

Method and apparatus for determining branch addresses in programs generated by binary translation

M. K. Gschwind

Problem solved by this invention

Binary translation allows to maintain compatibility across different architectures while still executing at native speeds. To this end, the original program is treated as input to a binary translator which analyzes the program and generates equivalent code for the current base architecture. Binary translation can either occur as a separate step prior to program execution, also referred to as 'static binary compilation', or binary translation can occur for each code fragment in the original program as it is executed for the first time during program execution, also referred to as 'dynamic binary translation' [1]. Alternatively, special purpose hardware can translate instructions on the fly either during the instruction decode, instruction fetch or instruction cache miss phases in program execution.

Data contained in registers or in memory locations represent the values of the original program before binary translation. This includes addresses of possible branch targets. Thus, branch target addresses stored in memory or registers refer to program locations in the original program, not the equivalent entry points in the binary translated program. To actually perform a branch to some address, the binary translated program has to check whether a suitable entry point in the translated program can be found, and if so, the address of that location has to be determined. In dynamic binary translation, this mechanism can also be used to branch between translation units (such as pages, tree regions, or superblocks) and accounts for changes in the memory mapping and reclaiming of translations from a dynamic translation cache. In static binary translation, it can be used to detect previously undetected entry points into the code, such as through register indirect branches.

If no entry point can be found,

- in a static binary translation scheme, the program has to branch to an interpreter which will interpret the original program code which is still maintained as part of the address space, or
- in a dynamic binary translation scheme, the program has to invoke the dynamic binary compiler to generate a translation of the code fragment in question and then continue execution at the newly translated code fragment.

In currently available systems, such as Mimic [2], Accelerator [3], VEST [4], FX!32 [5], and DAISY [6], checking for an existing translation and generating a branch address from the branch target address in the original (emulated) program is performed using software, commonly by a table lookup. The table format can either be a linear mapping table or a compressed table format such as hash tables. Both of these approaches have significant problems: linear lookup tables require significant memory space, whereas using compressed table formats require a significant computation time to access such a table.

This invention addresses problems arising from checking the existence of a translation for a given code fragment, and deriving the appropriate address of the translated code fragment from the

original branch target address of the emulated program. In particular, this invention addresses the cost associated with such checking and address translation, both in storage and execution time requirements, the latency arising from such translations, and interference with the memory hierarchy (access to memory tables leads to cache pollution).

This invention consists of:

- a hardware mechanism for translating addresses from the original (emulated) program space to the binary translation address space in order to reduce latency of such translation,
- a mechanism to manage the hardware storing these translations,
- a mechanism to check for the existence of a translation for a particular code fragment in order to reduce latency of such translation,
- a mechanism to deal with limited hardware storage capacity for translations,
- a mechanism to deal with non-existing translations both in static and dynamic binary translation, and
- a mechanism to deal with multiple processes executing binary translated code.

The present invention can be used for system or process-based binary translation. In system-based binary translation, both system and problem state mode of the migrant architecture are emulated by a binary translator and operating system code executes in migrant address space [6]. In process-based binary translation, the operating system executes in native host address space, and only problem state user programs are emulated by the binary translation system.

State of the art

Binary translation has been used by a number of system designs to achieve binary compatibility to allow execution of binary program compiled for a given instruction set architecture (migrant architecture) on a substantially different target architecture.

Binary translation can be used either statically or dynamically. Static translation is performed offline and affords exacter analysis of the compiled input program, but cannot fully translate all code because not all entry points can be detected (e.g., lookup tables, computed GOTOs). Dynamic binary translation works at run-time, and translates each group of instructions when control reaches the group for the first time.

