Computer Sciences Department

Intrinsic Compatibility in Process Virtual Machines

Nidhi Aggarwal
James E. Smith

Technical Report #1563

April 2006

UNIVERSITY OF WISCONSIN MADISON

INTRINSIC COMPATIBILITY IN PROCESS VIRTUAL MACHINES

Nidhi Aggarwal and James E. Smith

University of Wisconsin, Madison, Wisconsin, 53706
naggarwal@wisc.edu, jes@ece.wisc.edu

Abstract

A Process Virtual Machine (VM) provides an execution environment for a guest user application program on a host platform that may differ from the guest's native platform. Informally, virtualization is compatible if there is no observable difference between the execution of a program on its native platform and as a guest on a process VM.

We describe a general framework for modeling process VMs and discussing compatibility issues. A VM is intrinsically compatible if virtualization is compatible for all applications, under all conditions. Using the process VM framework we assert that an intrinsically compatible process VM must dynamically check for all exception conditions that are not checked with static analysis.

We then define an efficient process VM to be one where binary translation is used for emulation, and 1) dynamic exception checking is done implicitly by virtue of executing host instructions that result from binary translation, and 2) no spurious memory permission traps occur when binary translated code is executed. These properties are consistent with the way current process VMs like Dynamo, DynamoRIO, IA-32EL and Fx!32 are implemented. Based on the first of these properties we assert that an efficient VM, while emulating guest code, cannot access any read/write memory area beyond that corresponding to the guest's original read/write memory region. This implies that the host register file must at least be as big as the guest register file. The host ISA must also support an "execute only" permission for runtime pages. The second property leads to the assertions that the host virtual address space must be larger than the guest virtual address space, and the host permission types must be a superset of the guest permission types. Given the assertions, we discuss the implications for the construction of efficient, intrinsically compatible process VMs and dynamic binary optimizers.

1 Introduction

A Process Virtual Machine (VM) provides an execution environment for a user-level program binary (executing as a guest process) on a host platform. The host platform often differs from the native platform for which the guest program was originally developed, although this is not necessarily the case. A number of commercial and research process VMs have been developed (although they are often not explicitly called "VMs"). A recent example is the Intel IA32-EL [9] which enables execution of IA-32 application binaries on Intel's Itanium Processor Family [IPF].

A VM is intended to provide a compatible execution environment for guest software. Informally, we say that virtualization is *compatible* with native execution, or is *transparent* to the user, if all observable guest execution behavior is the same on the VM as it is on the guest's native platform. The abstraction level at which compatibility is supported depends on the type of virtual machine. For a process VM, compatibility is supported at the Application Binary Interface (ABI), which consists of the user portion of a particular Instruction Set Architecture (ISA) and system calls for a particular operating system (OS). A very well-known example is the IA-32/Windows ABI.

In practice, compatibility is not only a property of the VM; it may also depend on the guest software. We define a VM to be *intrinsically compatible* if it provides compatibility for all possible executions of all possible guest programs. A VM is *extrinsically compatible* if compatibility is guaranteed only for a subset of guest programs that satisfy certain conditions external to the VM implementation. Extrinsic compatibility may be acceptable for systems where there is a specific workload or class of programs that can be analyzed and certified for compatibility.

An intrinsically compatible VM has significant advantages to the user; in particular, the user can treat the VM exactly as a native platform without consideration of the software that is to be run on it. In contrast, if the VM is only extrinsically compatible, then the user must be aware of the requirements for reliable software execution. In some cases, these conditions may not be well-defined and/or the user must be very familiar with the internals of the application software. For example, it may be the case that only software constructed by certain compilers or using certain libraries can be reliably used. Or, the program may need to be free of certain types of bugs, such as accesses to out-of-range addresses. This may require either a sophisticated user or certification by the software developer that a given piece of software will run correctly on a given VM. In general, an intrinsically compatible VM can be deployed for use by a less sophisticated and broader set of users than a VM that offers only extrinsic compatibility of some form.

