Dynamic Binary Translation *

Mark Probst
CD Laboratory for Compilation Techniques
http://www.complang.tuwien.ac.at/schani/

Abstract

This paper presents an overview of dynamic binary translation. Dynamic binary translation is the process of translating code for one instruction set architecture to code for another on the fly, i.e., dynamically. Dynamic binary translators are used for emulation, migration, and recently for the economic implementation of complex instruction set architectures.

Most of the problems occuring in dynamic binary translation are discussed and solutions are presented and weighed against each other.

Finally, the dynamic binary translator `bintrans`, developed by the author, is presented.

1 Introduction

Binary translation is the process of translating machine code (binaries) from one instruction set architecture to another, for the purpose of running a program written for the one instruction set on the other one. Systems doing this are usually referred to as "emulators". Binary translation, hence, is an efficient form of emulation.

Most binary translation systems currently in use are migration tools which aid the transition from old architectures to newer ones. A recent prominent example for binary translation not following this pattern is the Transmeta Crusoe processor, which dynamically compiles x86 machine code to a proprietary VLIW machine code.

Binary translation can be done statically by translating a whole binary to a new binary for the target platform, or dynamically, by translating code on the fly. For von Neumann architectures, where code and data reside in the same memory, static binary translation is never a complete solution, due to problems such as dynamic linking and self-modifying code. Dynamic binary translation overcomes these problems, while at the same time creating new ones. Most prominently, dynamic binary translators must be fast. Otherwise the translation overhead would be higher than the cost of running the translated code.

This paper discusses the most important issues arising in the design of a dynamic binary translator. We will use the terms "foreign" and "native" to distinguish between the platform which is to be emulated and the platform on which the emulator is to be run.

The paper is structured as follows. Section 2 describes the differences between emulating a whole hardware platform and emulating the user-level environment which an application sees. This paper does not discuss in detail issues of the former approach. Section 3 lays out the overall architecture of a dynamic binary translator. Section 4 discusses instruction selection, which means choosing which native instructions to use to implement a foreign instruction. Section 5 describes how to handle the issue of different byte order between the foreign and the native platforms. Section 6 discusses several issues regarding the emulated address space. Section 7 presents several ways of mapping foreign registers to the native register set. Section 8 argues why self-modifying code is a problem for a binary translator and presents solutions. Finally, section 9 briefly presents the dynamic binary translator `bintrans`, developed by the author at the Vienna University of Technology and hopefully released as Free Software by the time you read this.

*This work is partially supported by the Austrian Science Fund as part of Project "Compiler Based Disjoint Eager Execution" under Contract P13444.

2 Emulation at System-Level versus at Application-Level

There are two common ways of making an application run under a foreign system (hardware/operating system combination).

The first one is to emulate the complete foreign hardware and let the foreign operating system run under this emulation. Systems taking this approach include VirtualPC [5], VMWare [9], and Bochs [1]. Emulating a hardware platform usually involves emulating that hardware's memory management unit (MMU), it's peripherals (including, for example, graphics hardware), as well as emulating the foreign instruction set. Most of the latter is not necessary when the foreign instruction set architecture is the same as the native one. VMWare is an example for such a system.

The second way of running a foreign application is emulating only what the application sees of the system. This usually means emulating the system call interface as well as the foreign instruction set architecture. A prime example of this approach are Digital's VAX and MIPS to Alpha migration tools `mx` and `vest` [7]. An example which only involves emulating the system call interface is FreeBSD's ability to run Linux i386 binaries on i386 systems. Wine takes an even higher-level approach in not emulating the Windows system call interface (native ABI) but in providing its own implementation of the system libraries (DLLs).

Both approaches have their merits and disadvantages. System-level emulation makes it possible to run arbitrary operating systems running on the foreign hardware. It also makes sandboxing the emulated system very easy.

Application-level emulation, on the other hand, provides much nicer integration of the foreign application into the native environment. Also, since only the system call interface has to be emulated, as opposed to the whole hardware platform, application-level emulation is usually easier to implement as well as more efficient, if only for the reason that the operating system runs natively, i.e. at full speed.

This paper does not discuss issues of system-level emulation.

3 Overall Architecture

The architecture of a dynamic binary translator is very similar to that of other Just-in-time compilers.

In the simplest case, the binary translator is a process running the foreign application in its own address space. For more elaborate schemes, see section 6. The first step is loading the foreign application.

