Concept and implementation of the programming language and translator, for embedded systems, based on machine code decompilation and equivalence between source and executable code

Samir Ribić
Elektrotehnički fakultet, Sarajevo
megaribi@epn.ba

Advisor: dr Adnan Salihbegović, Elektrotehnički fakultet, Sarajevo

Abstract

In this thesis it will be investigated the possibility of developing the programming language translator, heavily based on decompilation. Instead of keeping program in source code, it will be kept in native machine code, but it will be transparently visible as high level language program, with the help of the specialized editor.

1. Introduction

The most of the programming language translators are realized in a form of pure compilers, pseudo compilers, interpreters and hybrid translators.

The pure compilers translate from the high level languages to machine or assembly language. Compared with all of the mentioned approaches, this concept generates the fastest executable code [3], and therefore it is most often used on large computer systems. This concept is suitable for embedded systems too, if the translation is performed on application developing computers, but it is not the most suitable method for translating on the system where the program is executed. This is because of the memory requirements for both source and executable code (memory on many embedded systems is limited) and compilation time.

The interpreters are slower than compilers, but they do not make difference between source and object code. They analyze the source code, and based on the source code they call different subroutines. They do not require translation of the whole program even if just one line of the code is changed. The disadvantage of the concept is slower work and the requirement for the routine library and complete interpreter in memory.

Pseudo compilers have similar behavior like compilers. They translate high level language to pseudo machine code. This code requires special interpreter, and both source and pseudo code.

Alternatively there are just in time compilers which internally convert pseudo machine code to native code.

Hybrid translators are designed in several ways. The most known is translator for programming language FORTH. This language introduced translations of subroutines (called words) just after they are entered to the compact code called Indirect threaded code [7], from which the source code can be reconstructed. But the format of the code is not native machine code, it is usually interpreted. Furthermore, FORTH is very hard to understand, due to reverse Polish notation concept.

The opposite process to compilation is decompilation, i.e. translation form machine code or pseudo machine code to high level language. This area is far less developed than compilation area. The success of the process varies depending on computer architecture, source language, format of the translated code, optimization level etc. Although less developed than compilation, the decompilation has growing success in recent years. [4]

2. The proposed approach

The compilation, editing and decompilation, although regarded as related [6] are usually different processes. Decompilation in the most of implementations is not an integral part of the translator, and even if it is part of the translator, usually it is not decompilation of machine code, but pseudo machine code or some other compact format.

This thesis is researching the possibilities of realization of realizing of the high-level language translator on a quite uncommon basis, which will unite some good properties of the interpreted and compiled languages. These properties are: unique code in main or external memory (without separation to source and object part), avoiding need for translation of the complete program, execution speed of the native machine code, and independence from interpreters and included libraries. The programming language needs to...
have standard elements of the procedural programming languages (algebra notation expressions with brackets and priorities, conditions, arrays, loops, subroutines, different data types)

The specialized editor takes the role of classic compiler or interpreter. This editor does not only care about the programing code being entered, but also about program's memory interpretation. Every entered line is incrementally translated to the machine code, and high level ASCII representation is removed from memory. However, the machine code is written in such way that it is possible to recover the statement in the form very close to the original. When the user wants to edit or list the line of the program, it will be decompiled and shown on the screen as a line in high level language. The editor is not required to be in memory during program execution.

3. Comparison with other approaches

The difference of the proposed approach, compared with the existing ones can be shown in the Table 1:

<table>
<thead>
<tr>
<th>Approach</th>
<th>ASCII source code exists as separate entity</th>
<th>Intermediate code exists as separate entity</th>
<th>Machine code exists as separate entity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compiler</td>
<td>Yes</td>
<td>No</td>
<td>yes</td>
</tr>
<tr>
<td>Compiler w. intermediate code</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Source level interpreter</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Pseudo compiler</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Compact interpreter, hybrids</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Just in time compilers</td>
<td>No (at least near final user)</td>
<td>Yes (although often unseen)</td>
<td></td>
</tr>
<tr>
<td>Proposed approach</td>
<td>No (but it is transparently visible)</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Table 1: Different approaches comparison

It can be seen from the table, that this approach differs from other existing approaches because it does not use neither intermediate code nor ASCII based source code, but quick, native machine code for data and program representation. However, the displayed and editable representation of the program code is in easily understandable, classic, high level language.

4. Research areas

The issues investigated in the thesis are:
- Expression generating and recognizing priority
- Code optimization, and problems of ambiguity
- Implementation of some programming languages, the simple custom language for very small controllers and language close to ANSI C
- Syntax and semantics
- Memory organization of the programs
- Local and global variable naming
- Relocatability and code insertion and deletion
- Patching of executables on some OS-es

5. Areas of usage and evaluation

While this approach has some disadvantages, (poor portability is the most serious one), it has some good properties of both interpreted and compiled languages. It is expected to be useful on low memory computer systems where the savings gained with not keeping both source and object code in the memory is significant advantage. Other possible uses of this approach are for easier maintenance of the programs that need to be to be frequently patched and updated, and for open source applications in a situation where maintaining of both source and binary versions are met with various kinds of difficulties.

The approach will be tested by real life and concrete implementation of the editor-translator on different platforms [1] [2] and compared with classic approaches in terms of execution speed, memory consumption and compilation speed, for different small and medium sized testing cases.

6. References