Examples of such binary translation systems are:

- Mimic - dynamic binary translation of System/370 code to RT/PC code [2]
- Accelerator - a static binary translator for TNS CISC code to Tandem's MIPS-based Himalaya architecture [3]
- VEST and mx - static binary translation of Vax/VMS and MIPS/Ultrix code to DEC Alpha, respectively [4] :
- FX!32 - hybrid dynamic/static translation of Intel x86 code to DEC Alpha code under Windows NT [5]
- DAISY - dynamic binary translation of PowerPC code to a VLIW architecture [6]
- Shade - dynamic binary translation of SPARC V8, SPARC V9 and MIPS code to SPARC V8, with instrumentation for profiling data collection [7]
- ATOM - profiling tool set based on static binary translation from DEC Alpha to DEC Alpha code [8]

- several Java Virtual Machine implementations use static [9] or dynamic binary compilation [10] [11] [12]

Both static and dynamic binary translation mechanisms must maintain all data, including all addresses which may reside in registers or tables, such that they refer to program locations in the original code. Alas, to actually execute branches, these addresses need to be translated to host format, using one of several methods. In addition to performing such translation, a check must be performed if the address being translated was detected as a valid entry point. If not, the program must invoke

- for static compilation: an interpreter to interpret the original code which must be included with executables generated by static binary translation
- for dynamic compilation: invoke the dynamic binary compiler to generate a translation of such previously untranslatable code fragment starting at the desired entry point.

Previously, migrant to host address translation has been performed in software using one of three ways:

1. Direct branches can be inlined under a number of circumstances, using native branch instructions in the executable code generated by binary translation.
2. Linear lookup table: a sparse table is maintained, where each possible input address has an entry which may list the corresponding address in the translated code. A special value (such as address 0, or the address of the interpreter/translator) indicates that an address is not a valid known entry point. A register indirect branch is then performed to the translated address.
3. Compressed lookup table: compression schemes can be used to compress the table, e.g., by using a hash table, or only storing part of the address and computing the remainder. A register indirect branch is then performed to the translated address, or the interpreter/translator is called if no such address was found.

The table lookup operations used in 2 and 3 can be contained in a subroutine, but are typically inlined to reduce the cost of such translations. Additional performance improvements can be gained under some circumstances by generating inline branch code for a few common cases.

In previous work, a fixed mapping has been proposed to reduce or eliminate the cost of table lookups in translations [6]. This approach eliminates the need for table lookups because a simple, algorithmic method exists to generate a host address from a migrant address. In this proposal, every address in the original address is scaled so that an emulated instruction can map into multiple native instructions.

Limitations of the state of the art

Each of the previously discussed schemes incurs some limitations, which fall into one of several categories:

- not applicable to all branch types and/or translation schemes
- code size penalty
- speed of generated code
- code generation speed
- memory requirements to store meta-information

Inlining branches is an effective method for the majority of branches, if the branch target address is well known and immutable (provided that self-modifying code need not be supported). A branch target address can be determined to be immutable because the original program executes an actual direct branch, or by using data flow analysis to establish that an indirect branch will always branch to a certain address (or set of addresses). This approach is most efficient for static binary translation, where significant code analysis can be performed, and also because all code will be generated at the same time. For dynamic binary translation, code generation time is a significant factor in overall performance, so less code analysis can be performed. Also, special care has to be taken to correctly generate code when the code fragment at the target address has not yet been translated, to ensure that the dynamic binary translator will be invoked, using trampolines or some other method to invoke the translator.

Linear lookup tables provide the simplest implementation for branch target address translation by associating each possible branch target in the original program with a target address in the newly translated executable code. However, this approach requires to maintain very sparse and large lookup tables which are equal or larger in size to the original program code. Especially when the original program representation is in a variable-length instruction format, each byte location has to be associated with a pointer into the new executable, which can cause these lookup tables to be significantly larger than the original program [2]. In addition, lookup tables will likely suffer from poor locality and have high cache miss rates.

Compressed table formats allow a significant reduction [3] [7] in table size. However, when a translation is attempted, the compressed table format has to be accessed using possibly significant computation time. Using a compressed table format, an address is assembled from several parts, with redundant information being stored just once. Accelerator [3] incurs a cost of 11 cycles to generate an actual branch target address. An alternative approach is to store <original address, translated address> address tuples using a hash function, and compact tables. This incurs the cost of generating the hash value for requested values, and following hash chains in the case of collision [7].