After that, several initialization steps must be taken. In the case of UNIX-like foreign platforms, the stack of the foreign application must be initialized to hold the environment variables, command line parameters and other information in a specific format. Furthermore, the registers of the foreign machine must be initialized.

Now, the foreign application can be started, which means beginning to execute its machine code, starting at the start address (which is specified in the executable file). The obvious strategy for a dynamic binary translator is to always translate foreign code to be executed and to execute the generated target code. It could, for example, translate from the first instruction to be executed up to and including the next jump. More elaborate schemes are discussed in section 3.4.

The jump would be translated into target code jumping to a so-called "dispatcher". The dispatcher is given an address in the foreign application and has the duty of resuming execution of the foreign application at that address. It usually does this by first checking whether the code at that address has already been translated. If it has, the corresponding target code is executed. If it hasn't, it will be translated first. That way, control switches back and forth between the dispatcher and the generated native code, with the dispatcher sometimes calling the compiler to translate yet-unknown foreign code.

3.1 System Calls

To be useful, an application must have some contact to the "outside world". In modern operating systems, this is achieved via system calls. A system call is initiated via some special instruction. On the i386 under Linux, this is the "`int $0x80`" instruction. That instruction performs a context switch into the operating system kernel. The kernel's system call handler examines the application's state (usually its register contents) to determine which system call to perform and to get its parameters, if any. The encoding of information to be passed from the application to the kernel and vice versa is referred to as the "system call interface".

The system call interfaces between the foreign and the native platform usually differs. Differences can range from different system call numbers, over different structure layout, to foreign system calls which are not available on the native platform. That alone is reason enough why the binary translator must handle system calls of the foreign application. Another reason is that some system calls must be handled differently in the presence of a dynamic binary translator, for example because they could allow the foreign application to meddle with the binary translator's state. Memory management system calls are an example of this kind (see section 6).

In order to handle foreign system calls, the translator must make the code generated for a system call instruction jump to a system call handler. That system call handler is similar to the handler in a kernel. First, it must determine which system call is to be invoked. Depending on the system call, it must decode the paramaters passed by the application. The actual handling of the system call depends very much on its nature. Some system calls (like `write`) can be passed directly to the native kernel, while others (like `mmap`) must be handled by the binary translator. In that sense, the dynamic binary translator takes over the role of the operating system kernel from the point of view of the foreign application.

3.2 Patching of Direct Jumps

We previously layed out that the generated code would have to jump to the dispatcher whenever the foreign application performed a jump instruction. The dispatcher would then have to look up the native address corresponding to the foreign address to be jumped to (or compile the foreign code first, if necessary). Then it could jump to the native code.

Given that jump instructions are executed very frequently (once every ten instructions is a good estimate for many programs), the dispatcher can easily become a tight bottleneck.

A simple solution to this problem is to modify the code for a direct jump once the target code has been translated. Instead of transferring control to the dispatcher, the new code simply jumps to the native code corresponding to the foreign target code.

It is possible to do this with very little overhead. If the dispatcher can be passed the native address from where it was jumped to, i.e., the address of the jump that has to be modified, it can easily perform that modification after having looked up or translated the target code. An easy way of providing the dispatcher with that information is to use a subroutine call instruction (like `call` on the i386 or `bl` on the PowerPC) to transfer control to it from the native code.

3.3 Translation Cache

The memory region where generated code is placed is commonly called the "translation cache". Usually, this is a memory block which is filled in a linear fashion. Sometimes, the generated code fills up the translation cache. In such a situation, several strategies are possible.

The simplest one is to discard the contents and to start translation anew. This can become a problem when the native code for the working set of the foreign application is larger than the translation cache. In that case, the dynamic binary translator would thrash, much like an operating system does when a program's working set is larger than the physical amount of memory.

Hence, the binary translator should include some heuristic for determining when the translation cache needs to be enlarged. One indication is that the time the cache takes to get filled up is short.

It might seem a good idea to never purge the translation cache and simply enlarge it whenever necessary. This might, however, lead to excessive memory usage for large foreign applications. When running a large application, eventually nearly every part of the application will have executed at least once, but the working set at any given time during execution will usually be much smaller than the size of the whole application.

3.4 Unit of Translation

One of the major design decisions for a dynamic binary translator is when to translate and what (or how much).

The answer to the first question seems obvious: translate whenever code has to be executed which has not already been translated. This is not strictly necessary, however.