Although there are clear advantages to implementing intrinsically compatible process VMs, the ability to do so in an efficient manner depends on properties of both the host and the guest's native platform. In this report we examine the structure of process VM implementations and describe properties of the host and native instruction set

architectures (ISAs) that affect the ability to implement efficient, intrinsically compatible process VMs.

1.1 Approach and Summary of Results

When a guest process executes under the control of a process VM, a layer of runtime software encapsulates the guest process thereby giving it the same outward appearance as a single user-level host process as shown in Figure 1. The runtime software and the guest application share the same process virtual address space (VA) and registers. Sharing of these finite logical resources has significant impact on both the compatibility and efficiency of a process virtual machine implementation.

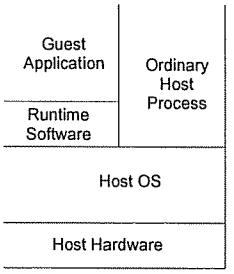

Figure 1. Runtime software and guest application appear as a single host process to host OS

VM compatibility and implementation efficiency are also affected by particular features of the ISA and OS. Relative sizes of the architected state and the properties of the host and the native architectures such as permission granularity and types of memory permissions affect the efficiency with which intrinsic compatibility can be achieved.

To enable clear delineation of specific features that permit construction of intrinsically compatible process VMs, we consider process VMs that employ a general implementation style. This implementation style is used in most (if not all) current and recent process VMs. This approach of implementation-based VM analysis is the same as the one taken by Popek and Goldberg [2] when analyzing conditions for implementing system level virtual machines. An important feature of the process VM implementations we consider is the use of binary translation and code caching. This feature is key for efficient (good performing) implementations, but, as we will show, it also leads to a

number of significant constraints for implementing intrinsically compatible VM implementations.

Using the process VM framework that we define in Section 2, we first observe that an intrinsically compatible VM must dynamically check for all trap conditions that are not checked via static analysis. Then, we put forward two properties for *efficient* intrinsic VM implementations: 1) dynamic exception checking should be done implicitly by virtue of executing host instructions that result from binary translation, and 2) no spurious memory permission traps occur when binary translated code is executed. Both of these properties are satisfied by all current, high performance process VM implementations including DynamoRIO [18], Dynamo [1], FX!32 [4], IA-32EL [9].

Based on the first of these properties we assert that an efficient VM, while emulating guest code, cannot access any read/write memory areas beyond those accessible by the guest's original data region. In particular, there cannot be a memory region to spill guest register values. This implies that the host register file must at least be as big as the guest register file. Another corollary is that the host ISA must also support "execute only" permission for runtime pages.

The second property for efficiency leads to additional assertions for implementing efficient and intrinsically compatible process VMs. In particular, we assert that the host virtual address space must be larger than the guest virtual address space, and the host permission types must be a superset of the guest permission types.

Dynamic binary optimizers such as Dynamo [1] [22], Mojo [11] and ADORE [8] have the same ISA for both the host and guest, and present some interesting challenges for implementing intrinsically compatible VMs. Mapping the ISA onto itself severely restricts the availability of virtual address space and registers. The implication, and a key result, is that it is extremely difficult (we are tempted to say it is practically impossible) for such optimizers to maintain intrinsic compatibility and still provide improved performance.

1.2 Related Work

A number of process virtual machines have been developed for translating from one ABI to another. The Digital/Compaq FX!32 allowed IA-32/Windows guest applications to run on Alpha/Windows hosts [4]. Wine [10] allows Windows

applications to run on Linux systems, where both systems use the running on IA-32 ISA. Apple's Rosetta and Transitive's QuickTime [16] allows PowerPC/Mac OS X binaries to run on IA-32/Mac OS X.

Dynamic binary optimization/code modification systems such as HP Dynamo [1] [22], DynamoRIO [17], Mojo [11] and ADORE [8], are an important class of process VMs that do not translate software from one ABI to another, rather they either optimize, instrument, or inspect application software at run time.

An interesting process VM variant is Omniware [15] which is a mobile code system for producing and executing mobile code in web applications. However, Omniware does not use existing binaries, and requires recompilation as a means for implementing efficient dynamic memory permission checking.

