs.bgu.ac.il
yunfeiyangli@gmail.com

Oleg Mazonka
Dept. of Computer Science,
New York University
mazonka@gmail.com

ABSTRACT

This paper presents the approach for long-term software archiving that is based on preserving programs as bit blocks that can be interpreted by the Turing machine. We suggest to compile a program into the bit representation and to keep the resulting bit stream in a reliable storage. At any moment this bit stream can be executed by an interpreter that possesses the functionality of a Turing machine. Due to e.g. attacks on media storage systems or just transient media failures a corrupted bit block may be streamed for execution. Reliable execution is ensured by mixing error-correcting codes into bit streams.

We choose the One-Instruction-Set computer (OISC) as an example of the underlying interpreter. Any program is compiled into an OISC-compatible assembly language Subleq. We also provide a prototype compiler that converts a program written in a procedural language (like C) into the Subleq assembly language and further translates it into the binary executable format. Error correcting is achieved by mixing Hamming codes into bit streams and retransmitting a bit block if its checksum appears to signal failure. Our architecture is applicable to wide variety of underlying hardware and software.

Keywords

Data archiving, long-term bit preservation, OISC, Subleq.

1. Introduction

Our goal is to ensure preservation of code for a long time – tens of years, even centuries. Put another way, we suggest to store code in a "time capsule" that should be opened in the future and to help historians [5]. We reasonably assume that the semantics of these artifacts will become less relevant in a certain amount of years. Our assumption is motivated by the fast pace of software evolution. In 1970s efficiency was among 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 had become "affordable" and actual issues. The design rationale behind modern OO libraries (such as JavaBeans) may become less relevant in the upcoming 30-40 years but sometimes we might be willing to run programs based on these libraries. What we need from old programs written 30-40 years ago is their *execution effects*, not their *design details*.

We therefore focus on *bit preservation*, i.e. ability to restore the bits of a program stored during long years and obtain their effect, i.e. run the bytecode of the restored program. Semantics will eventually become less relevant. The bit-level interpreter should thus embody some fundamental computing idea based on bit manipulation (e.g. the RAM-based Turing machine) which has not changed (and won't probably change) since the beginning of the computer science era.

Yet one challenge posed by long-term preservation is storage media degradation, hardware obsolescence and hardware failures [4]. These are aggravated by targeted attacks against media systems staged by malicious parties. As a result corrupted data may be fetched from the storage. To stand up to this challenge we suggest to employ redundancy when storing the data (i.e. augment data with the *error correcting information*) and to enforce integrity checks when transmitting requested data from storage media (i.e. build the checksum for every transmitted block, repeating the action in case of any failure).

In practice we use the Subleq assembly language [2] as an appropriate format for keeping program artifacts of any complexity. This is because the Subleq machine equipped with RAM is Turing-complete. A Subleq assembly module is then converted to a bit stream.

Bit streams enriched with the Hamming error-correcting codes [3] are placed into media storage. However, transient failures may still disturb the retrieval process, resulting in corrupted data retrieval. To cope with this problem the receiver sends the Hamming-based checksum back to the media provider as an acknowledgement. If data is corrupted retransmission will be repeated (Automatic Repeat Request, [3]).

Paper organization. Section 2 outlines the principles behind compiling any procedural language into the Subleq assembly language and further into byte-code. Section 3 describes the error-correcting approach taken by us. Finally, concluding remarks appear in Section 4.

2. The Subleq assembly language

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 numbering starts from zero. The program is defined as a sequence of instructions read from the memory with the first instruction at the address zero. Ultimately a Subleq program can be stored as an array of bits.

The Subleq interpreter considers this array as a sequence of instructions; each instruction has 3 operands {**A**,**B**,**C**}. 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 or equal to zero, the execution jumps to the address specified in **C**.

Commented [S1]: It would be nicer to get some works that confirm our (so far speculative) suggestion. After all, the long-term understandability of the preserved data is still needed!

Otherwise execution continues to the next instruction, i.e. the address of the memory cell following **C**.

The Subleq machine is Turing-complete [2] so any reasonable program can be converted and executed on it. We use the compiler from a simplified version of C language (mentioned by the authors as *Higher Subleq*) into the Subleq assembly language [2], referenced throughout the article as *the compiler*. Higher Subleq does not have preprocessor, does not support structures, bit fields, abstract declarations and bit operations. It has only one underlying type **int**, but the keywords **int**, **char**, **void**, and their pointer derivatives can be used as synonyms.

A number of high-level command languages such as Basic possess the similar syntax and semantics so the compiler can be easily adapted to these languages. Taking into account the modularity of the compiler, we can seamlessly adapt it to many procedural languages.

3. Error Correcting

Long-term preservation of bit streams may be challenged by storage media degradation, hardware obsolescence and hardware failures [4]. Media systems may be also attacked by malicious users. As a result the bit stream delivered from the storage to the interpreter may become corrupted.

To preserve the integrity of bit streams we supply them with Hamming error-correcting codes before they are sent to the storage. The storage system recognizes code excerpts by their IDs. Periodic storage scans of stored bit streams can exploit this redundancy, preserving the data integrity.

When a user wants a code excerpt to be retrieved, 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.

The whole process can be described as a pseudo-code as follows:

```
store(program)
  begin
    BitStream hsq,hamming;
    Int ID;
    hsq:=compileIntoSubleq(program);
    hamming:=enrichWithHammingCodes(hsq);
    ID:=save hamming in the media;
  end;

fetch(ID)
  begin
    BitStream hsq,hamming;
    hamming:=fetchFromStorage(ID);
    hsq:=stripHamming(hamming);
    execute hsq;
  end;
```


4. Acknowledgements

Our thanks to ACM SIGCHI for allowing us to modify templates they had developed.

5. REFERENCES

[1] Campisi P., Maiorana E., Ducci Teri E., Neri A. 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, USA ©2009.

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

[3] Arazi, Benjamin. *A Commonsense Approach to the Theory of Error-Correcting Codes*. Computer System Series, The MIT Press (February 17, 1988), ISBN-10: 0262010984.

[4] Factor, M. , Henis, E. , Naor, D.,Rabinovici-Cohen, S., Reshef, P. , Ronen, S., Michetti, G. and Guercio, M. Authenticity and Provenance in Long Term Digital Preservation: Modeling and Implementation in Preservation Aware Storage. In *TAPP '09: First Workshop on on Theory and Practice of Provenance,* Pages 6:1-6:10, 2009, San Francisco, USA.

[5] The Time Capsule Idea, http://en.wikipedia.org/wiki/Time_capsule