A Study on the Code Optimizer Generator for the Smart Cross Platform

Yunsik Son*, YangSun Lee**

*Dept. of Computer Engineering, Dongguk University
26 3-Ga Phil-Dong, Jung-Gu, Seoul 100-715, KOREA
sonbug@dongguk.edu

**Dept. of Computer Engineering, Seokyeong University
16-1 Jungneung-Dong, Sungbuk-Ku, Seoul 136-704, KOREA
Corresponding Author : yslee@skuniv.ac.kr

Abstract. Every platform used in the development of smart phone contents has a unique development environment. The need for platform-specific methods and languages entails extra development costs when content is created for a number of platforms. The Smart Cross Platform is the virtual machine solution that supports various programming languages and platforms, and its aims are to support programming languages and smartphone platforms such as Android and iOS. Various contents that developed by supported language on the platform can be executed on Android and iOS platforms at no additional cost. But, can assume that optimization of executed code is very important because act thing which VM is slow than that process by real processor. In this paper, we deal with code optimizer generating techniques to automatically generate an optimizer for virtual machine.

Keywords: Smart Cross Platform, Virtual Machine, Intermediate Language, Optimization, Code Optimizer, Code Optimizer Generator.

1 Introduction

Content development environments for existing smart phones require the generation of object code specific to the target machine or platform, while the development language used is different for each platform. Therefore, if the same content is to be used over a range of different objective machines, a different compiler is needed for each machine. This makes the content development process very inefficient. Smart Virtual Machine (SVM) is a virtual machine solution that aims to resolve such problems by making use of the Smart Intermediate Language (SIL) designed by our research team. But, the VM based execution method rise a problem that is too slowly execution performance compare with native execution method.

In this study, we deal with a code optimizer generator to enable that automatically generating code optimizer for smart virtual machine(SVM) in SCP.


2 Related Studies

2.1 Smart Cross Platform

In existing smart phone contents development environments, different object codes should be presented for each target device or platform. Development languages that can be developed are also varied depending on the platforms. The smart cross platform was developed based on a virtual machine by the research team of this study to address this problem. The smart cross platform is based on a virtual machine that uses a platform-independent medium language with multiple smart devices executed regardless of the platform [1]. SIL (Smart Intermediate Language), the intermediate language, accommodates both procedural C and object-oriented C++/Java.

The smart cross platform consists mainly of the three sections: compiler, assembler, and virtual machine. The hierarchical structure minimizes the loads of retargeting to other devices and operation systems. The figure shows a model of the smart cross platform [2].

![Fig. 1. System model of Smart Cross Platform](image)

2.2 SIL

SIL, the virtual machine code for SVM, is designed as a standardized virtual machine code model for ordinary smart phones and embedded systems. SIL is a stack based command set which holds independence as a language, hardware and a platform. In order to accommodate a variety of programming languages, SIL is defined based on the analysis of existing virtual machine codes such as bytecode, .NET IL and etc. In addition, it also has the set of arithmetic operations codes to accommodate object-oriented languages and successive languages [2, 3].

SIL is composed of meta-code (shows class declarations and specific operations) and arithmetic codes (responds to actual commands). Arithmetic codes are not subordinate to any specific hardware or source languages and thus have an abstract
form. In order to make debugging of the languages such as the assembly language simple, they apply a name rule with consistency and define the language in mnemonics, for higher readability. In addition, they have short form arithmetic operations for optimization. SIL’s arithmetic codes are classified into seven and each category has its own detailed categories.

2.3 Comparison of the VM / Native Execution Methods

Aspects of executing contents, there are differences between the VM method and the Native. Firstly, the VM method can easily executing the same VM application even if H/W changed because it has the H / W independence. Also, it has the stability for the target system when the errors exist in the VM contents and can be easily ported to various H/W. On the other hand, the execution performance of the algorithm code is slower than the native method. In the case of the native method, need the changes of the contents by the characteristics of H/W platforms and OS, and it can affect the system due to the errors in the contents. But, the executing performance of the algorithm code is faster than the VM method [2].

3 Code Optimizer Generator

In this paper, we deal with the COG (Code Optimizer Generator) model shown in Fig. 2, to generate code optimizer automatically from the given optimization pattern description. By the COG, the code optimization method can be more easily apply than the manually implementation when the newly optimization patterns are added.