In addition to these adverse effects on running time and program size, secondary effects can severely affect overall system performance by overburdening and overflowing various system resources:

- Instruction caches may suffer from increased program size as a result of inlining the computation of addresses from compressed tables.
- Data cache performance may suffer as useful program data is displaced by translation table lookups. (Note that we cannot expect much spatial locality in these pages, so each lookup will replace a full cache line of program data for accessing a single address translation.)
- System memory may become a scarce resource as lookup tables require significant system memory (and the working set of binary translated processes is increased), leading to additional paging activity and degrading overall system performance.
- TLB performance may degrade as increased program working sets lead to a higher TLB miss rate.

When using a fixed mapping from migrant to host target address, the cost associated with translating addresses is eliminated because an algorithmic way exists for translating a migrant to the corresponding host address. However, this scheme poses significant restrictions on the code

generations, and still requires a mechanism to establish both valid entry points and whether a code fragment has already been translated.

Description of this invention

The invention implements address translation in hardware using dedicated hardware resources and additions to the instruction set. Frequently accessed translations are stored in a lookup table, and can be accessed using a single machine instruction. This approach allows the most common translations to be available in a single cycle. Translation cache misses are handled in software in this embodiment to minimize hardware complexity which deals with the uncommon case of a cache miss.¹

Mode of operation

In the branch target address translation cache, migrant/host address translation pairs for frequently used branch target addresses are maintained in a CAM-like memory structure in the processor core. Instructions are provided to set and query address translations.

When binary translated code attempts to transfer control to an unresolved address, the branch target translation cache is queried to provide the address of the translated code corresponding to the original migrant code at migrant branch target address.

The processor implements a single-cycle instruction of the form 'btat rd = rs' which translates a migrant address contained in register rs and stores its translation in target register rd. During code generation, these address translation instructions are inserted at the point of branches which require address translation, such as register indirect branches or branches across translation units.

When an address translation is requested, the processor uses the supplied migrant target address, and uses it as an index into a CAM-like translation table which is maintained as a part of the processor state. The actual translation table is managed as a cache, and may be accessed using either a virtual address, the concatenation of the virtual address with the process identifier (PID), or real address of the emulated (migrant) machine.¹Note that the latter case requires emulation of the migrant memory management mechanism, and is mainly useful for systems which attempt 100% architectural compatibility, such as DAISY [6].²

Only a subset of the address bits are used to index the translation table, in a cache like fashion the remaining bits are present as tags. The translation storage can be direct mapped and store only one translation which corresponds to the selected bits of the input, or multiple associations can be maintained in a set or fully associative manner, with the appropriate translation selected by a multiplexer after the tag check, where the tag would consist of the remaining bits of the input.

The size, organization (direct mapped, n-way set associative, fully associative) and replacement (LRU, FIFO, random, etc.) strategy of the branch translation lookup table is independent of the mechanism. Such lookup table can vary significantly in size, and the organization and

¹ Hardware implementations of cache miss management are an alternative implementation option. The implementation of such features based on a well defined full translation table format will be obvious to anybody skilled in the art.

replacement strategy can be any of those that are currently in use for TLBs, caches, and other such structures. A study of the impact of such implementation decisions has been performed and is reported in [14].

This mechanism is significantly different from TLBs in that it does not map fixed size address ranges to another address range, but only contains the address of a single translation. However, many of the same management issues addressed for TLBs (software vs. hardware management, optimization of translation table formats, and implementation of hardware walks of translation tables) can be applied to the proposed structure.

Handling translation cache misses

When no translation is found in the translation cache, the situation is handled in software by an address translation miss handler. The lack of a translation indicates that a translation does not exist, or that it has been displaced by another translation in the translation cache. Typically, this address translation miss handler will first establish whether a translation already exists, or whether the interpreter/translator should be invoked. If a translation already exists, the software will load it into the translation table (possibly displacing other translations).

When no translation for a requested address is found in dynamic binary translation, the dynamic binary translator will be invoked to translate the program group starting at the offending address, and then continue as if an address had been available, but not found in the lookup table. In static translation, control may either be transferred to an instruction set architecture interpreter, or an error status be reported.

The software implementation of the address translation miss handler can reside either in the user address/protection space or in the operating system address/protection space, according to different implementation and security requirements.

