A New implementation of binary-translator in the L0 system.

GAN Bo HKUST

Abstract: In the L0 system[1], binary-translator is an important module which is responsible for translating i0 instructions to x86_64 instructions for direct execution. The new implementation provide efficiency and correctness in complicated situations and carefully manages the resources to prevent memory problems.

1. Introduction:

The Layer-Zero system is designed to provide run-time support for the DVM [1] cloud computing system. To provide high efficiency, special sematic and platform independency, a new instruction set, i0 [2], is used to programming in the Layer-Zero system. The functionality of the binary-translator module is to speed up the execution of i0 program by performing binary-translation from the i0 instructions to the target native instructions for direct execution during run-time. In the current implementation, our target platform is the x86_64 (amd64) platform, hence the native instructions translated will be the x86_64 instructions.

2. Overview of the system:

In Layer-Zero system, the 'VPC' process encapsulates the execution of i0 program. The binary-translator is a module within the 'VPC' process. The 'VPC' process will load the i0 executable image to a fixed memory address for translation. In the 'VPC' process, execution will take turns among the service module, the binary-translator module and the native code translated by the binary-translator. The detail is shown as follows:

When the scheduler tries to schedule a new runner [1], the service module in 'VPC' will be given a memory address in the i0 executable area as 'entry point'. Execution will then be passed to binary-translator (shown as ***a*** in Figure 1) which tries to translate from the 'entry point' and generate native code. Then execution will continue to be passed to the native code (shown as ***b*** in the Figure 1). During the execution of native code, the execution will be transferred back to binary-translator when there is a branch to the i0 code that has not been translated yet (shown as ***c*** in Figure 1) or to the service module when there is special branches including runner spawn, runner commit or trap/ syscall (shown as ***d*** in Figure 1). In the second case, the type of the branch instruction translated will determine whether the execution will be transferred back to native code (see i0 specification for detail [2]) (shown as ***e*** in graph 1).

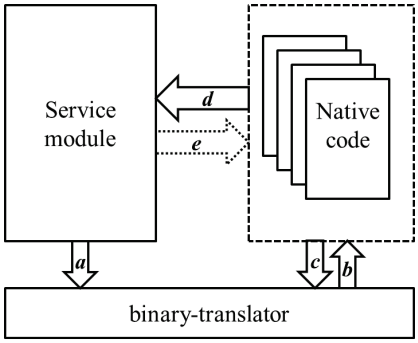
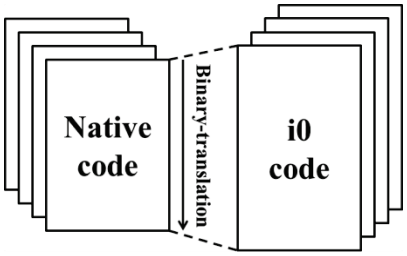


Figure 1

3. Translation process:

The main policy of the binary-translator is: (further explain below)

- Traverse i0 instructions by linear scan.
- Generate native (x86_64) instructions in blocks.
- Reuse the translated code as much as possible.
- Merge blocks when suitable.



After the i0 executable is loaded into the memory space of 'VPC', the i0 image will appears to be one or several code sections and data sections located at different areas in the memory. This property is implied by the nature of i0 executable format, ELF format (currently a much simpler format is implemented) which guarantees that i0 instructions to be translated will actually be in blocks. Taking advantage of that, binary-translator translates i0 instructions to x86_64 instructions in blocks for efficiency. Given a block of consecutive i0 instructions, generating native code by block means to translate the i0 code block to native code block which has the identical effect after execution. It is done by scanning the i0 code block linearly and mapping each i0 instruction to one or several x86_64 instructions which have the same execution effect.

3.1 The 'static' instruction mapping rule:

Binary-translator does a straight approach by making the translation process context independently which ensures that every i0 instruction will be translated statically (regardless of the run-time environment and any other instructions in program order). That is to say, the mapping between i0 instruction and the translated native instruction is defined before run-time. If the binary-translator is given an i0 address to translate, and that i0 instruction is covered in the previous translated i0 code block, then binary-translator can easily locate the corresponding native instruction in the middle of the native block by doing a quick 'fake' translate. (Figure 3.1) After native address is located, the execution can be directly passed to the native instruction and still have identical execution effect.