![Diagram](image)

**Fig. 2.** Code Optimizer Generator Model

The code optimizer generator receives pattern description as input and writes a DFA (Deterministic Finite Automata) optimization table as Fig. 2 by way of procedures displayed in Fig. 3.
The code optimizer generator consists of a scanner, parser, gathering information, calculating C0, making DFA and writing table. The scanner is the compiler’s first procedure, and it creates tokens, syntactically significant minimum units. In other words, it receives a pattern description as an input and generates a series of tokens. The parser receives the tokens output from the scanner, checks for errors and creates sentences according to the program syntax to generate an AST (Abstract Syntax Tree). The “gathering information” stage collects information as the pattern description content is traversed into a tree structure through the scanner and the parser. The data structure is a two-dimensional layout with an integrated linked list. It is converted into a data structure where the pattern information and LOOKAHEAD are integrated to calculate C0. The “make DFA” stage turns the calculated C0 into a DFA format. The DFA normalization process goes through repetitions of “calculating C0” and “making DFA” stages. Since the resultant DFA is normalized, optimization actions are easily performed. In other words, DFA-type backtracking task is not necessary.

4 Pattern Description for Code Optimizer Generator

Description of optimization pattern for pattern description consists of replace and pattern parts in the BNF style shown in Table 1. There are no length restrictions for the replace and pattern parts, and a single pattern comprises a single line.
Table 1. Pattern Description Format (BNF Style)

<table>
<thead>
<tr>
<th>pattern_description</th>
<th>-&gt; patterns =&gt; DESCRIPTION;</th>
</tr>
</thead>
<tbody>
<tr>
<td>patterns</td>
<td>-&gt; pattern;</td>
</tr>
<tr>
<td></td>
<td>-&gt; pattern patterns;</td>
</tr>
<tr>
<td>pattern</td>
<td>-&gt; instr_code_sets</td>
</tr>
<tr>
<td></td>
<td>':=' instr_code_sets</td>
</tr>
<tr>
<td></td>
<td>'%%' '%eval_methods' '%%' =&gt; PATTERN;</td>
</tr>
<tr>
<td>instr_code_sets</td>
<td>-&gt; instr_code_set;</td>
</tr>
<tr>
<td></td>
<td>-&gt; instr_code_set instr_code_sets;</td>
</tr>
<tr>
<td>instr_code_set</td>
<td>-&gt; '%instruction' opt_parameters '/' =&gt; INSTR_CODE_SET;</td>
</tr>
<tr>
<td>opt_parameters</td>
<td>-&gt; ;</td>
</tr>
<tr>
<td></td>
<td>-&gt; parameters;</td>
</tr>
<tr>
<td>parameters</td>
<td>-&gt; '%param';</td>
</tr>
<tr>
<td></td>
<td>-&gt; '%param' parameters;</td>
</tr>
</tbody>
</table>

In order to distinguish instruction code sets, replace parts and pattern parts, pattern descriptions are described with the separator "::=". The pattern part’s instruction code sets are described. The instruction code sets are described with instructions, parameters and the separator "/".

Following Table 2 shows the example of the pattern description.

Table 2. Pattern Description Example for COG

SYNTAX SIL_pattern_description

```
SIL_pattern_description -> patterns => PATTERN;
patterns -> addiv;
    -> subiv;
    -> muliv;
    -> diviv;
...  
addiv -> 'lod.i' 'lod.i' 'add.i';
subiv -> 'lod.i' 'lod.i' 'sub.i';
muliv -> 'lod.i' 'lod.i' 'mul.i';
diviv -> 'lod.i' 'lod.i' 'div.i';
...  
```

Next Table 3 shows the generated DFA optimization table for code optimizer using optimization pattern in Table 2.
Table 3. Generated DFA table for code optimizer using patterns in Table 2.

<table>
<thead>
<tr>
<th>State</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0, 0, 0, 0, 7, 0, 0, 6, 0, 5, 4</td>
<td>3, 2, 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>-3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
<td>0, 0, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
<td>0, 0, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>-4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0</td>
<td>0, 0, 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5 Conclusions

A virtual machine has the characteristic of enabling application programs to be used without alteration even if processors or operating systems are changed. It is a core technology for executing a variety of contents in the recent mobile, embedded and smart systems. The virtual machine solution has the advantage on the portability, but it has the low performance due to the S/W interpretation for the instructions.

In this paper, we examined a code optimizer generator in order to automatically generate code optimizer for the SVM. By the COG, the code optimization method can be more easily apply than the manually implementation when the newly optimization patterns are added. In the future, we will evaluate performances of the proposed COG (Code Optimizer Generator).

Acknowledgments. This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT and Future Planning (No.2013R1A2A2A01067205).

References