Dynamic Binary Translation and Hypervisors

Anoop Karollil
Department of Computer Science
University of British Columbia, Vancouver, Canada
karolli@cs.ubc.ca

1. Introduction

Dynamic binary translation is the process of converting an executable binary compiled for a particular ISA (say PowePC) into an executable binary for a different ISA (say x86) at runtime of the executable. This enables transparent execution of a binary compiled for a particular architecture on a different architecture, thus breaking the binding of an executable to a particular machine. This is done by virtualizing the source ISA and in the case of system level binary translators, virtualizing the whole platform including system code, boot code and BIOS.

Virtualization using hypervisors is the current technique for hardware multiplexing. A Virtual Machine Monitor (VMM) / hypervisor creates virtual machines that can be used to run multiple operating systems. This report tries to explore the possibilities of integrating virtualization as offered by current hypervisors with the virtualization in binary translation so that the combined system provides not only multiple OS on single machine capabilities but also multiple architecture based OSes on the same machine/architecture. A single high performance VLIW architecture based server machine can then host multiple operating systems for multiple architectures thus allowing for true application execution transparency. The report is organized as follows: Section 2 provides background for binary translation, section 3 explains a few key projects in binary translation, section 4 explains virtualization, virtual machines, and the possibilities of hypervisors with binary translation and section 5 concludes.

2. Background

Binary translation is the conversion of a binary executable file compiled for one platform (called the source architecture) to another platform (called the target architecture). The source might be a legacy architecture binary that needs to be run on the latest architecture, or maybe a program that was built for a particular architecture but needs to be run on a different one or the source architecture might even be the same as the target architecture with the binary being better optimized by the binary translator. The following subsections give an overview of the benefits, the types and the problems associated with binary translation, with a focus on dynamic binary translation.

2.1 Benefits

The benefits offered by binary translation are many. One important benefit is the decoupling of the binary executable from the underlying architecture. This decoupling is necessary for truly innovative ISAs as the necessity for backward compatibility is removed by dynamically translating legacy binaries to the new ISA. The decoupling in an ideal binary translator could well provide true execution transparency – write once, execute anywhere of any binary.

The second benefit of binary translation is a side effect of the translation process. When translating a binary executable from one ISA to another, it is possible to further optimize the executable. Certain static compiler optimization limits can be overcome when the optimization is done at run-time as dynamic optimization can cross boundaries such as indirect calls, function returns, shared libraries and system calls. Profiling can be done dynamically and often-executed traces of code identified and specifically optimized – across function calls, loops etc.

A third benefit is reduced hardware complexity. The architecture implemented in hardware can be simple with the translation to it and associated optimization done in software. Since the translation and optimization results are cached in memory, the use of hardware logic is brought down which saves power.

2.2 Types of Binary Translation

The process of binary translation is usually done by either using an emulator, a static translator or a dynamic translator.

Emulators are easy to implement as they just interpret each instruction in a legacy program at runtime without caching of interpreted instructions. Since this is done an instruction at a time and with no caching, the performance of an emulated legacy application is not too good.

Static translation (e.g. UBQT [3]) is done offline and since there is no overhead on runtime performance in static translation, more rigorous optimizations can be performed. Static translation can also use profiling information from previous runs of the executable.

Dynamic translation is done at runtime. It is generally done in stages, an interpretive stage which produces a basic translated trace of the source ISA instruction block being executed, in the target ISA format, and an optimizing stage where frequently executed parts of the binary (termed as Hotspots/Hot code etc) are optimized. Profiling here might be done either during the interpreting stage [2, 5, 9] or during execution of the translated non-optimized code [8]. The latter has the advantage that it provides more precise information for later stages as the translated code can run for longer time than interpreted code and still maintain low overhead.

Binary translation can also be classified according to the level in the system at which the translation is done. Some systems [2, 3, 8] provide translation at the user-level – user application binaries from one platform are converted into binaries that can be executed on another platform ([2] actually converts binaries between the same platform but optimizes them). Other systems like Daisy [5] and Crusoe [9] emulate the whole machine and so they run below the OS in the platform. Daisy supports the PowerPC architecture (and touts generality for emulating S/390 and x86 too) while Crusoe supports binary translation for x86.

