Using Local Library Function in Binary Translation

Abstract: In order to improve the execution speed of a binary translation system, this paper proposes a method to use local available library function codes called Jecket, which uses the executable file’s symbol Table s and program linking Table s to substitute local available codes for library functions in the translation stage, thus eliminating translation overhead and reducing a lot of generated codes. Firstly, this paper analyzes the argument lists and returned values of library functions which need localizing, then analyzes library function’s names and load addresses of library files which dynamically linking executable files depended on. Secondly, directly translates all invocation library function’s instructions to code blocks of parameters parsing, local invocation instruction and return value restoration. Finally, when executing the generated codes, local functions will be directly called to complete the computation. The experiment based on QEMU and nbench benchmarks shows that after using Jecket algorithm, the speedup is up to 20.9 times.

Keywords: binary translation; QEMU; library function

1 Introduction

Binary translation [1] has been widely used in software security analysis [2], software reverse engineering, system virtualization and other areas, and has been an essential technology of software transportation. Dynamic binary translation is a just-in-time compiler [3-5], which dynamically generates the required codes when running the target programs, it can detect code detection and locate better [6]. Some techniques improve the efficiency of dynamic binary translation such as hot-path optimization [7], register mapping [8], multi-thread optimization [5] and so on, but these techniques can't increase the low efficiency significantly.

Because of program locality, only 20% of the codes can take up 80% of the execution time [9]. The quality of codes generated by binary translation is the most important factor of the binary translation’s efficiency. Traditional optimization techniques focus on optimizing basic blocks from the intermediate code layer to target instruction layer, while ignoring the shared codes between different platforms [10]. In this paper, in order to take full advantage of the characteristic that the current
executable files use a dynamic linking library, to replace the dynamic link library shared code, thus eliminating translation overhead and reducing a lot of generated codes, system efficiency will be improved.

Applications heavily use dynamic libraries for application upgrades and maintenance under Linux, due to the dynamic library's position-independent code (PIC), which can be loaded into any address, the text segments of a dynamic library can be shared in multiple processes without relocation, thus improving the system performance. A dynamic library function which is localized means that the replace the dynamic library function on source machine with that on target machine. In this way, due to using the local optimized library functions directly, without translating the source machine library functions, it can avoid the particularity of dynamic library function implemented on different systems, on the other hand, for applications which are used frequently it can reduce the cost of translation.

The paper first presents the upper bound of code optimization using a formal method, then proposes a Jecket method to use local libraries and proves that the method on translated code blocks has reached the optimization upper bound in theory. Finally, implements the algorithm in the popular dynamic binary translator QEMU, and use nbench benchmarks to verify that the Jacket algorithm can improve operational efficiency significantly.

2 Related Work

Jens presented a formal model of dynamic binary translation [11], firstly, Jens presented a formal representation which is closer to real machines according to Turing machine, then, gave the principle of binary translation, finally presented the representation of the code optimization upper boundary. Some important theories presented as follows:

Let \( M(S, I, \gamma) \) be a machine, where
\( S \) denotes the set of states of the machine
\( I \) denotes the set of machine instructions
\( \gamma: I \times S \rightarrow S \) is the interpretation function for machine instructions over a machine state.

Then let \( Ms(S, I, \gamma) \) be the emulated or source machine, and let \( Mt(S, I, \gamma) \) be the host machine, based on this, the binary translation may be expressed as:

Searching for a map \( \varphi \) so that the source platform to interpret the instruction \( i \) in current state \( s \), there is always \( i' = \varphi(i) \) executed in a state \( s' \) corresponding to the target platform. The map \( \varphi \) is one of the core tasks in binary translation.

Since the map \( \varphi \) is not unique, in order to get optimal form, the mapping function \( \varphi \) is simplified by various intermediate representation optimization methods. For Yirr-Ma system, Jens has proven its optimal form of code optimization including
machine state mapping and instructions mapping. Jens has also illustrated that it can increase the cost by 290% in optimal conditions.

Jens is committed to optimizing the mapping of machine status and translating the instruction blocks, while ignoring the simulation from the semantic layer, resulting that optimization cannot be mapped from semantic blocks of a higher semantic level. From the perspective of the program equivalent transformation, this paper gives a binary translation measurable upper bound of the optimal, as to guide the engineering applications on the theory and practice.