Depending on whether the miss handler is implemented in user or kernel space, different mechanisms can be employed to effect control transfer to the handler in case of a cache miss:

- If the miss handler is to be implemented in kernel space, a address translation cache miss will raise an exception to transfer control directly to the kernel (see figure 1).
- If the miss handler is to be implemented in user space, a default target address can be returned by the instruction accessing the branch target address cache. This address would point to the translation miss handler. Using such scheme eliminates all special handling overhead to test whether a cache miss has occurred (see figure 2).

When the miss handler is implemented in kernel space, the operating system is responsible for resolving the condition either by updating the branch target translation cache and reexecuting the instruction performing the address translation, or by emulating the operation of the address translation instruction.

In the case of a user-space miss handler, a special purpose register supplies a default instruction when no appropriate entry is found. Typically, this address will point to a service routine which may check compressed tables for the existence of the target address, reload it in the lookup table and branch to the originally requested address (which would be stored in a special purpose register or be available by examining the processor state prior to control transfer to the miss handler).

The cost of handling address translation misses is greatly reduced by a user-level translation miss handler, since it eliminates expensive context switches between user and system processor state. (A kernel level implementation requires two context switches, from problem to system state, and back, for each table update.)

A secure environment does not require that software miss handlers reside in kernel space. The only requirement to ensure system integrity is that a process may not load address mappings for other processes. This ensures that a given process cannot interfere with the execution of other processes on the same system. In addition, since process-based emulation operates within the virtual address space of each process, it cannot pass control outside the memory space assigned to it by the operating system. To ensure that a process-based system cannot manipulate other processes' entries, it is sufficient to make the manipulation of process identification (PID) used in accessing the branch target address translation cache a privileged operation.

The target addresses in the translation lookup scheme can either be physical/real addresses in the target processor, or virtual/effective addresses relative to the current process. If the target addresses are represented by the physical/real address, then the branch target address translation cache has to be managed operating in the system level context to ensure integrity.

In process-based emulation, using virtual addresses would require branch target address cache invalidation on each process switch, similar to virtually addressed caches. By using a scheme based on the virtual address and a process (or address space) identifier, the performance penalty associated with such cache invalidation can be avoided. Using this scheme, translations for multiple processes can be maintained in the branch target address cache. By restricting modification of the PID source for address table lookup and modification to supervisor privilege level, it is possible to perform cache management in user space and conform to the previously stated system integrity requirement that each process may only install mappings for its own address space. The PID register would then typically be updated when a process is scheduled on a context switch.

Potential use of this invention

Uses of this invention are speedup of address mapping of branch target addresses from emulated to target machine address space for static and dynamic binary translation, in any type of microprocessor (CISC, RISC, VLIW, EPIC). The scheme is applicable to both register indirect branches and to branches between translation groups, as well as to any other type of paired address translations that may be necessary to correlated migrant and host instruction or data memory layout.

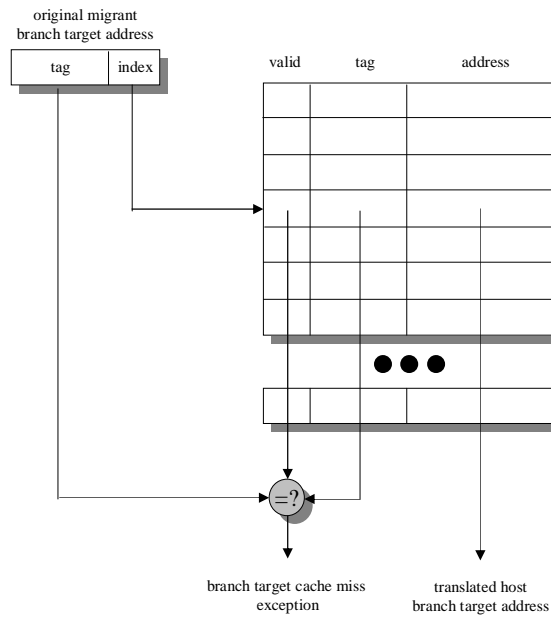


Figure 1: Branch address translation lookup with cache miss service at the operating system kernel level.