Binary translation systems can also be platform dependent or independent. UQBT [3] is a framework for providing binary translation which is both re-targetable as well as re-sourceable – it can translate binaries from multiple architectures running multi-platform OSes like Solaris or Linux to multiple architectures and OSes (binaries in Solaris to binaries in Pentium and vice versa – statically). Dynamo [2] converts HPUX binaries from HPPA8000 to optimized HPUX binaries on HPPA8000. Daisy [5] converts PowerPC architecture binaries (OS, boot, BIOS code as well as apps) to VLIW binaries. Crusoe [9] converts x86 binaries (again OS, BIOS, boot code and apps) to VLIW binaries. IA32 EL [8] converts IA32 binaries to IPF (Itanium PlatForm) binaries.

2.3 Problems faced in binary translation

The crucial problem that needs to be handled in dynamic binary translation is that of performance. The translation is performed at runtime and hence there is obviously some overhead. But most of the performance overhead is amortized by optimizing the translated binary and caching translations of the instruction sequences that are frequently executed. The amortization comes into effect only if the working set of the translated binary (which is cached) is small and fits in the cache.

A second problem is that of precise exception/signal handling. The translated binary instructions do not correspond one to one with the source binary instructions. In x86 ISA, exceptions need to be precise – when one instruction causes an exception, all instructions before it should have completed execution and all instructions after it should not get executed. When the instructions are translated into the target ISA and an exception occurs when executing target ISA binary, it is hard to pinpoint which instruction in the source binary caused the exception and thus providing a context for an exception/signal handler in the source architecture is tricky.

A third problem is self-modifying and self-referential code. The problem of self modifying code is both local to a particular binary (which changes with execution) as well as global in the case of a system level binary translator (where a file is modified/deleted by the OS). The problem is that of translation cache maintenance – if the source architecture binary changes or is deleted, the translated binary in the cache should be updated as well. In the case of programs that reference themselves, to say create a CRC, the binary translator usually leaves a copy of the original translated code that the translated binary can refer to.

A fourth problem is that of memory and address translation. A binary translation system that is below the OS should take care of virtual addresses and page faults. Memory mapped I/O addresses also have to be handled separately as they should not be cached. The number of registers in the new architecture should also be optimally greater than that in the source architecture – or they will have to be emulated in physical memory which will greatly affect load and store performance. Status registers like flag registers also need to be suitably represented in the new architecture to mirror the source architecture usage.

A fifth problem is management of the translation cache. The translation cache stores the optimized translated target architecture binary blocks or fragments that form the working set of an executable. The size of the cache is limited and hence there should be a good replacement policy.

A sixth problem is that of real time code with constraints on the execution time. Dynamic binary translation leads to variation in the execution time of a binary depending on whether the translated target code is present in the translation cache or not. If it's not present in the cache, the execution may even stall.

Finally the binary translation system should be reliable and correct in its working. All optimizations should be only done if it conserves the correctness of the source binary – correctness and performance are usually tradeoffs.

3 Binary translation projects

3.1 HP Dynamo

Dynamo is a dynamic optimizer that uses binary translation to optimize HPUX binaries on the HP PA8000 architecture. Thus the source and target architecture for the translation are the same – the HP PA8000. The translator runs in user space over the HPUX OS and is implemented as a shared dynamic library which applications link to. Thus the system is not transparent with the application that needs to be optimized having to call a function in the Dynamo library that hands over execution control to Dynamo. But the basic process of binary translation, even though it's for optimization alone, is the same as in the other dynamic translation projects described later in the report.