A different solution is to use an interpreter by default and to only translate pieces of foreign code which are executed frequently, using some appropriate definition of "piece of code" and "frequently". Such an approach has several advantages. First, less target code is produced, resulting in more efficient use of the translation cache and possibly a smaller working set, which improves cache performance. Second, since the compiler is used less frequently and the code it translates is likely to execute very often, it can spend more time generating better target code.

A promising new technique for choosing which pieces of code to translate has been pioneered by Dynamo [4], which is a dynamic optimization framework. Although Dynamo does not translate between different architectures, the technique used there can be applied to dynamic binary translators as well. The unit of translation is a trace. A trace is a sequence of instructions likely to be executed as a whole. It has a single entry-point and one or more exit points. One way to think of a trace is as a path through a control flow graph. If that path is executed very often, the spatial locality of the trace alone can speed up execution significantly due to better cache usage. Furthermore, several optimizations can be applied to traces. See the paper on Dynamo [4] for a detailed discussion.

We will subsequently refer to the unit of translation as "fragment".

4 Instruction Selection

Instruction selection is the process of selecting machine instructions for a source program. In our case, the source program is the foreign machine code.

A very simple instruction selector in a binary translator would have a fixed sequence of native instructions for each foreign instruction, which is generated whenever that foreign instruction has to be translated. This approach may work well if the native instruction set is semantically very rich or the foreign instruction set semantically poor. For example, an `add` instruction can usually be translated directly to a native instruction doing the same thing. In general, though, this will not give good results.

Take, for example, `rlwinm` on the PowerPC. The instruction "`rlwinm r3,r4,8,5,15`" has the following meaning: Rotate the contents of register `r4` 8 bits to the left, then do a bit-wise AND with a bit-mask which has bits 5 to 15 set, and put the result in register `r3`. Doing the very same thing on the Alpha takes 6 instructions, not counting the generation of the bit-mask. However, there are lots of special cases of `rlwinm` which can be handled quite efficiently on machines like the Alpha. For example, logical right and left shifts are implemented using `rlwinm` on the PowerPC. Hence, an instruction selector should identify special cases of instructions which can be implemented more efficiently than the general case.

4.1 Operands

The operand forms of foreign architecture instructions are an important consideration when writing an instruction selector. In fact, when translating an instruction set with complex operand forms, like the i386, handling operands correctly is one of the hardest parts.

The easiest operand to handle is a register. Given that the foreign register value somehow materializes in a native register (see section 7 for details on register allocation), that native register is simply given as an operand to the native instruction.

Immediates, although seemingly easy, are a bit more difficult to handle. The reason is that the sizes of immediates differ between instruction set architectures. The i386, for example, supports immediates ranging from 8 to 32 bits. On the PowerPC, immediates are most often 16 bits long. Depending on the instruction, they are sign extended or zero extended[1]. When confronted with a 32 bit immediate, which, for example, needs to be used in a native instruction which only supports 16 bit immediates, one should check whether that immediate fits into 16 bits. If it does, the immediate can be used without further

[1] Extension refers to the process of transforming an n bit number into an $n + m$ bit number. Zero extension simply prefixes the number with m zero bits, while sign extension prefixes the number with m bits, the value of which is the most significant (i.e. left-most) bit of the original number. Sign extension preserves the value of a two's complement number.

ado, otherwise it must be loaded into a register first. On most RISC architectures, loading a 32 bit value into a register costs either a load instruction or two non-memory instruction. Usually, two non-memory instruction will perform better.

More complex operand forms, like an i386 scale-index-base operand, usually require one or more native instructions to generate the operand.

4.2 Condition Bits

One of the most difficult aspects to implement correctly and efficiently in the translation of an instruction set architecture to another is the handling of conditionals and status bits.

In this section, we will first describe condition handling on the PowerPC, the Alpha, and the i386. Then we will discuss how such systems can be mapped to one another.

The Alpha's condition handling is by far the simplest of the three. Compare instructions compare the contents of two registers and produce either `1` or `0` in a target register, depending on whether the tested relation holds between the two registers or not. For example, the instruction "`cmpeq $1,$2,$3`" tests whether the registers `$1` and `$2` are equal. If they are, register `$3` is set to `1`. If they are not, it is set to `0`. Conditional branch instructions branch depending on the value in a register. The instruction "`beq $3,target`" branches to `target` if the value of the register `$3` is equal to zero. If it is not, it falls through.

