

Rapid return address determination in binary translation

M. K. Gschwind

Address space mapping between the original (emulated) instruction address space to translated (native) code is an important function in binary translation. Previous binary translation systems have used lookup tables to support mapping between emulated and native instruction address space. These lookup tables can be supported with appropriate hardware structures or implemented using software only.

While such a translation scheme is appropriate for general branches such as PC-relative branches, register indirect branches, or function invocation, such implementation is inappropriate for function call return. Using such mechanism does not preserve the predictability of function call return information, and thus leads to performance penalties.

To improve the performance of function call linkage in a binary translation environment, the present invention uses a hardware return address stack for binary translation architectures. The invention consists of:

- a function return stack, storing the return addresses of the original emulated address space and of the host address space,
- one or more instructions to record both the original and translated address space return address on the function return stack (executed when control is transferred to a subroutine), and
- one or more instructions to effect a function return using the information stored on the function return stack (executed to determine the subroutine return address upon function exit).

The instructions to manipulate the function return stack can work either by explicit manipulation of such stack, or the functionality can be performed as a non-architected optimization to improve performance of address space mapping. (The latter method is common for native function return stack implementations used for branch prediction.)

Using a return address stack to maintain the address space mapping allows to capture function call returns with only a few entries. By comparison, a hardware based caching mechanism for translation tables as previously disclosed in IBM Invention Disclosure Docket Number YOR8-1998-0334, incurs compulsory misses for the first use of each translation. Additional misses are incurred when the entry is displaced by a mapping in the same congruence class.

A hardware return stack requires only a few entries, yielding a high translation accuracy rate with low hardware cost. Experiments have shown that using a return address stack reduces the number of entries which need to be stored in a lookup table, when a hybrid scheme is used which employs hardware-cached lookup tables for general branches and a return stack for function return.

Unlike native address return stacks which are used to increase prediction accuracy of branch prediction, the present invention is used as authoritative return address information. To ensure correct operation in the presence of functions which modify their return address, such situations must be detected. A possible solution is to track changes to the function return address and

invalidate the function return stack when such changes are detected. (This requires detection of changes by asynchronous functions such as exceptions, interrupt handlers, etc.)

An alternative scheme is to store both the return addresses for the emulated and the native address space on the return address stack. When a function return is attempted, the emulated address is compared to the expected emulated return address, and if a match is found, a control transfer to the stored host return address is performed. This implementation can detect whether the emulated address space return address has changed, and can automatically use the appropriate host return address if no change of the return address has occurred. Since few functions modify their own return address, the described approach has a high translation success rate.

To implement a function return stack with dual native and emulated return addresses, both the native and emulated return address have to be established on a function call and added to the function return stack.

On function return, a binary translated program verifies that a function has not changed the return address for the emulated address space. If this has been changed, then the host address corresponding to the new emulated return address has to be established and execution continues at the translation of that changed return address. Otherwise, the return address can be taken from the stack.

The program logic for this test can be described in pseudocode as follows:

```
pop (emulated, host)           // pop native and emulated return address

if (architected_return == emulated) // has architected return address changed
    goto host;                   // no - go to native return address
else
    goto lookup(architected_return); // yes - translate new return address
```

When a translation is destroyed to reclaim translation cache space in a dynamic binary translation scheme, the associated entries in the return stack have to be destroyed as well. This can either be selective invalidation of only the concerned addresses, or of the entire return address stack.

By implementing a unified address translation unit which contains both a branch translation cache as disclosed in IBM Technical Disclosure Docket Number YOR8-1998-0334 and a function return stack with dual addresses (native and emulated), high efficiency can be achieved with only a single address translation request. A hardware implementation may access both structures in parallel and supply the translation irrespective of whether the translation was taken from the function return stack or the branch translation cache.