The Dynamo system consists of an interpreter, an optimizer and a fragment cache. The interpreter interprets the source binary until a 'hot' instruction sequence called a 'trace' is identified. The trace is a sequence of instructions that are executed very frequently and all the 'hot' traces taken together form the working set of the binary executable. When such a trace is identified, the optimizer kicks in and generates an optimized version of the trace called a 'fragment' which is stored in a fragment cache. The 'hot' trace is identified based on a 'start-of-trace' instruction which is at the target address of a backward taken branch (which is usually a loop header) or an exit branch target address from an optimized fragment. The 'start-of-trace' instructions are profiled and when they are encountered a given number of times, they signify the start of a 'hot' trace. All instructions that follow the 'start-of-trace' instruction for an identified hot trace are stored in a trace buffer until an 'end-of-trace' signifying instruction is interpreted. This is another backward taken branch or a branch to an instruction already present in the fragment cache or a fixed number of instructions after the 'start-of-trace' instruction. The new fragment added to the fragment cache is linked to other fragments by a fragment linker. As execution of the source binary proceeds, the parts of it that are often executed become 'hot' and have corresponding translated fragments in the cache. Dynamo first checks the fragment cache when interpreting a branch and thus with time, it executes the fragments in the cache alone. As these are linked together, interpretation is reduced quite significantly and the optimized fragments form the major part of binary execution leading to a performance boost.

Optimization is done through branch fall through to within fragment and the end result is a single entry, multi-exit sequence of instructions with no internal control join points. The fragments traverse indirect branches, function calls and returns and virtual function calls. The optimizer then eliminates unconditional branches, does copy propagation, constant propagation, strength reduction, loop invariant code motion and loop unrolling all in 2 passes aided by the straight single path fragment.

The problem of translation cache management is handled in Dynamo by a pre-emptive complete fragment cache flush using a heuristic based on the changes to the binary executable's active set (radical changes lead to a cache flush).

The problem of precise exception/signal handling is taken care of by Dynamo by it intercepting all signals and executing the program's signal handler code in the same manner in which it executes the rest of the application. Asynchronous signals are queued till end of a fragment – which forms the boundary for optimized code. Once outside the fragment, the application is being interpreted and thus precise signal handling is possible. In the case of synchronous signals or exceptions, a conservative trace optimization option is used by Dynamo to re-construct the source binary signal context and step through instructions in the source binary to pinpoint exception cause.

3.2 UQBT (University of Queensland Binary Translator)

UQBT is a static binary translation framework which is built for making re-sourceable and re-targetable static binary translators. It is re-sourceable as in it can build translators for binaries built for different architectures and it is re-targetable as in it can build translators for multiple target architectures. It does all this by separating machine dependent and machine independent concerns. The translation is at the user level and it has several re-usable components that a binary translator builder can use to build a custom binary translator for a particular platform.

The translation starts off with the conversion of the source binary to a source Register Transfer List (RTL) that describes the source machine instructions effect on the source architecture's hardware registers. This is done with the help of a Specification Language for Encoding and Decoding (SLED) that translates a binary stream (the contents of which can be determined based on the binary file format) into the corresponding assembly instruction. A semantic mapper is then used to map the assembly instructions to the source architecture RTL based on specifications in a Semantic Specification Language (SSL).

The source machine RTL is then converted to a machine independent Higher level Register Transfer Language (HRTL). Instructions whose semantics are not fully captured by RTLs like control transfer instructions are mapped by a Control Transfer Language (CTL) to HRTL's high-level control-transfer instructions. An operating system's application binary interface calling convention is specified in a Procedure Abstraction Language which is used to define parameter locations, return value locations and local variable locations in the

HRTL for the source binary. The HRTL is independent of the underlying source architecture and it allows binary translation specific optimizations to be included in the UQBT framework.

The HRTL thus derived is then converted into C and assembly files using a portable optimizer. This can then be compiled using the target architecture's optimizing C compiler and assembler.

Static binary translators have to handle portions of non-translated code in the target binary. This is usually done by having an emulation environment that helps out the translated binary or by having interpreter hooks in the translated binary. The hooks are used for those portions of the binary that have not been translated and the interpreter uses the source to target address mappings which are stored in the target binary along with a copy of the source binary to handle other problems that were described in section 2. Aliasing problems are handled by having a copy of the data sections in the target binary and a link-map file which retains the virtual address mappings of the source binary.

3.3 Transmeta Crusoe

The Crusoe system provides whole system level dynamic translation for the x86 platform. The binaries that run on x86 including OSes and applications are translated dynamically into binaries for a simple VLIW target architecture. In the case of binary translation at the whole system level, the translator (or Virtual Machine Monitor), has to faithfully translate the lowest level code controlling the source architecture like the boot code, BIOS and memory controller code.