The PowerPC has a special purpose register called the "condition register", which can be seen as 8 separate 4 bit fields. Each comparison instruction sets one of those fields. Three of the four bits are set to the results of the comparison (less than, greater than, equal). The fourth bit is set to a copy of a specific bit of another register, the purpose of which is not relevant for our discussion. A conditional branch instruction branches on an individual bit (specified as an immediate operand of the instruction) in the condition register. Many PowerPC instructions have an alternative form which automatically compares the result of their operation with zero and set the first condition register field (the four most significant bits of the condition register) accordingly. For example, the instruction "`addi. r3,r3,-1`" decrements the register `r3` by one and compares the result to zero. Bits 0, 1, and 2 of the condition register[2] are set to whether the result is less than, greater than, or equal zero. The instruction "`bs 2,target`" branches to `target` if bit 2 in the condition register is set (in our case corresponding to whether the result of the `addi.` instruction was equal to zero), otherwise it falls through. Additionally, the PowerPC has an integer exception register (XER) holding, amongst others, two bits which can be set to the carry and overflow of arithmetic operations.

The i386 has four condition bits[3] of interest, which reside in the flags register. Their contents reflect whether a value has its sign bit set, whether it is equal to zero, and whether an arithmetic operation generated carry and overflow. A compare instruction is just a subtract operation setting those bits depending on the result. Arithmetic relations can be determined by examining these bits.

Deciding how to generate code for setting and examining these bits on the native architecture depends very much on the foreign-native combination. Fortunately, it is not always necessary to compute all condition bits whenever a foreign instruction would do that. Especially on the i386, condition bits are very often computed and not used afterwards, i.e., written over without being read. Simple liveness analysis [3] can determine lots of such dead computations.

For an i386 to PowerPC translator, the four i386 condition bits can be mapped to two bits in the condition register (equal and less than) and to the carry and overflow bits in the integer exception register. That way, computing them costs little extra effort. An addition of two registers correctly setting all four bits, for example, requires only an `addco.` instruction. Examining the bits sometimes requires moving bits out of the integer exception register into the condition register, which costs an extra instruction.

On the Alpha, generating the i386 condition bits costs much more effort. In such a translator, it is beneficial to reserve a native register for each of the four bits.

The PowerPC condition bits can be surprisingly

[2] Bits are counted starting with the most significant one in the PowerPC architecture. Hence, bit 0 is the most significant bit.

[3] There are two others, namely the parity bit and the auxiliary carry bit, but they are used extremely rarely these days.

efficiently mapped to the Alpha. It helps to observe that the first (leftmost) condition register field is by far the one used most often. Efficient handling of the other 28 bits is not crucial for good emulation speed. An approach giving good results is to use a native register for each of the four bits in the field. That way, computation and access of a bit costs only one instruction each. It might be still better to keep all four bits in a single Alpha register. By virtue of the Alpha's conditional move instruction, each bit computation still costs only one instruction. However, accessing a bit costs two instructions in the general case, because the correct bit has to be selected first. This cost might be returned by the fact that it requires three native registers less than the other approach, hence requiring less loads and stores for register mapping (see section 7).

It is unclear how PowerPC condition bits can be efficiently mapped to the i386. To our knowledge, there is no published example of such a system.

4.3 Intermediate Representations

So far, we have focused on generating native code directly from foreign machine code. In classic compiler design [3], however, source code is first transformed into some intermediate representation (IR), which is then optimized. Afterwards, a code generator generates target assembler or machine code from the optimized IR.

IRs have at least two important advantages. First, it is easier to perform optimizations on IRs than to transform either the source or the target code. Second, IRs separate the front end, the optimizer, and the back end. Once there is a back end for, say, the Alpha and the SPARC, it would suffice to write a front end translating i386 code to the IR to get full-blown i386 to Alpha and i386 to SPARC translators. Furthermore, optimizations could be written independently from source or target machine.

In the context of dynamic binary translation, however, IRs have two distinct disadvantages. First, going through an IR is slower than translating directly from source to target code. The dynamic binary translator UQDBT [8], for example, reportedly uses 180,000 machine cycles to translate one byte of Pentium code to SPARC code, on average. Second, it is hard to come up with an IR which can capture all architecture idiosynchrasies, like condition bits, and at the same time making generation of efficient target code possible for these specialities. These difficulties make IRs for dynamic binary translation an interesting area for future research.

5 Byte Order

Today, there are two common ways of storing multi-byte entities to memory. Little-endian byte order stores less significant bytes on lower addresses while big-endian does it the opposite way. Say we wanted to store the 4-byte word `0x00010203` at address `0x80`. This is how the memory contents would be after the store:

Address	80	81	82	83
Little Endian	03	02	01	00
Big Endian	00	01	02	03

Processors using little-endian order[4] are the i386 and the Alpha, while big-endian machines include the PowerPC and the SPARC.