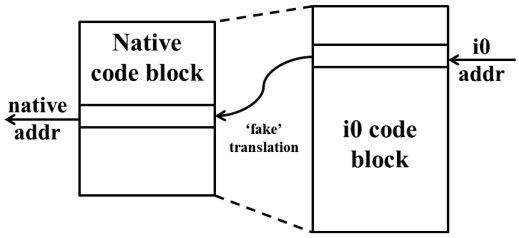
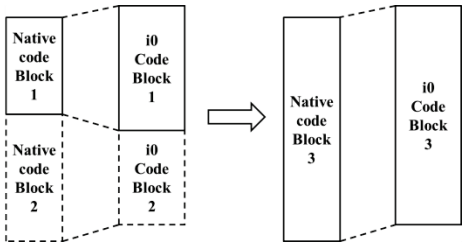


Figure 3.1

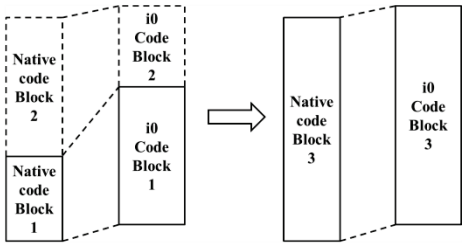
3.2 Merge condition:

As the translated block begins to accumulate, there is possibility that some native code blocks can be merged to large blocks to reduce branch and instruction locating overhead. The merge condition will be discovered in three conditions:

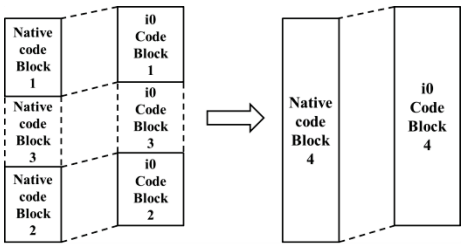
1. Translation of the i0 block begins right after the ending of an existing translated i0 code block.



2. An existing translated i0 block starts from an instruction boundary of the currently translating block.

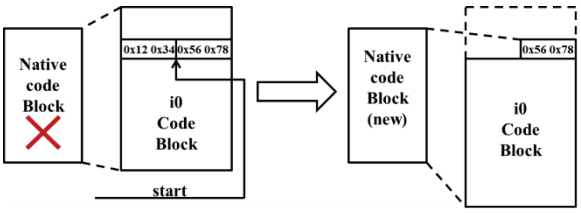


3. The combination of both above conditions.



3.3 Instruction reuse and code / data Obfuscation problem:

The i0 instruction set is essentially a CISC like ISA. The variety of the instruction length makes binary-translator complex since there are different ways to decode a block of i0 code given different entry address. For example, given an i0 code block and an entry address at the very beginning of the block, binary-translator will decode the i0 instructions linearly and generate native code. Afterwards, another entry address within the same block is given which falls in the middle of one i0 instruction that decoded and translated previously. In this kind of situation, binary-translator has to treat two entry addresses as the starting of two different i0 blocks even though they overlap each other. However considering the behavior of cc0 (compiler of c0 [3]), chances that this kind of situation occur and both blocks are executed is very small. So the design of binary-translator forbids the translated i0 blocks to overlap. If binary-translator is indeed given an i0 address which falls in the middle of some instruction of a translated i0 code block, which implies the case does happen, then binary-translator determines that there were some data or fake instructions in the i0 block translated and it caused trouble. Hence, the entire translated native code block is deleted and a new native code block is created by translating from the new i0 address.



Since the cost of deleting a translated block is high, the size of the translated block becomes critical. During the translation process, a lower limit, an upper limit and a 'look ahead' limit are set. The 'look ahead' limit is used to define the size of the 'look ahead' block. Binary-translator will:

- Never translator an i0 code block with size larger than the upper limit.
- Abort the translation whenever an invalid i0 instruction is encountered.
- After the size of i0 code translated exceeds lower limit, translation can be aborted when an unconditional jump instruction is encountered and there is no i0 address referenced in the 'look ahead' block.

The Figure 3.3 shows a sample of the abort condition 3. In this example, the translated i0 code size has exceeded the lower limit when an unconditional jump instruction is encountered. Since there is no instruction referencing the 'look ahead' block, translation is aborted.

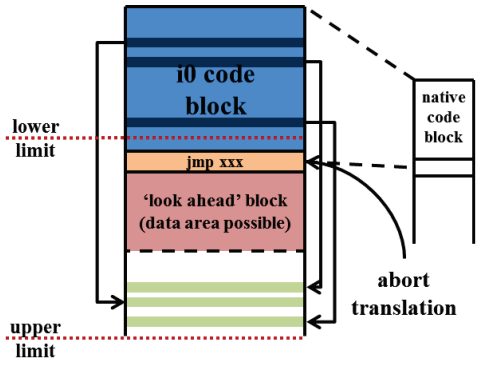


Figure 3.3

3.4 Code location problem:

While translating branch instructions, the branch target is originally an address pointing to i0 code. It is binary-translator's duty to convert it to an address pointing to native code. To optimize branch efficiency, different techniques are used to solve code location problem.