The Crusoe system VMM/translator, called a Code Morphing Software (CMS), does the translation dynamically and is the interface between the source x86 and the target VLIW architectures. Translation and optimization are similar in method to what is done in Dynamo except that Crusoe translates from x86 to VLIW. The CMS interprets x86 instructions, creates optimized VLIW translation traces for frequently executed sequences of instructions and caches these in a translation cache. The CMS profiles the interpreted code and uses heuristics to either interpret, or provide translation without optimization or translation with heavy optimization for the target VLIW instruction sequences.

The precise exception problem is handled in Crusoe by using working and shadow sets of registers for holding x86 state. The translated VLIW instructions modify only the working register set until the end of a translated block is reached. At this point the working set register contents are 'committed' by copying their values into the corresponding shadow registers. If an exception occurs when a translated VLIW block is being executed, the working set register values are overwritten with the values from the shadow registers and the CMS does a step by step interpretation of the source x86 instructions to reach the point of exception. Memory changes are rolled back by buffering stores in a gated store buffer that also gets committed to memory when a translated block is completely executed.

The CMS converts x86 instructions into VLIW instructions called molecules which might have up to 4 sub-instructions called atoms corresponding to x86 instructions. These atoms are re-ordered in the molecules for optimization and the molecules themselves might be re-ordered. Problems of aliasing arising due to this re-ordering are handled in hardware using modified load and store operations that raise an exception in the case of improper aliasing.

The self modifying code problem is solved in Crusoe by write protecting the page of x86 memory containing the block of code that needs to be translated. This is done invisible to the x86 source architecture. If during the execution of the translated binary, the protected page is written to, an exception is raised and the associated translated block in cache is invalidated.

3.4 IBM Daisy

The IBM Daisy system is in many ways similar to Transmeta Crusoe. It provides dynamic binary translation from source PowerPC architecture binaries to target VLIW architecture binaries. Like Crusoe, it also has its own translation of the BIOS, boot and memory controller code.

Binary translation mechanics are quite similar to that in Crusoe and Dynamo with the Daisy VMM interpreting PowerPC instructions, profiling the instructions to figure out the most frequently executed sequence of instructions ('trace' in Dynamo), translating those into optimized VLIW instruction sequences and caching the translations. But unlike Dynamo with single path 'fragments', Daisy allows multiple paths in its optimized translated instruction sequence, which provides more opportunities for parallelism and lessens dependence on good branch prediction.

The precise exception problem is handled in Daisy by the VMM which like Dynamo handles the execution of the program exception handler as it does the rest of the program. It traces the source architecture instruction that caused the exception using a Virtual Page Address register that contains the address of the current page in the source architecture binary code. The offset of a source instruction corresponding to the start of a VLIW instruction is stored with each VLIW instruction as a NOP or a table is maintained that maps the VLIW instruction to the start of a particular source instruction sequence which might need to be executed sequentially to trigger the exception. The VMM then performs the interrupt actions that are required by the source architecture and finally it branches to the translated system code that handles the exception.

Aliasing is handled using exception tags on registers and a load-verify operation before loads. Self-referential code is trivially handled as all the registers corresponding to the source architecture in the VLIW architecture contain the values that they would have contained if the binary was executing in the source architecture. Self modifying code is handled in the same way as is done in Crusoe with a read-only bit for units of source architecture binary code. Modification of any of the units leads to invalidation of the translation for the particular unit in the translation cache.

3.5 IA-32 Execution Layer (EL)

The IA-32 EL provides dynamic binary translation of IA-32 binaries to IPF (Itanium2) binaries. The EL is more like Dynamo rather than Daisy or Crusoe as it provides translation at the user level and is specifically designed for native Itanium based OSes which includes Windows and Linux.

The OSes are configured so that the IA-32 EL becomes the execution vehicle for IA-32 applications and thus translation is transparent. When initialized IA32 EL gets control from the OS to execute IA-32 applications within the same address space. Interactions with the OS to acquire system resources, executing system calls, signal handling, exceptions etc are handled by a small BTLib abstraction layer that is the only OS dependent component in IA-32 EL. The rest of the translator is made up a major OS independent component called BTGeneric that uses BTLib to handle OS interactions required by the IA-32 application. The BTLib provides a 2 way interface with the OS. Memory allocation request go from BTGeneric to BTLib to OS while exceptions/signals go from OS to BTLib to BTGeneric.