Clearly, binary translators must somehow make byte order differences between the native and foreign systems disappear, should they exist. This section examines ways to do this.

5.1 Byte Swapping

The most obvious technique for dealing with different byte order is to change the byte order before stores and after loads. Newer processores in the i386 line, for example, provide a byte swap instruction which handles that task quickly. Similarly, the PowerPC provides load and store instructions which automatically reverse the byte order. On systems which have such support, this is a good solution.

5.2 Address Space Swapping

On some architectures, swapping the bytes in a word may be a quite expensive operation. This can result in the run-time of a binary translated program to be dominated by the byte swapping process. On such

[4] Some processors, like the Alpha and the PowerPC can be configured to use either byte order. The enumeration states the configuration they are usually used with.

machines, it may be better to do the swapping the other way around. Instead of swapping the bytes in a word, one can turn the whole address space upside down so that the native byte order may be used.

Swapping the address space simply means using decreasing native addresses for increasing foreign addresses. The address `0` would be mapped to the highest available address, for example, and vice versa. Hence, byte addresses are simply inverted, which usually costs one instruction.

For loading and storing words it does not suffice to invert the address, however. The reason is that when a memory access instruction is given the address a, it uses the addresses $a, \ldots, a + (n - 1)$ to access an n byte word, while this technique requires using the words $a, \ldots, a-(n-1)$, where a is the inverted foreign address. The solution is obviously to subtract $n - 1$ from the inverted address, which can usually be done by giving an offset to the memory access instruction and hence requires no extra instruction.

Most operating systems reserve some portion of an application's address space for their use, usually at the top of the address space. It is quite common, for example, for an application only having control over the lower 2 GB of its address space on 32 bit machines. In such a case a solution would be to invert all bits of foreign addresses but the most significant one. If one register can be spared for holding the required bitmask, this only costs one XOR instruction.

5.3 Address Munging

On some target machines there is a solution available more efficient than the two discussed above. Assume we want to emulate a 32 bit system. There are two observation we are very likely to make (even when emulating the i386). First, 32 bit memory accesses are much more frequent than 16 bit and 8 bit ones. Second, there are very few, if any, unaligned memory accesses[5].

If we had the luck of emulating a machine where all memory accesses were aligned and for 32 bit words, we had to do nothing at all and could store the words in the native byte order. The application had no way of knowing we were storing the words in a different byte order.

[5] Accessing an n byte quantity at address a is an aligned access if a is a multiple of n.

The situation is more difficult when 16 bit (aligned) and 8 bit accesses are allowed. Fortunately, all we have to do to make 16 bit accesses access the correct half of a 32 bit word, is make it access the other half than it would access with a given foreign address without our intervention. This can be accomplished by inverting the second least significant bit of the address. The same technique can be applied to 8 bit addresses: Inverting the two least significant bits of an address is sufficient.

This leaves us with the case of unaligned addresses. Assume we are emulating a big-endian machine on a little-endian machine with the technique just described. After storing `0x00010203` at address `0` and `0x04050607` at address `4`, the application assumes the memory contents to be thus:

00	01	02	03	04	05	06	07

while in fact they are

03	02	01	00	07	06	05	04

A load at address `2` would be expected to fetch the word `0x02030405`, but we would load `0x06070001`. Somehow, we need to catch unaligned memory accesses.

Fortunately, some machines, like the Alphas do not handle unaligned memory accesses at all. Instead, they treat them as exceptions and let the software handle them. On such machines, it is easy to catch such an exception, examine the instruction that caused it, and fix up the memory access, i.e., perform the correct operation and resume execution of the code.

6 Address Space

Unless the dynamic binary translator is part of the operating system kernel, it seems that both the foreign application and the binary translator must somehow coexist in the same address space. This is no problem if the native platform has a larger address space than the foreign platform, for example when emulating a 32 bit platform on a 64 bit one. However, it has to be dealt with if the address spaces have the same size.

A simple but for most applications sufficient solution is to look for some part in the foreign address space which is usually not used. The binary translator can be compiled and linked to map itself to that region of memory. It is likely that the foreign and the native stacks collide, so the stack will usually have to be relocated as well, which is a simple matter. Additionally, the binary translator will have to take care that the foreign application does not meddle with the translator's memory region. This will usually involve forbidding write access to the pages in that region as well as designing the system call handler in such a way that attempts to somehow change those pages or their memory protection will fail.