In contrast to system VMs [3] like VM/370 [5], VMware workstation [12], Xen [14] where efficiency in resource utilization is the key property, virtualization at the process level involves both resource (state) mapping and function transformation, with function transformation (at the ISA and OS interfaces) being the key property. Function transformation makes both efficiency and compatibility with the native system important issues in process VMs.

Co-designed VMs – Transmeta Crusoe [7] and Daisy [6] use similar VM implementation technology (based on binary translation), but they are implemented at the ISA level and support all guest software, including the OS. Here intrinsic compatibility is essential – the same standard for compatibility as a hardware implementation of an ISA. Gschwind et al. [21] also identify issues with compatibility and maintaining precise exceptions in a common execution platform for different ISAs. They propose pretesting for exception conditions before altering architected state as a solution.

Although compatibility has been a consideration for almost all the process VM implementations referenced above, we found that the DynamoRIO work [17] discusses compatibility issues most completely. DynamoRIO has a goal of being "transparent" where transparency means there is no change in the semantics or behavior of an unmodified program running under DynamoRIO; this is equivalent to our notion of compatibility. The DynamoRIO developers classify transparency based on resource usage conflicts, original application modification, and recreating native application state

when it has been changed. They also describe many of the challenges in performing OS and instruction emulation compatibly. The DynamoRIO approach and solutions to transparency are guided by their benchmarks and application usage scenarios and do not target intrinsic compatibility in a systematic way. In cases where the cost of transparency is too high, for example watching all memory loads for return address translation, they sacrifice transparency for efficiency. In contrast, we abstract away the compatibility problems due to OS emulation and focus on the ISA features of the host and guest that will inhibit efficient intrinsic compatibility. In a sense, ours is a bottom up approach because we do not require any knowledge of existing or potential application or features specific to a particular OS to guarantee compatibility. Furthermore, our analysis of compatibility applies to a general class of process VMs and not just dynamic code modification systems.

Gschwind et al. [18] also identify a compatibility problem that occurs when trap handlers expose otherwise dead state and present a solution to maintaining precise register state in the presence of aggressive optimizations in dynamic compilation systems. They annotate and patch the dynamically compiled code to jump to a fix-up routine before transferring control to a trap handler. They also point out that analyzing all applications and trap handlers *a priori* is not a practical option for full compatibility.

The methodology and motivation of our work is most similar to seminal work by Popek and Goldberg [2] for system VMs. They develop a model of third generation computers (which happen to still be with us today) and state and prove sufficient conditions for efficient virtualization for system VMs. The instructions that trap if executed from the user mode are called *privileged instructions*. All instructions that can modify or depend on physical state of the system are called *sensitive instructions*. If an instruction is not sensitive then it is *innocuous*. Popek and Goldberg assume all virtual machine to be *efficient* by definition and it is property of VMs that innocuous instructions are executed by the hardware directly with no intervention at all by the virtual machine monitor. The sufficient condition for implementing an efficient system virtual machine is that the set of sensitive instructions is a subset of the privileged instructions. As part of our research, we have constructed a formal framework similar to the one used by Popek

and Goldberg, and present conditions for compatible VM implementations in a assertion/argument format.

1.3 Report Overview

We describe the structure of process VM implementations in Section 2. We discuss compatibility issues and present design principles for intrinsically compatible process VMs in Section 3. Section 4 describes the necessary conditions to adhere to these design principles. Section 5 presents necessary conditions for efficient, intrinsically compatible VMs with no explicit analysis of exception conditions and for VM implementations that introduce no extra memory permission traps. We briefly discuss extrinsic compatibility in Section 6 and conclude in Section 7.

2 Process VM Implementations

The inherent complexity in virtualizing one platform on another and the multitude of approaches that can be employed to accomplish virtualization (with vastly varying degrees of performance and compatibility), makes it impossible to even consider all possible implementations, let alone analyze them. In order to make reasoning about VM implementations more tractable and to arrive at useful results, we consider the ways that commonly used process VMs are constructed and limit our discussion to that general class of VM implementations. Therefore we impose a certain structure to the implementation of process VMs and analyze that structure, similar to the way Popek and Goldberg do in defining "third generation computers" and system VMs. Of course, this limits the applicability of our results to the VMs that follow our structure and excludes other possible implementations (that might achieve intrinsic compatibility in other ways). Because the VM structure is representative of all the process VMs of which we are aware, we do not feel this is a severe limitation.

