High Level Language Translator with Machine Code as Representation of the Source Code

Samir Ribic, MScD,
Elektrotehnički Fakultet Sarajevo
Zmaja od Bosne b.b
Sarajevo, Bosnia Herzegovina
megaribi@epn.ba

Adnan Salihbegović PhD
Elektrotehnički Fakultet Sarajevo
Zmaja od Bosne b.b
Sarajevo, Bosnia Herzegovina
adnan.salihbegovic@etf.unsa.ba

Abstract. *High level programming languages are usually implemented as compilers, interpreters, pseudo compilers, just in time compilers and compact form interpreters. All of the mentioned approaches have their advantages and disadvantages. The approach proposed in this paper eliminates need for source code as physical entity, making the native machine code the only file that exists and is archived, while the special viewer/editor shows and edits it in a form understandable as a high level language. The viewer/editor acts as integrated all-in-one editor, decompiler and incremental compiler.*

Keywords. Compiling, decompiling, programming languages, reverse engineering

1. Introduction

The compilers most commonly perform translation from higher level languages to the machine code. They generate reasonably fast executable code, but they require compiling time, and to store and maintain separate source and machine code. The alternative to the compilers are interpreters, who only need to hold source code, but the code execution is slower and require additional program for execution support, the interpreter. There are also some hybrids, as an attempt to utilize the advantages from both approaches: pseudo compilers that convert source code to intermediate code (like Java compiler), or interpreters of the compact code (like FORTH interpreter).

There are also programs which convert from low level languages to the high level languages. A decompiler is a reverse engineering tool that takes as input a program in the form of an executable, and produces a high-level language representation of that program [9].

However, compilation and decompilation are usually regarded and carried out as two separate processes. The compilation is used during development process, again and again after every change in a program, while decompilation is used mainly for recovery of the lost source code, and then manual patching of the received HLL code [5].

It is proven that interpreting of pseudo-machine code can lead to decompilation, and symmetry between source and machine code [7]. But, is it possible to integrate compiler and decompiler in a unique program that will work so closely and eliminate the need for source code at all? In this paper we shall propose one approach to keep the user native program in executable machine code, without the existence of the source code as a separate entity. The special editor actually investigates the executable program, recognizes the machine code fragments as high-level language statements and displays them on the output media. If the programmer wants to edit the statement, the high level interpretation of the machine code fragment will be brought to the editor buffer. The programmer can then edit the statement, and it will be instantly recompiled to the machine code.

This approach differs from the usual debuggers, which display machine code instructions as mnemonics. It is also different from the approach of the debuggers that display high-level source code line next to the machine code instruction too, like GDB. With these debuggers, the source code is read from separate file, and it is not editable at high level language.

2. Overal architecture of the proposed system

The proposed system consists of front end editor, recognition and editable buffer, temporary buffer for generated code, syntax and semantic analyzer with code generator ("compiler part"), code recognizer ("decompiler part"), patcher of the user executable program and executable program itself, as displayed on Figure 1.

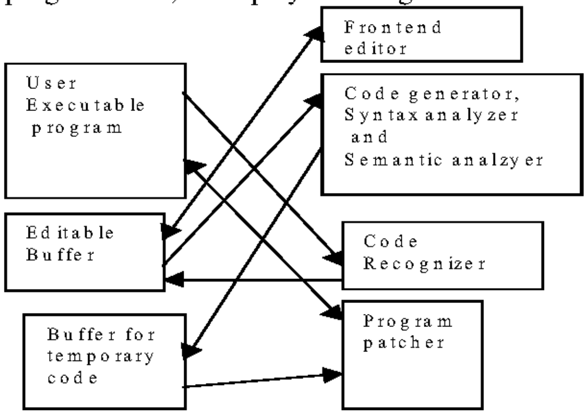

Figure 1: System components

The front-end editor is used for interaction with programmer. It reads from editable buffer area and writes into the editable buffer. The code recognizer (decompiler) can write into the editable buffer too. The edited lines are submitted to code generation and converted to temporary code buffer. Program patcher reallocates user executable program and copies code from temporary code buffer into the user executable program. The temporary code buffer is not required on the architectures with simple binary program format.