By using two processes—one for the translator and one for the foreign application—the binary translator's data and code can exist in a different address space than the foreign application's. A similar approach is used by User Mode Linux [2] to give the user mode kernel and it's processes different address spaces. There is one problem, however, which diminishes this technique's attractiveness: Even if the binary translator existed in a separate process, the native code generated by it must be in the address space of the foreign application.

There is a solution to even this problem, albeit requiring a more difficult strategy. The translator can be designed so as to generate position independent code. Then, whenever the foreign application needs access to the region occupied by the translation cache, it can be relocated to another free region. Since it is highly unlikely that the whole address space is used by the foreign application, this solution can handle all situations occuring in practice.

There is one caveat, however, to using two separate processes. Whenever the generated native code must transfer control to the dynamic binary translator, two context switches must take place (one from the foreign application process to the operating system kernel and one from the kernel to the binary translator process). Context switches are expensive, hence such control transfer should be avoided.

6.1 Page Sizes

The foreign platform may have a different page size than the native platform. This can lead to problems if the foreign application's workings depend on the page size.

Page sizes are almost always powers of two, so two different cases can arise.

If the native page size is smaller than the foreign page size, there is no problem. The binary translator can simply treat a number of native pages as a unit.

If it is the other way around, the following problem can arise. Assume that the foreign pages f_1 and f_2 share the same native page n. The foreign application may now request that f_1 and f_2 have two different memory protection masks. For example, it could request that f_1 be write-enabled, while f_2 be write-protected. The page n, however, can only either be write-enabled or write-protected as a whole.

Two solutions exist to this problem. The first one is very simple, but assumes that the foreign application is well-behaved and does not actually violate memory protection. It simply protects a native page with the bitwise conjunction of the protection masks of the corresponding foreign pages. In our example, page n would be write-enabled.

If that approach can not be used, i.e., if the foreign application is not well-behaved, a native page must be protected with the bitwise disjunction of the protection masks of its foreign pages. In our example, page n would be write-protected. In that case, however, whenever the foreign application tried to write to page f_1, which is a legal operation, the binary translator would violate memory protection, and hence be delivered a `SIGSEGV` signal. The signal handler would have to check whether the access causing the signal was legal, and if so, would have to carry out the access by itself. That would require changing memory protection first to enable the access, and then changing it back. All in all, such an access would involve three kernel traps. This makes it obvious that this strategy, although correct, is slow.

7 Register Mapping

Different architectures have different register sets. A binary translator has to somehow map the foreign architecture's registers to the native architecture's. This section first describes the register sets of three different architectures and then presents several solutions to the problem of register mapping.

7.1 Register Sets

Let us begin with the most widely used architecture—the i386. This architecture has 8 general purpose registers. Most of these registers additionally have some special purpose. The `esp` register, for example, is used by the instructions `push` and `pop`, acting as a stack pointer. The `ecx` register is used as a counter in the repeat instructions, and registers `eax` and `edx` serve a special role in the multiplication and division instructions. This list is not exhaustive. A special purpose register called `eflags` holds some flag bits, most importantly the status bits `cf`, `zf`, `sf`, and `of`, which are discussed in section 4.2. The i386's floating point stack will not be discussed here.

The Alpha, in contrast, has a very simple register set. It has 31 general purpose integer and 31 general purpose floating point registers. Apart from the floating point status register, it has no user-visible special purpose registers.

The PowerPC has 32 general purpose integer registers, 32 general purpose floating point registers, and five special purpose registers. The link register holds the return address after subroutine call instructions (called "branch and link"). The counter register can be used in conjunction with "decrement counter and branch" instructions to implement fast loops. The integer exception register holds several status bits for integer operations, as does the floating point condition register for floating point operations. Finally, the condition register holds results of comparisons and is discussed in section 4.2.

7.2 Direct Mapping

Register mapping is trivial when the native architecture provides more general purpose registers than the foreign architecture has registers. Foreign registers can simply be statically mapped to fixed native registers for the whole run time. This is usually the best approach to take whenever possible.

7.3 Registers in Memory

The other extreme is to not statically assign any register, but to keep registers in memory by default. Within fragments, they are loaded into native registers when needed and stored back to memory at the end of the fragment. Within fragments, therefore, standard local register allocation techniques can be used. If the native architecture supports memory operands, like the i386, it may sometimes not be necessary to explicitly load and store a foreign register, since its memory location can be given as an operand to a native instruction

The problem with keeping registers in memory is that because registers are only allocated locally in fragments, large numbers of loads and stores have to be executed to get the values of foreign registers into and out memory. This can become a bottleneck.

