Proceedings of the International Multiconference on
Computer Science and Information Technology, pp. 1037 – 1039
ISSN 1896-7094

Machine Code Can Be Representation of Source Code With Optimization

Samir Ribić, Adnan Salihbegović

Elektrotehnički fakultet, Zmaja od Bosne bb,
71000 Sarajevo, Bosnia and Herzegovina
{samir.ribic,adnan.salihbegovic}@etf.unsa.ba

Abstract. In the recent times the authors of this paper have been doing research on possibility of developing programming language, which would be neither compiler, nor interpreter. The concept is based on holding complete program in native machine code, while the specialized editor can decompile machine code and display it in high level language. The displayed code can be reedited and saved again as pure machine code. This paper investigates the possibility of optimizing generated code, while still retaining decompilability features.

Rather then traditional approach for converting programs from high level languages to machine code using compilers and interpreters, the authors of this paper presented one more approach, in [1] and [2], where the source code does not exist as separate entity from executable (similarly to interpreted languages), while the program is executed in native machine code without aid of additional interpreter (similarity with compiled languages). The role of editor, compiler and decompiler is given to only one program, which displays the high level program representation from native code program. ASCII representation exists only during editing process.

Bidirectional transformation between high level language code and machine code is established by generating the code in such a way that it can always be decompiled. For example, to access the integer variable, in the native code the instruction LEA EBX,[EBP-nnnn] is executed, followed by MOV EAX,[EBX]. Each subroutine has special header, (this header does not slow down program execution), that provides information to editor about positions and data types of the variables on the stack. Hence, during decompilation for the purpose of listing or editing the program from the sequence mentioned, it is possible to restore that it is accessing the variable. From the offset nnnn it is possible to restore the variable name. Bidirectionally recognizable sequences are simple, but they are far from optimal when combined with larger language sentences. For example, the statement vb=va+2; in high level notation is represented in machine code as

538D1DNNNNNNNN8B03538D1DMMMMMMMM
8B0350B8020000005B03C35B890390

where NNNNNNNN and MMMMMMMM represent memory addresses of the global variables vb and va. In assembly code these instructions are

```
lea ebx,[vb]/ mov eax,[ebx] /push ebx/ lea ebx,[va] / mov eax,[ebx] /
push eax/ mov eax,2/ pop ebx /add eax,ebx/ pop ebx / mov [ebx],eax.
```

Such a long sequence results from translation symbol by symbol. When access to the variable is recognized, for addition it is enough to know value of the variable, for the assignment only the address is required, while pointer operator (*) needs both. Until we reach next symbol, the worst case sequence is generated. Another reason for longer code than necessary, is in requirement that functionally the same, but logically different sequences (like testing expression value in **if** and **while** statements) need to be represented differently in machine code, to make decompilation easier, even if this means a longer sequence than necessary.

One solution to this can be with the table which would contain sequences in shorter and expanded form, with the modification of the compiler part. In our example, the shorter sequence is `A1NNNNNNNN05020000008905MMMMMMMM`, meaning `mov eax,[va] / add eax,2 / mov [vb],eax.` Before the code is generated, it is first required to look into the table to find a shorter sequence equivalent to the originally planned generated sequence. If such a sequence exists, the compiler part of the editor produces shorter sequence. On the other hand, during code recognition, the decompiler part will first check if the sequence exists in the table, and if it does, it will repeat the recognition over larger version and therefore find the decompiled version. The most important sequences to be optimized are: assigning constant or variable to variable, arithmetic operations between two variable or variable and constant, negative constants, unary operations over variables, accessing to the array element, comparison of variable and constant (or two variables) in a condition part of **if**, **while**, **for** and **do/while** statements, comparison and assignments of two array elements etc.

Another problem is need for changing the editor itself if new optimization sequences are added. This problem can be solved by holding optimization sequences in a separate file (for example Windows.DLL, or Linux.so) which keeps track not only on optimized and non-optimized sequences, but also about versions of each translation. The version of the optimizer is saved somewhere inside user executable. For embedded systems which contain the compiling/decompiling editor in ROM, this table is an integral part of the editor.

References

1. 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
2. Samir Ribić, Adnan Salihbegović: *High Level Language Translator With Machine Code As Representation Of The Source Code,* Proceedings 29th International Conference on Information Technology Interfaces (ITI 2006), 25-28 June 2007, Cavtat, Croatia. pp. 777-782
3. Eldad Eilam, *Reversing: Secret of reverse engineering,* Wiley, 2005.