In the rest of this paper we shall propose solutions for some problems in design of the subsystems. As this part is much technical, we suggest additional reading like [1], [2], [3], [4] and [5], to get more knowledge about compilation and decompilation.

3. Generating reversible machine code and resolving ambiguity problem

The code is held in a native machine language, but the possibility to uniquely regenerate high level code, poses certain requirements.

a) The most important requirement is that executable code has to be relocatable. Without this property, inserting, editing and deleting the program statements would be impossible. Every insertion or deletion of the code moves the rest of the user executable program in memory, and updates some pointers stored inside program. This requirement can be achieved with separation of the program to fixed and relocatable part, and generating only relative jumps and indirect address jumps.

b) Distinct high-level language constructs must be implemented with distinct machine code constructions. For example both **if** and **while** statements need to generate comparison of expression with zero and jumping forward to skip execution if the expression is zero. So, the comparison with zero must be implemented in two different ways. For example, instruction **AND EAX,EAX** will be part of **if** statement; **CMP EAX,0** will be part of **while** statements).

c) Machine code needs to have some redundant information, like variable names, which is described in a separate paragraph.

d) The largest group of the classical programming languages like C, Pascal, Fortran, BASIC, and even some object oriented languages can be designed in a way to satisfy these requirements, but some elements can not be implemented. The obvious example for this is C preprocessor, because it generates very ambiguous code.

e) The microprocessors with insufficient instruction set must have capability to implement some code constructions using subroutines at distinct addresses.

f) The code optimization is limited to set of special cases.

The code-generating phase is well-explored area. It can be realized using recursive descent parsers with addition of code generation. Such algorithms are well described in literature. [3, 4]

4. Additional information and variable names

Machine code and actual data are not sufficient for full recognition of the instructions. There must be some additional information. For example, to rediscover variable, not only variable value needs to be stored in the memory, but also the variable name and probably information about variable type. For each function, one has to store in the variable area the following: function name, address of the procedure code and information about their parameters. The address of the procedure code is updated after every code insertion and deletion. Depending on the microprocessor architecture, might be required (although not necessary) some redundant information like relocation tables, code comments, pointers to every program line etc.

5. Recognizing machine code

When the program is in machine code, the program recognizing and displaying it in ASCII format of the high level language is essentially modified transformation of the expression conversion from postfix to infix form expanded with recognizing of composite statements.

The main task of the recognizing algorithm is to convert from machine code instructions (which calculate expressions in a precedence order) to the standard, infix form, expression representation [8]. After the complete expression is recognized, it is checked if the expression is part of statement or standalone expression (for C like languages).

The recognizing algorithm, for C like languages, takes groups of bytes from the binary executable file (or RAM memory) and works, basically, in the following way:

1. Set output string to empty string. The output string is located inside recognition buffer. The recognition buffer is also the editable buffer, as mentioned in overall architecture.
2. Get sequence of bytes until any known sequence is found, but first search for longer sequences to avoid ambiguities.
3. If the sequence is instruction that stores intermediate results to stack, append the symbol "(" to the output string.
4. If the sequence is code that reads variable value, find the variable name from the variable address (or generate generic name) and append the variable name to the output string.
5. If the sequence is code that sets constant value into the register, convert the constant to ASCII value, and append it to the output string.
6. If the sequence is set of instructions which perform binary arithmetical operation, append the ")" symbol to the output string and then go through output string from the end to the beginning, counting open and closed parentheses. Put the necessary operator in front of "(" symbol which matches just appended ")" symbol.
7. If the sequence is set of instructions which perform unary arithmetical operation, put the required operator in front of whole expression (if there are no commas, and function calls inside output string), or after the last comma, or after the first open bracket which follows function name.
8. If the sequence is the code that allocates stack for the local variables inside function, put some special symbol at the end of output string.
9. If the sequence is the code that puts some value to the stack to be used as function parameter, put the comma symbol to the output string.
10. If the sequence is subroutine call, find the name of the function from the subroutine address, return to the remembered special symbol in step 8 and replace it with function name and "(" symbol. Append the "(" symbol to the output string.
11. If the sequence is part of "**if**", "**while**" and similar statement, append the corresponding keyword and "{" symbol to the output string. Increment bracket level (initially set to zero).
12. If the sequence is end of statements "**if**", "**while**" append the "}" symbol to the output string. Decrement bracket level.
13. If the sequence is expression end, append the ";" symbol to the semicolon; if the bracket level is zero, the complete statement is recognized, go to step 16.
14. If the sequence in steps 2-13 is recognized, go to step 2.
15. In other cases the expression is not recognized. Convert the sequence to hex dump and put it to the output string.
16. Now clean the expressions of the redundant brackets and parentheses.

Many microprocessor architectures (like MIPS RISC) require somewhat more complex recognition algorithm, especially those where arguments are mixed within the same byte as operation code. [1]

6. Memory organization

The executable (binary) program can be, therefore, organized in a way displayed on the following Figure 2.

Executable header
Pointers to other data
Fixed code
Variables
Patch area
Line information
Code area
Executable trailer
Stack , heap, editor buffer

Figure 2: Memory organization

At the beginning of the program, there is a fixed size header of the executable file, defined by the rules of the operating system. It can vary to the great extent. For some operating systems it can be even empty (for example MSDOS COM program, or with the most custom systems) or more complex (Windows PE executables, Unix ELF format). The editor must also update this area, whenever the other areas are changed.

The pointers to other data should be at fixed position inside executable program and they help to discover information such as "Where the actual program is located" or "How many variables there are".

The fixed code area is necessary on the microprocessor architectures without all required instructions. For example, many microprocessors have no floating point, long integer arithmetics nor (essential for this language and translator concept), indirect subroutine calls. These instructions have to be simulated and recognized using calls to subroutines at fixed addresses.

Global variables, unnamed variables, labels and subroutine headers are located in a separate program section. They are organized as chained list. For each named variable there is variable name, value and data type. The insertion of the new variable moves the complete program code and updates all pointers. But, deleting the variable is more difficult problem. The value of the variable is at some absolute address. To ensure correct program execution if the variable is deleted and memory space reclaimed, the complete program needs to be updated in a garbage collection process by converting each statement from machine code to ASCII and back to machine code (but now with new addresses referred to). Alternatively, the patch area may contain all variable references to simplify variable deletion process.

The patch area is optional. It may contain addresses that must be updated whenever the code is moved. Inclusion of this area may speed up the deletion and insertion process, at the price of larger memory requirements.

The statement addresses area is also optional. For each editable statement, there can be start address and statement size. Introduction of this area speeds up listing and execution of the program (due to easy recognition of statement end without introducing redundant instructions), but slows down memory inserting process.

The actual program code is fully relocatable. The complete "if" conditions and loops need to be recognized as single statements, because they contain relative jumps.

The executable trailer is defined by operating system in the same way as executable header.

If the operating system, or hardware architecture, does not allow mixing of code and data in the same sequence of code, the code sections and data sections need to be separated.

7. Example

To show method of the code generation and recognition, we shall present one example. To reduce size of the example, due to the paper size limitation, we shall opt for very simple program format: MS DOS COM file, because it does not have complex headers. The most of the methods developed here, can be applied to any operating system, and many other executable formats, for example Win32 PE, Unix ELF format, and many others. To get more knowledge about 8086 instruction set, see [2].