7.4 Hybrid Approach

In some circumstances, a hybrid approach may prove beneficial. Some foreign registers are statically assigned to native registers while the others are locally allocated in fragments as described in the previous section. Obviously, the registers chosen for static allocation should be those used most frequently in foreign applications.

Unfortunately, not much research has been done in the area of register allocation for dynamic binary translators, so there are few results demonstrating the usefulness—or lack thereof—of a hybrid approach. One project [10] reports having used this technique, resulting in good performance. Few details are given, however.

7.5 Allocating between Fragments

The main problem with keeping registers in memory by default is that they must be loaded anew in each fragment and stored at the end if they have been modified. A better solution would be to carry over a register allocation from the end of one fragment to the start of the next one, to save loading and storing registers which are used by both fragments. In some way, this would be a global register allocation scheme acting only on local knowledge, since a dynamic binary translator does not usually maintain a control flow graph and live ranges.

Unfortunately, we know of no work in this direction. Future research will show if and how this can be made to work.

8 Self-Modifying Code

Programs which modify parts of itself present a special problem for binary translators. The problem arises when code which has already been translated is changed by the foreign program. If the binary translator is not aware of this change, it will execute the native code for the previously translated, old version of the foreign code instead of the modified one.

Self-modifying code occurs in programs which dynamically generate code, like just-in-time compilers and, of course, dynamic binary translators. A special case of self-modifying code can occur as a side-effect of using shared libraries. A shared library may be loaded dynamically at a certain address. After a while, it may be unloaded and after that, a different shared library could be loaded at the same address. That address would now contain different code than before. Such cases can arise in applications with plug-in systems.

8.1 Identifying Modified Code

Depending on the foreign architecture and on the required accuracy of identification, recognizing modified code can be very simple to moderately complex.

Most architectures provide instructions for informing the processor about changed code. This is necessary because modern processors usually have separate instruction and data caches which may not be kept in synch automatically. Even on machines which do keep them synchronized, the processor may have fetched an instruction which was modified subsequently.

The Alpha, for example, implements a PAL code[6] which must be called whenever code has been modified. It applies to the whole memory contents.

The SPARC takes a more fine-grained approach. It provides an instruction informing the processor that an 8 byte block of memory has been modified and may contain instructions to be executed. This instruction usually has to be called several times to ensure that all modified code is announced.

Due to its history, the i386 architecture does not provide any such instruction. Instead, the processor has to identify modified code by itself.

[6] PAL codes can be thought of as instructions which are handled in software.

Obviously, recognizing that code has been modified is easy for foreign architectures which require that special instructions be called under such circumstances.

For foreign architectures like the i386, identifying modified code is usually done by write-protecting all pages containing code which has already been translated. Whenever such a page is written to, the binary translator traps to a signal handler which can, given the address to be written to, identify the page with the to-be-modified code. It is usually not worthwhile to use a finer granularity than pages. This approach is also worthwhile for architectures which provide no fine-grain synchronization instructions, like the Alpha, when such fine-grain information is needed.

8.2 Adapting to Modified Code

Given that we know that code has been modified, and possibly even have a range for the modified code (e.g., a page), what do we do about it?

Clearly, the simplest solution is to invalidate all generated native code and start from scratch. This is also the only viable solution if we cannot identify which code has been modified and hence must assume that all code was modified.

A more sensible implementation would only invalidate native code corresponding to foreign code lying within the determined range. In the easiest case all that is necessary is to remove the entries for the fragments in question from the fragment look-up table.

On systems which patch direct jumps (see section 3.2), this is not sufficient, however, because code jumping to the obsolete fragments will still jump there. There are two solutions to this problem. First, we could patch the patched jumps back to their original form, i.e., to calls to the dispatcher. This requires knowledge about all jumps in the native code to a given foreign address. Essentially this means maintaining a come-from table, containing all addresses of jumps to a given target address. If such a table cannot be maintained, or if the overhead is too large, a better solution is available. Instead of redirecting all jumps to the invalidated fragments, we can change the code of the fragment itself (which is no longer relevant anyway) to simply call the dispatcher. That means that a jump to the modified code, once it is translated again, involves two native jumps (first to the obsolete fragment code and from there to the new native code) when coming from code which was translated before the code was modified. In practice, this is usually not a big problem, though. Note however, that in special cases this strategy might lead to a chain of jumps through obsolete native code. Malevolent foreign code could use this to slow down a dynamic binary translator.