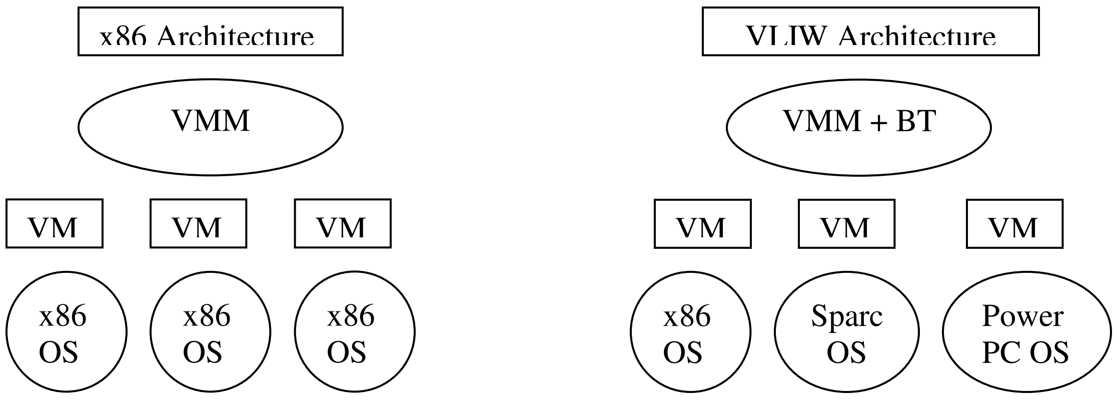
The binary translation mechanism is again similar to the dynamic translation mechanisms described before. There are 2 phases – cold code translation phase and hot code translation phase which correspond to interpretation cum basic translation and optimization phases. The cold code translation phase involves decoding IA-32 code and generating IPF code for a few basic blocks using hand coded optimized templates. Instrumentation for profiling is done at this stage rather than at interpretation as in Dynamo, Crusoe and Daisy. Translated blocks are linked together and hence execution jumps from one block to another.

Hot code translation begins when the threshold count for execution of blocks leads to enough of them being registered as hot or when one block registers twice. A trace based on the profiling done in cold translation phase is selected which might involve multiple paths as in Daisy. The IA-32 code corresponding to this trace is then re-translated to a highly optimized Intermediate Language data structure. A scheduler then re-orders the IL structures, builds recovery information for exception handling and is placed in the translation cache and linked to its predecessors.

Precise exception handling is done by the IA-32 EL layer by building the source IA-32 register state based on the target IPF register state. The IA-32 exception handler is then simulated to handle the exception. In the case of an exception when executing cold code translations, the translation from IA-32 to IPF instruction is such that IA-32 state is committed only after the last IPF instruction that can cause a fault is executed so that rollback is possible. In the case of an exception when executing hot code translations, re-ordering of the source IA-32 instructions into IPF instructions makes pinpointing the exception source hard. This is simplified by using commit points which are barriers that have a fixed number of associated potential faulty instructions. Re-ordering is limited to instructions belonging to the same commit point and hence the source of exception is narrowed down.

4 Virtualization, virtual machine monitors and binary translation

Virtualization with hypervisors is the current trend in



Current x86 virtualization

Current virtualization + Binary Translation

Figure 1

machine multiplexing. It could be considered as taking multitasking to the next level – multitasking operating systems rather than processes, with the hardware switched/shared between the multiple operating systems. Virtualization using hypervisors multiplexes hardware by creating virtual machines that emulate the underlying hardware for multiple operating systems all hosted on the same machine. But this virtualization using hypervisors is currently limited to multiple virtual machines of the same architecture on a single machine of a particular architecture. For e.g. Xen provides virtualization for the x86 platform – it creates multiple x86 virtual machines that can each host an x86 OS on an x86 based machine. [4] talks about extending the virtualization to include architecture virtualization too. This will allow for the same machine to not only support multiple operating systems but multiple operating systems for multiple architectures on the same single architecture as modeled (VLIW) in Fig 1.