7.1. Code generation

Assume that a test program in a C like language is

```
int test;
int a;
test=2;
a=5;
while (test<10) {
    test=(test+a)%2;
}
```

After entering first line, the data area of the program contains name of the variable "test" and reserved space for the variable value.

After entering second line, the data area of the program contains names and reserved space for both variables.

Entering third line will take the syntax analysis, recognize the assignment statement, where constant is assigned to the variable. The variable "test" is defined, so the syntax and semantics are correct. Then, the code is generated.

The process for the fourth line is similar, while the fifth to seven lines require a bit more complex syntax analysis, but still put the program code at the right place. Here is a generated COM file (they are always executed from address 100h):

```
E9140016017701020001000074657374
00018D1E6100C38D1E0A018B0753B802
005B8907908D1E12018B0753B805005B
8907908D1E0A018B0750B80A005B39C3
0F9CC025FF0009C07503E929008D1E0A
018B07538D1E0A018B07508D1E12018B
075B03C350B802008BD85899F7FB8BC2
5B8907E9BCFF90
```

7.2. Code recognition

When the listing is requested, the decompiling algorithm first goes through variable area and then prints their names and types. The program can easy to recognize and understand it.

The decompiling module continues with recognition of the instruction codes. For example, at address 117h there is sequence 8D1E0A018B07 (LEA BX,[010Ah] followed by MOV AX,[BX])

The recognized sequence actually means: Reading integer variable located at address 010Ah. Due to special format of variables, the name of variable is stored just after the value (in our case at address 010Ch). So, the recognizer can put the variable name to the output string.

The next recognized sequence is 53 (mnemonic PUSH BX), which is recognized as symbol "(". So, the output string contains 'test ('.

Now, the decompiler encounters B80200 (instruction MOV AX, 2), which is loading accumulator with constant. The constant is 2. The recognition buffer is 'test (2'. The sequence 5B (POP BX) is recognized as closed parenthesis, and the recognition buffer is 'test (2)'. The sequence 8907 (MOV [BX], AX) is recognized as assignment operator. It should be put before open parenthesis we now have in the recognizing buffer '`test=(2)`'. Finally, sequence 90 (NOP) means 'Semicolon' and the recognition buffer contains '`test=(2);`'. With some additional checking it is possible to get rid of redundant brackets, so we have beginning position '`test=2;`'. At this point the recognition buffer is printed on the screen and freed.

The recognition of the while loop in our example is a bit more complex, but it is done in a similar way. The decompiler firstly recognizes '`test<10`', and then discovers sequences that represent beginning of the while loop, so the recognition buffer now looks like '`while (test<10) {`'. Because there are open '{' brackets in recognition buffer, the recognition must continue until the end of loop is recognized. At this point, the recognition buffer contains

```
while(test<10){ test=(test+a)%2}
```

8. The advantages of the approach

Since the program is held in native machine code, expected execution speed should be faster than execution speed of the interpreter based languages.

Due to the instant program translation during editing, translation process will be faster in comparison with classic compiler or pseudo compiler based languages.

Small memory size programmable embedded systems without disks will find it beneficial, since there is no need to store both source and machine code in memory.

Adopting our proposed approach having only one file, the distribution of Open Source applications would be highly simplified and would take much less time to complete

Patching and upgrading of the programs could be done immediately, without waiting for service pack release

Executable program is not dependent on virtual machines, separate run time libraries and interpreter programs, because the program is stored in a native machine code.

If all development versions of the program need to be kept and archived, memory savings become significant even on the systems with large memory resources.

9. Disadvantages of the approach and possible ways to overcome them

Unlike classic compilers, which keep the code in standard ASCII format, keeping code in machine code format is not directly portable from one processor architecture to another. This problem can be solved with a special program which converts executable from one architecture to another (decompile and recompile).

Some language constructions can not be implemented (most notably preprocessors), or at least can not be implemented on some platforms. In this case, these must be replaced with other equivalent platforms.