9 `bintrans`

`bintrans` is a dynamic binary translator developed by the author at the Vienna University of Technology. Its primary aim is to serve as a vehicle for research in dynamic binary translation, with a priority on supporting multiple foreign-native combinations. Already, three such combinations are supported:

- PowerPC to Alpha
- i386 to PowerPC
- i386 to Alpha

The best supported combination is currently PowerPC to Alpha, but i386 to PowerPC is quickly gaining ground. This section discusses some of the design points of the `bintrans` system.

First of all, `bintrans` is an application-level binary translator. Currently, the only supported foreign system call interface is Linux's, but adding others is mostly simple, though tedious, work. An effort is under way, however, to automate most of the tasks involved in implementing a system call handler.

The unit of translation in `bintrans` is usually a sequential block of instructions, ending with a jump. We have also implemented a method which translates traces, very similar to Dynamo. It currently only works with the PowerPC to Alpha translator, however.

We have experimented with automatically generating instruction selectors [6]. For regular instructions, this approach works fairly well. Unfortunately, it does a bad job dealing with conditions and status bits. Two of the three instruction selectors are currently hand-coded. It turned out that writing an instruction selector is not as much work as one would expect. A fairly complete instruction selector can be implemented in about a month, if there are no unusual problems.

Nonetheless, machine descriptions, which describe instruction formats and instruction semantics, are still very important in `bintrans`. We have developed several programs (or rather, Lisp functions) which read machine descriptions and produce code operating on the described instruction set. One such program is an interpreter generator. Given a machine description, it generates an interpreter for the described instruction set. Such an interpreter can be used to get the infrastructure running (for example, the system call handler) before the instruction selector is implemented. Furthermore, it serves as an indispensable debugging aid. `bintrans` has a mode of operation which runs a program concurrently under the interpreter and with native code generated by the instruction selector. After each executed fragment, the state of both virtual machines is compared, and if there is a difference, the process stops. The error can then usually be found in the fragment translated most recently. Other things generated from machine descriptions are disassemblers, composers (macros which assemble machine instructions, given the operands), and liveness analyzers.

Byte order is dealt with in `bintrans` depending on the native platform. On the PowerPC, we simply swap bytes when accessing memory, which is very efficient, since the PowerPC has load and store instructions which do that implicitly. On the Alpha, we use address munging. For unaligned accesses, `bintrans` receives a `SIGBUS` signal and fixes up the access.

Register allocation depends on the foreign-native combination. The only combination supported by `bintrans` which does not use direct mapping is PowerPC to Alpha. For this combination, we keep registers in memory and do simple register allocation within fragments. We have also experimented with register allocation between fragments, but have not achieved significant speed-up yet.

9.1 Availability

The `bintrans` system will soon be available under the GNU General Public Licence from `http://www.complang.tuwien.ac.at/schani/bintrans`. Contributors are very welcome.

10 Acknowledgements

I thank Eric Traut, from whom I have learned a few things about dynamic binary translation, and Andi Krall for supporting me and for his comments on an early draft of this paper.

References

[1] Bochs. `http://bochs.sourceforge.net/`.

[2] User mode linux. `http://user-mode-linux.sourceforge.net/`.

[3] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[4] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banerjia. Dynamo: A transparent dynamic optimization system. In *SIGPLAN '00 Conference on Programming Language Design and Implementation*, pages 1–12, 2000.

[5] Connectix Corporation. Virtual PC for Mac. `http://www.connectix.com/products/vpc5m.html`.

[6] Mark Probst. Fast machine-adaptable dynamic binary translation. In *Proceedings of the Workshop on Binary Translation 2001*, September 2001.

[7] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Communications of the ACM*, 36(2):69–81, February 1993.

[8] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. In *Proceedings of the ACM Sigplan Workshop on Dynamic and Adaptive Compilation and Optimization (DYNAMO-00)*, volume 35.7 of *ACM SIGPLAN NOTICES*, pages 41–51, N.Y., January 18–18 2000. ACM Press.

[9] VMWare Inc. VMWare. `http://www.vmware.com/`.

[10] Jianwen Zhu and Daniel D. Gajski. A retargetable, ultra-fast instruction set simulator. In *Proceedings of the Design Automation and Test Conference In Europe*, 1999.