The VMM in Fig. 1 is similar to the VMMs described in Daisy [5] and Crusoe [9] – the similarity being that both provide a layer of abstraction of the hardware – the former with the purpose of multiplexing hardware to support multiple OSes for more efficient resource utilization and fault isolation and the latter with the purpose of providing virtualization of an architecture different from the underlying hardware, using binary translation. The complete system level binary translation by Daisy and Crusoe VMMs involves the same challenges as in the full virtualization provided by virtual machine suites like VMware.

For a virtualization scheme to include binary translation, it should involve interpreting machine instructions which should then be translated. Hence only full virtualization schemes that interpret source architecture instructions can support binary translation and architecture transparency. Thus para-virtualization systems such as Xen [11] which provides a 'hyper-call' interface to OSes cannot easily virtualize multiple architectures on the same machine. The VMware workstation software executes source/host binary code natively on an x86 CPU except for kernel/boot code which are interpreted and translated. To support architectures other than x86 dynamically, VMware workstation would need to be modified to have a dynamic optimizing binary translator that translates from multiple architecture binaries to a binary that can execute on the target architecture that VMware is hosted on.

5 Conclusion

This report tries to provide an overview of binary translation followed by a survey of several types of binary translation systems, their similarities and differences. It then gives a brief exploration of the possibilities of expanding current hypervisor functionality to involve architecture transparency along with operating system multiplexing. Binary translation is definitely deeply tied with hypervisor virtualization. The commonalities between providing multiple virtual machines and providing support for multiple architectures with binary translation should be evident. Hence there should be good scope to integrate hypervisors with more wide binary translation capabilities. This requires more research into hypervisor technology.

References

[1] Eric R. Altman, David Kaeli, and Yaron Sheffer, *"Welcome to the opportunities of BinaryTranslation"*, IEEE Computer 33(3), March 2000.

[2] Vasanth Bala, Evelyn Duesterwald, and Sanjeev Banjeria, *"DYNAMO: A Transparent DynamicOptimization System"*, Programming Language Design and Implementation, June 2000.

[3] C. Cifuentes and M. Van Emmerik, *"UQBT:Adaptable Binary Translation at Low Cost",* Computer, Vol. 33, No.3, pp. 60-66, 2000.

[4] Eric R. Altman, Kemal Ebcioglu, Michael Gschwind and Sumedh Sathaye, *"Advances and Future Challenges in Binary Translation and Optimization"*, Proceedings of the IEEE Special Issue on Microprocessor Architecture and Compiler Technology, November 2001.

[5] Kemal Ebcioglu, Erik R. Altman**,** Michael Gschwind and Sumedh Sathaye, **"***Dynamic Binary Translation and Optimization"*, IEEE Transactions on Computers 50(6), June 2001.

[6] M. Gschwind, E. Altman, S. Sathaye, P. Ledak, and D. Appenzeller, "*Dynamic and Transparent Binary Translation"*, Computer, vol. 33, no. 3, pp. 54-59, Mar. 2000.

[7] Cifuentes, C.; Malhotra, V. "*Binary translation: static, dynamic, retargetable?*" Proceedings of the International Conference on Software Maintenance, November 1996

[8] Leonid Baraz, Tevi Devor, Orna Etzion, Shalom Goldenberg, Alex Skaletsky, Yun Wang and Yigal Zemach, "*IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium®-based systems*", Proceedings of the 36th International Symposium on Microarchitecture, 2003

[9] A. Klaiber, *"The Technology behind Crusoe Processors"*, Technical report, Transmeta Corp., Santa Clara, Calif., Jan. 2000.
[10] Shiliang Hu, James E. Smith, "*Reducing Startup Time in Co-Designed Virtual Machines*", Proceedings of the 33rd International Symposium on Computer Architecture, 2006
[11] Barham, P. Dragovic, B. Fraser, K. Hand, S. Harris, T. Ho, A. Neugebauer, R. Pratt, I. Warfield, A., *"Xen and the Art of Virtualization"*, Proceedings of the nineteenth ACM symposium on Operating systems principles, 2003