3 Binary translation equivalence and optimal upper bound

Program equivalent transformation means on the premise of the program function unchanged, to make a transformation of the program’s structure, execution mode, memory mode, program mode. The purpose is to improve the implementation program’s execution efficiency, security and portability, etc., including transform from low-level language to low-level language, such as binary translation, performance tuning of math library; transform from high-level language to low-level language, such as the typical high-level language compiler; transform from serial programs to parallel programs, such as high-performance platform software parallel migration. Program transformation equivalence, can only determine the equivalence of program transformation functionality semantics on the given system, there is no uniform definition of equivalence now. Binary translation is a command-level program equivalent transformation.

3.1 Definition of Binary translation equivalence

According to Jens’s binary translation model, combine with the program transformation correlation theory, then give out the equivalence definition of binary translation in instruction level and semantic level.

The source platform $Ms$ interprets the instruction $i$ in current state $s$, corresponding to the target platform $Mt$ executed $i' = \phi(i)$ in a state $s'$.

**Definition 1.** Instruction-level equivalent.
When $\gamma(s, i) = \gamma'(\phi(s), \phi(i)) = \gamma'(s', \phi(i))$ is satisfied, binary translation instruction mapping and state mapping are equivalent.

This definition describes the case of strong equivalence in instruction-level simulation, is the strong equivalence mode of program equivalent. In some binary translators, interpreting instructions for source platform instruction by instruction meets the requirement of instruction-level equivalent. However, this strong equivalence
definition limits the level of optimization functions, which can only be optimized for each instruction, but cannot be optimized according to the instructions of basic blocks or function semantic.

**Definition 2.** Semantic-level equivalent.
For any instruction sequence \( t = \{ i_1, i_2, i_3, \ldots \} \), when \( \gamma(s, t) = \gamma'(\varphi(s), \varphi(t)) = \gamma'(s', t') \) is satisfied, binary translation instruction mapping and state mapping are equivalent.

Semantic-level equivalent requires that the target platform \( Mt \) implements the translated target platform instruction sequence \( \varphi(t) \) in the initial state, then gets the final state corresponding to state which implemented \( t \) on source platform. It doesn't require a corresponding intermediate state after each instruction implemented, so the optimization which based on the basic block, super block and function-level can be implement.

Equivalence definition is similar to the mathematical concept of isomorphism, it can reflect the consistency on computability of source and target platforms. According to the definition of equivalence, then give the upper bound of binary translation optimization.

### 3.2 The upper bound of binary translation optimization

Binary translation optimization is to simplify mapping function \( \varphi \) under the premise of program equivalence, so as to use target instructions as fewer as possible to denote the instructions of source platform. Let \( |t| \) denote the instruction number of instruction sequence \( t \), \( \psi \) represents optimization function, if \( |\psi(\varphi(t))| < |\varphi(t)| \), then the optimization function is effective to mapping function \( \varphi \).

**Theorem:** suppose that the source file \( B \) generates executable file \( E_s \) by compiling, then executable file, which is compiler-generated from \( B \) for the target platform instruction set, is the upper bound of binary translation optimization, namely, \( |\psi(\varphi(E_s))| \geq |E_t| \).

**Proving:** let \( \delta_s \) and \( \delta_t \) denote the compilation process of platform \( Ms \) and \( Mt \), respectively, \( E_s = \delta_s(B) \), \( E_t = \delta_t(B) \), then executable file \( E_s \) that generated by binary translation can be expressed as \( E_s = \varphi(E_s) = \psi(\varphi(\delta_s(B))) \).

So, the process of binary translation is the one which program transforms from source file \( B \) to executable file of \( Mt \) platform, similarly, \( \delta_t \) is a transformation of \( B \) to \( E_t \), and \( \psi \times \varphi \times \delta_s \subset \delta_t \).

Binary translation process is a special case of the compilation processes, so the upper bound of binary translation optimization is a direct compilation process, that is \( |\psi(\varphi(E_s))| \geq |E_t| \).
The theorem is obvious usually due to the complexity of binary translation system and the particularity of different platforms, the efficiency of binary translation is 1/5 of the local codes, or even lower. So, taking full advantage of the local code can greatly improve the efficiency of binary translation.