3.4.1 Static branch:

When translating i0 branch instruction which has a determined target address (encoded as immediate in the instruction), binary-translator

will generate different types of branches in native code according to whether it is an inter-block branch or an intra-block branch. If target i0 instruction is translated in the same block, the target native address will be resolved during translation, and relative jump will be generated directly. If the target i0 instruction is not translated in the same block, including the situation of 'instruction reuse problem' (see above), then an absolute jump to the trampoline will be generated.

A 'trampoline' is a list associated with each native code block. Each entry in the trampoline contains a pointer points to a native inter-block jump instruction and a piece of code which can be executed. This piece of code will emulate a function call to binary-translator handler function. Eventually the branch jump instruction is modified to directly jump to the target block and execution is passed to the target block. (Figure 3.4.1) The reason of this complicated design is to 'pass argument' to the binary-translator function so that the binary-translator will be aware of the address of the jump instruction and the address of the trampoline slot. A trick is played by using the 'call' instruction in both the jump instruction and the trampoline which can push the 'return address' to stack. Thus, handler function reads the stack and gets both addresses.

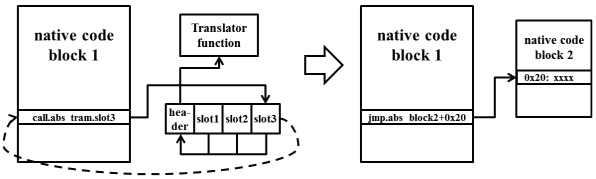


Figure 3.4.1

Note: The reason of using relative jump intra-block and absolute jump inter-block is to favor the rebasing of a native code block so that merging of blocks is much easier. (See section 4 below) Also note that x86_64 ISA encodes relative jump and absolute jump differently and the length of both instructions are different. To be more specific, the length of relative jump is shorter than the length of absolute jump. In order not break 'fake translation' (3.1), the binary-translator inserts some 'nop' instructions after relative jumps to eliminate the length difference.

3.4.2 Dynamic branch:

The dynamic refer to the situation that the branch target can only be determined at run-time. The 'indirect branch' in the i0 ISA is regarded as this kind of instruction. When such instruction is 'executed', the mapping between native address and the given i0 address must be found as quickly as possible. A PC (program counter) hash-table is designed to store the cached mapping. When translating 'indirect branch' instruction, binary-translator generates instructions which read the target i0 address to register, hash it to PC hash-table slot, check if the slot stores valid native code address and branch to native code if valid or branch to binary-translator if invalid. (Figure 3.4.2) After the target native block is located or created, the mapping will be added to the hash-table.

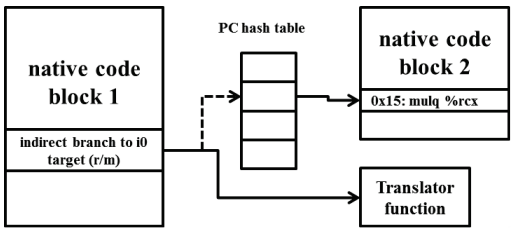
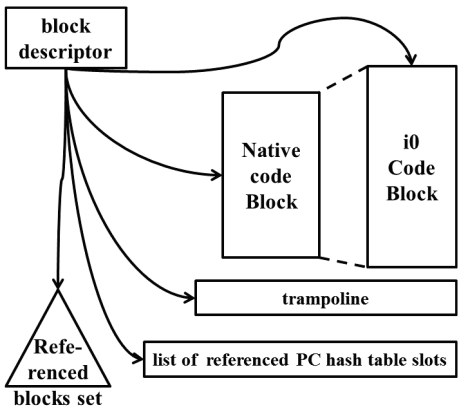


Figure 3.4.2

4. Manageable translation.

Binary-translator manages the translated blocks for time and space efficiency. By design, binary-translator supports creating, deleting, rebasing and merging (if situation, see above, allows) translated blocks. Since a translated block may have cross reference with several other data structures, a block descriptor (Figure 4) is used to describe:

1. The original i0 code block it translated from
2. The native code block it self
3. The relocation list / trampoline (3.2.1)
4. The set of pointers to the descriptors of blocks that reference the current block (also named as 'block reference-set' below)
5. The set of pointers to the PC hash table slots that is being referenced by the current native code block (also named as 'hash-table reference list' below)



The 'block reference-set' and 'hash-table reference list' data structures are added to support block deleting and rebasing operations. (Figure 4.1, Figure 4.2)