Eventual change in code generation requires different version of the editor, because the old one will not correctly recognize the generated code.

In the case of code damage, the executable file is less repairable than ASCII file. Backup is obvious solution for this situation.

Team development of large programs requires separation of the program to several dynamic libraries or executables.

10. Areas of possible utilization

The primary areas of utilization of the proposed programming language translator, written in a described way, are on device programmable embedded systems. [11] Many diskless embedded systems still have small memory size. The memory savings of having executable code only, compared with classic style compiling (which requires that source, executable and symbol tables are resident in memory at the same time) or interpreters (which require huge interpreter program in memory during execution), is significant.

Described method of code generation and recognition can improve debugger technology, because it is proven that source code and executable code are not always in synchronization.

The programs generated by this method can be used as virus detectors, because it is easy to recognize if virus changed the program.

Described method of translation is suitable for various small and medium sized applications. The development process will be easier because there is no time lost in waiting for compilation or problems with supplying interpreters.

Applications that require fast and frequent changes at the execution place, like industrial applications, or applications that often require security patching, can be modified in a much easier way.

The open source software distribution will find it beneficial, to supply just one version (executable which is source as well) rather than two separate versions (executable and source).

In computer science courses, the students will better understand relationship between the high level languages and machine code.

11. Implementation

To prove the feasibility and implementation capability of the above-described translator, at the Faculty of Electrical Engineering in Sarajevo, Bosnia, an experimental language translator without source code (for C like language and WIN32 platform) has been developed. The generated code is comparable with the code written with classic style compilers, as it was illustrated with the small example presented earlier in the paper.

12. Conclusion

It was demonstarted to be feasible to design and implement high level language, where the machine code is in the same time representation and contains the source code. The implementation of such all-in-one programming language requires that several presumptions and additional information are included within the generated code. Although this approach has some disadvantages, it is interesting alternative to the classic style compilers, interpreters and pseudo compilers and in certain type of applications can be valuable option.

13. References

[1] Dave Jaggar, *Advanced RISC Machines Architectural Reference Manual,* Prentice Hall, London 1996.

[2] Intel Corporation, *IA-23 Architecture Software Developer's Manual,* Intel Co, MT Prospect, 2002.

[3] Steven Muchnick. *Advanced Compiler Design and Implementation.* Morgan Kaufmann; 1997.

[4] Adnan Kulenović, *Osnove projektovanja kompajlera,* Svjetlost, Sarajevo, 1988.

[5] Eldad Eilam, *Reversing: Secret of reverse engineering,* Wiley, 2005

[6] Christina Cifuentes, *Reverse Compilation Techniques- PhD Dissertation,* Queensland University of Technology, Department of Computing Science, Brisbane, 1994

[7] Mads Sig Ager, Olivier Danvy, Mayer Goldberg, *A Symmetric Approach to Compilation and Decompilation,* BRICS, Department of Computer Science University of Aarhus., Aarhus, 2002

[8] M. N. Bert and L. Petrone. *Decompiling context-free languages from their Polish-like representations. Calcolo,* XIX(1):35-57, March 1982.

[9] Mike Van Emmerik and Trent Waddington: "*Using a Decompiler for Real-World Source Recovery*". *Proceedings of the 11th IEEE Working Conference on Reverse Engineering,* Delft University of Technology, the Netherlands. Delft, pp. 27-36, 2004

[10] Christina Ciufentes, K. John Gough: "*A Methodology for Decompilation*". *Proceedings of the XIX Conferencia Latinoamericana de Informatica,* Buenos Aires, 2-6 August 1993. pp. 257-266

[11] Samir Ribić: "Concept and implementation of the programming language and translator, for embedded systems, based on machine code decompilation and equivalence between source and executable code.". *Proceedings 13th Working Conference on Reverse Engineering (WCRE 2006), 23-27 October 2006, Benevento, Italy,* pp. 307-308