4 The Jecket arithmetic

This paper uses the Jecket algorithm to encapsulate a local library function, simulate the source platform parameter and return rules in the target platform, achieve the function that call the local library functions in the runtime of target library files. Mainly three stages in the following are included: library function identification, instruction translation of call library functions, local library function call executes.

4.1 Library function identification

In order to support dynamic linking, ELF dynamic link libraries and executable programs which use dynamic link libraries have the procedure linkage Table s (PTL), PLT Table increases an indirect memory access for function call, that jumps to PLT Table first during the function call to obtain the entry address of function, then jumps to the real function entry from PLT Table.

The function call is implemented by specific instruction. When calling a dynamic library, the branch target address of instruction which function called is the corresponding PLT entry address of function, meanwhile, it is also the corresponding offset of relocation Table entry. Therefore, traverse the relocation Table to obtain all the corresponding offset of relocation Table entries, then scan the dynamic symbol Table and dynamic string Table to obtain the function name.

Therefore, when the binary translation system loads binary codes of source machine, the binary codes to obtain function names of dynamic library functions and PLT entry address can be analyzed. Then select the functions need to be replaced, in order to establish the hash Table using PLT entry addresses for the index. When translation functions call the instructions, browse the hash Table by branch target addresses of instructions. Checking the hash Table will be able to obtain the name of pre -selected functions. The identification of dynamic library function is completed.

According to the.sym Table s and.plt Table s of executable files, the correspondence between the call address of instructions and the name of library functions can be analyzed. Figure 1 shows the algorithmic procedure to obtain the information of functions through the executable files symbol Table s.
function: load executable file, identify function information of dynamic link library
input: the symbol header of executable file (sym)
output: library function information list (func_array)
For each elf_sym in sym
  extract sym’s name, size, type
  if the type is STF_FUNC, the name contains libc, the size of non-zero, add sym to symbol array of library function(func_array)
End for

Figure 1. Algorithm of extracting library function information

4.2 Instruction translation of library function call

Generally in the translation stage, if functions call the dynamic-link library functions, firstly extract all parameters with parameter passing rules of source platform according to the parameter lists of library functions, then use the extracted parameters to call library functions corresponding to target platform. After function returns, obtain the return value of the function, and return the return value to corresponding registers or variables in accordance with the rules of source platform.

Figure 2 shows the general instruction translation algorithm of calling library functions. When localizing library functions, the information of library functions that needs to be obtained in advance. Since the library functions are open and the documents are very abundance, it is very convenient to obtain information of such functions. The algorithm aims at the condition that source instructions is the function call instructions, when target functions that called in func_array, the parameter passing instructions and call instructions are generated according to the function parameters and returned value of func_array, the function returned processing instructions are generated finally.

Special library function call requires complex processing mechanism, when the library function parameters that called are structure pointers, the realization of source and destination platform structure is not exactly the same, such as the data type of the field is different. Apply for structure place of target platform before calling the target library function. After the function returns, store the value of temporary structures in position that corresponding to source platform structures.
function: translate library function invocation instructions
input: codes of executable files (.text), library function list and current execution environment env
output: corresponding instruction block that invoke target library function.
For each insn in text
switch insn
  case: call insn
    if the destination address of call instruction is func_array
    generate the local parameter passing instructions
    generate the local library function invocation instructions
    generate the library function return value storage instructions
    break;
  case …
  default:
    …
    break
End for

Figure 2. Algorithm of translating function invocation instructions

4.3 Execute the local library calls

In order to realize conveniently and debugging facilitate, calling the library functions during executing, just like use the helper function mechanism of QEMU. The helper function extracts and calculates the parameters, writes the results into the position corresponding to source platform CPUState. Figure 3 shows the helper code example that call of the sin function.

```c
void helper_sin(CPUX86State *env) {
    double * doublep = (double *)(&env->xmm_regs[0]) ;
    *doublep = sin(*doublep) ;
}
```

Figure 3. An code example of helper to call sin library function

Because of the complexity and diversity of library functions, some issues need to be solved during library function encapsulation such as parameter obtaining, return value obtaining, format string analysis and so on. Library function encapsulation is a challenge. In the case of calling library functions several times by source executable programs, library functions localize can improve the operating efficiency significantly.
5 Experiments

The performance test of Jecket is provided below, and compared with QEMU. Using nbench tests to test the performance of CPU and memory system that Jecket algorithm implemented before and after in qemu-1.7.2 version.