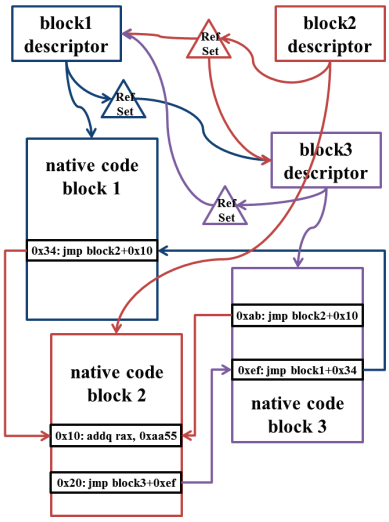


Figure 4.1

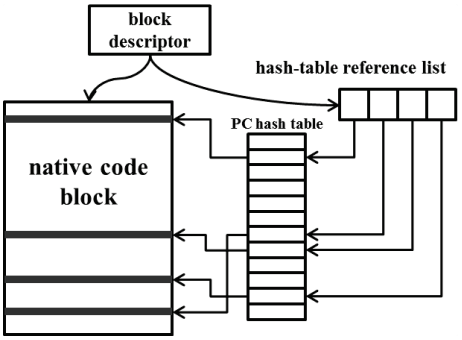


Figure 4.2

The block descriptors play the role like a handle to a resource. Globally, an index is built to organize these descriptors using both red-black tree and doubly linked list ordered by address of the corresponding i0 code block. If an i0 address is given, binary-translator can quickly tell whether there is a translated native code block present for that i0 address and find the descriptor if it exists.

4.1 Block operations in detail:

Block creation: The creation process is simple since it does not interact with other blocks. After the native code block is translated and trampoline and descriptor are set up, the block descriptor will be inserted to the global index. Both 'block reference-set' and 'hash-table reference list' are empty at this time.

Block rebasing: The rebasing process only requires moving the native code block to a desired place in memory and fixing the values in the PC hash-table. Since the 'hash-table reference list' contains all the hash-table slots that currently referencing the native code block, binary-translator only needs to 'fix' the values of those slots.

Block deletion: The deletion process is costly. First, all the blocks in the 'block reference set' have to be 'informed' and all the inter-block jumps which are targeting the deleting block in those blocks have to be redirected back to the

trampolines. Then the trampoline of the block will be traversed to remove the block from other block's 'block reference set'. Also all the hash-table slots referencing the deleting block have to be invalided. Finally the descriptor and the resource it described are deleted.

Block expansion: The expansion process is a combination of rebasing operation and the process to resolve intra-block branches after expansion. The 'merge condition 1' and 'merge condition 2' (3.2) are both treated as expansion.

Block merging: Only when the 'merge condition 3' (3.2) satisfies, blocks can be merged. To merge two blocks, not only should the code blocks be merged but also the other data structures. In the global descriptor index, one of the two descriptors is deleted. For all other blocks, the reference to the deleted descriptor in the 'block reference-set' is deleted. (Figure 4.3)

These operations support the translation process described in section 3.

5. Conclusion:

In theory, the new implementation of binary-translator does provide correctness in the cases that i0 program does not self-modify, that is, the 'executed' i0 code is read-only. And binary-translator manages resources in a controllable manner preventing memory-leak problems. However, the performance that L0 system will gain depends dramatically on the size of the i0 code block to translate. So the 'lower limit', 'upper limit' and 'look ahead limit' (3.3) should be carefully chosen. More tests should be conducted in the future in order to probe the appropriate values.

[1] Zhiqiang Ma, Zhonghua Sheng and Lin Gu. *DVM: A Big Virtual Machine for Cloud Computing.* IEEE Transactions on Computers

[2] i0 specification

[3] cc0 compiler
http://lazero.net/documentation.htm

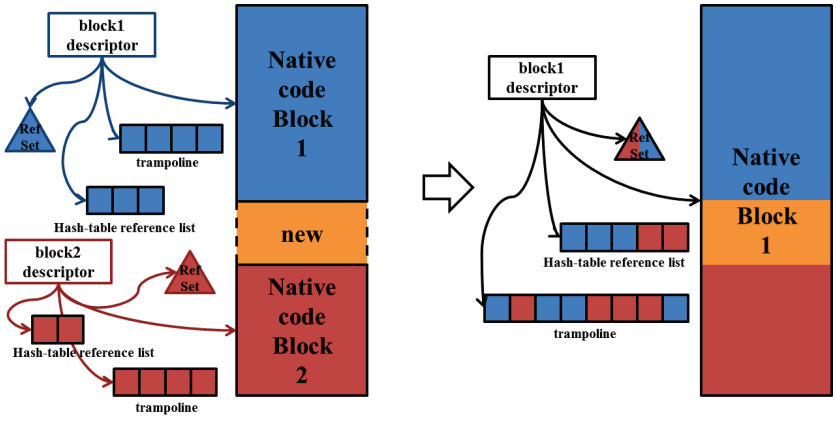


Figure 4.3