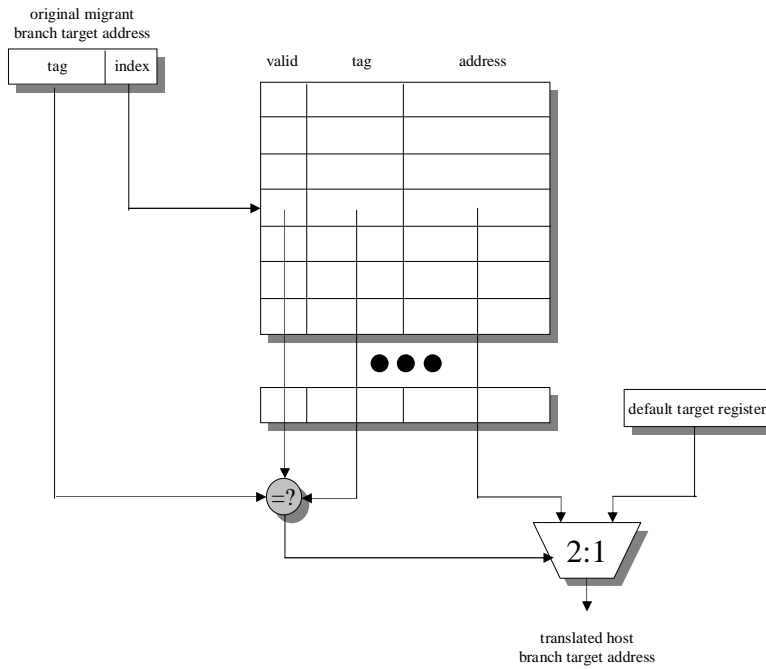


Figure 2: Branch address translation lookup with default target address register for user level cache miss handling.

References

- [1] Phil Emma, Method and Apparatus for the Transparent Emulation of an Existing Instruction-set Architecture by an Arbitrary Instruction-set Architecture, U.S. Patent No. 5619556, April 1997.
- [2] Cathy May, Mimic: A Fast S/370 Simulator, ACM SIGPLAN 1987 Symposium on Interpreters and Interpretive Techniques, 1987.
- [3] Kristy Andrews and Duane Sand, Migrating a CISC Computer Family onto RISC via Object Code Translation, Architectural Support for Programming Languages and Operating Systems (ASPLOS-V), 1992.
- [4] Richard L. Sites, Anton Chernoff, Matthew B. Kerk, Maurice P. Marks, and Scott G. Robinson, Binary Translation, Communications of the ACM, 36(2):69-81, February 1993.
- [5] N. Rubin and Anton Chernoff, Digital FX!32: A Utility for Fast Transparent Execution of Win32 x86 Applications on Alpha NT, Hot Chips IX, Palo Alto, CA, August 1997.
- [6] Kemal Ebcioglu and Erik Altman, DAISY: Dynamic compilation for 100% architectural compatibility, Proc. of the 24th International Symposium on Computer Architecture, pp. 26-37, Denver, CO, June 1997.
- [7] Robert F. Cmelik and David Keppel, Shade: A Fast Instruction-Set Simulator for Execution Profiling, 1994 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems, May 1994.
- [8] Amitabh Srivastava and Alan Eustace, ATOM: A System for Building Customized Analysis Tools, 1994 SIGPLAN Conference on Programming Language Design and Implementation, June 1994.
- [9] Todd A. Proebsting, Gregg Townsend, Patrick Bridges, John H. Hartman, Tim Newsham, and Scott A. Watterson, Toba: Java for Applications, University of Arizona, Tucson, AZ, 1997.
- [10] Tim Wilkinson, KAFFE: A free virtual machine to run Java code, <http://www.kaffe.org>, 1997.
- [11] Andreas Krall and Reinhard Grafl, CACAO - A 64 bit JavaVM Just-in-Time Compiler, Workshop on Java for Science and Engineering Computation, June 1997.
- [12] Kemal Ebcioglu, Erik Altman, and Erdem Hokenek, A Java ILP Machine based on fast Dynamic Compilation, International Workshop on Security and Efficiency Aspects of Java, 1997.
- [14] Michael Gschwind, Branch Target Address Translation Caches for Binary Translation, 1997.