Let $T_{QEMU}$ and $T_{SQEMU}$ are the execution times of executable programs for QEMU and SQEMU respectively, let $S_{\text{speedup}}$ be the speedup of SQEMU versus QEMU.

$$S_{\text{speedup}} = \frac{1}{\frac{T_{SQEMU}}{T_{QEMU}}} = \frac{T_{QEMU}}{T_{SQEMU}}$$

5.1 Experimental environment

The purpose of experiment is to verify the correctness and measure the efficiency of SQEMU, nbench tests have correctness verification module. Table 1 shows the experimental environment, Table 2 shows the test cases.

Table 1. Binary translation environments

<table>
<thead>
<tr>
<th></th>
<th>Ms</th>
<th>Mt</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS</td>
<td>Fedora 2.6.27.5-117.fc10.</td>
<td>NeoKylin 3.8.0</td>
</tr>
<tr>
<td></td>
<td>i686</td>
<td></td>
</tr>
<tr>
<td>CPU</td>
<td>Intel(R) Core(TM)2 Quad CPU</td>
<td>Q9500 @ 2.83GHz</td>
</tr>
<tr>
<td></td>
<td>Domestic processor</td>
<td></td>
</tr>
<tr>
<td>Compiler</td>
<td>gcc-4.3.2</td>
<td>gcc-4.5.3</td>
</tr>
</tbody>
</table>

Table 2. Nbench-2.2.3 test cases

<table>
<thead>
<tr>
<th>test case</th>
<th>tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>NUMERIC_SORT</td>
<td>Sorts an array of long integers</td>
</tr>
<tr>
<td>STRING_SORT</td>
<td>Sorts an array of strings of arbitrary length</td>
</tr>
<tr>
<td>BITFIELD</td>
<td>Executes a variety of bit manipulation functions</td>
</tr>
<tr>
<td>FP_EMULATION</td>
<td>A small software floating-point package</td>
</tr>
<tr>
<td>FOURIER</td>
<td>A numerical analysis routine for calculating series approximations of waveforms</td>
</tr>
<tr>
<td>ASSIGNMENT</td>
<td>A well-known task allocation algorithm</td>
</tr>
<tr>
<td>IDEA</td>
<td>A text and graphics compression algorithm</td>
</tr>
<tr>
<td>HUFFMAN</td>
<td>A relatively new block cipher algorithm</td>
</tr>
<tr>
<td>NUEURAL_NET</td>
<td>A small but functional back-propagation network simulator</td>
</tr>
<tr>
<td>LU DECOMPOSITION</td>
<td>A robust algorithm for solving linear equations</td>
</tr>
</tbody>
</table>
The instruction count of generated codes is reduced 27.73% according to the statistics of nbench test programs. As it shows from Figure 4, after using the Jecket algorithm, the speedup on nbench test programs is up to 60-70 times. NEURAL NET uses library function exp frequently, after Jacket package, calls the local library functions directly, in this way, it reduces a large number of code generation and translation time and improves the execution efficiency greatly. For the tests that don't use local library functions such as NUMERIC SORT, BITFIELD, FP EMULATION and so on, function localization does not affect its execution efficiency. The test result shows that for applications which contain library function calls, Jecket algorithm makes greatly speedup.

![Figure 4. Speedup on nbench of Jecket to QEMU](image)

### 6 Conclusion

This paper firstly gives a formal representation of binary translation as a way of program transformation, then gives a upper bound of binary translation optimization, discusses solutions approaching to the upper bound of optimization and proposes a Jecket algorithm that uses local library function to replace the translation processes and execution codes, and proves that it can achieves the upper bound of optimization by using Jecket library functions. Finally, this paper implements the Jecket algorithm by QEMU binary translation system and tests the acceleration of nbench tests with Jecket algorithm. The experiment shows that for the applications whose core areas contain the library functions, the algorithm can increase system efficiency significantly.

**Acknowledgment:** This work is supported by National Natural Science Foundation of China under Grant No. 61472447, and also supported by the National High Technology Research and Development Program of China (863 Program) No. 2009AA012201.
References