A process VM is implemented via runtime software, "the runtime", which includes an emulation engine, a code cache, and various side tables. Runtime initialization code allocates memory, initializes runtime data, and registers signal handlers (if the host OS supports them) for ABI-defined traps and signals. The emulation engine emulates the guest code using interpretation, binary translation, or a combination, and uses side tables to hold data that supports this process. Binary translated code is placed in a code cache

and is executed natively on the host processor; binary translation is essential to high performance (efficient) VM implementations. To improve efficiency, translated code blocks are chained together when branch targets are fixed, and a software jump table is used for translating indirect jump targets.

The guest application binary image, including code and data, serve as emulation input for the runtime software. The guest data image is mapped uncompressed to host state, and the original guest code image is also maintained to facilitate emulation (especially in the face of self-modifying code) and precise state materialization for traps. Runtime software maps guest virtual addresses to host virtual addresses; that is, the guest's address space is contained within the host process's address space. The host OS maps host virtual addresses to real addresses. Similarly, guest pages' permissions are mapped to host virtual and real pages' permission. The guest registers are mapped uncompressed to either the host registers or memory in the host VA space. We note that the simplifying assumption of uncompressed mapping of guest data image and registers is not unrealistic because of two reasons – the finite possibility that there can always be data that can not be compressed high enough and the potentially negative performance effect due to the time taken for compression and decompression.

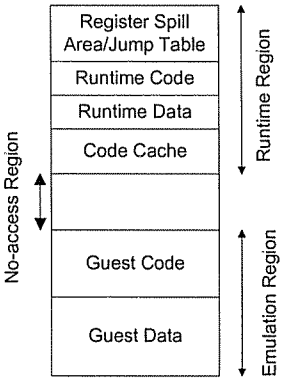

Figure 2. Address space mapping in a process VM

Process VM runtime software combined with the guest application appears as a single user process to the host platform. The mapped portion of the host process' VA space can be logically divided into two regions – a runtime region and a guest region (Figure 2). A region of the process address space assigned to runtime code, runtime data (e.g. a jump table for indirect branch linking), and the code cache is the *runtime region.* This region also contains any scratch space (e.g. a register spill area for memory mapped

registers) that the runtime software uses to store intermediate results during the emulation process. The region of the process address space mapped to the guest application code image and data is referred to as the *guest region.* The runtime region and guest region are not necessarily contiguous and their relative mappings within the host process virtual address space might change during execution. The VA space that is neither mapped to the runtime region or the guest region is non-accessible and is referred to as the *no-access region.*

As the virtual machine runs, its operation can be divided into two modes -- emulation mode and runtime mode. In *emulation mode* guest code is emulated, as the name suggests; this is either done via interpretation or through direct execution of translated code in the code cache. Emulation mode instructions result in state transformations in the guest process's memory area. In the *runtime mode* the runtime code performs supporting functions like collection of profile data and building and chaining hot code traces or superblocks [1], emulating traps, system calls etc. As it runs, the VM process switches back and forth between these two modes, much like a conventional system switches back and forth between user and system modes.

3 Compatibility

Two computing platforms are *compatible* for a given program, if all observable results from executing the program are the same on both platforms. The platforms are *intrinsically compatible* if they are compatible for all program binaries and for all input data. This implies that the execution of a program on an intrinsically compatible VM will be equivalent to execution on a correctly implemented native platform, for all possible programs, under all execution conditions, excluding purely performance differences. For example, compatibility must hold even for an assembly program designed to exploit the corner cases of an ISA, or programs that are attempts at intentionally breaking compatibility.

Compatibility is defined in terms of functional behavior and not performance. Performance is often an issue, however, and relaxing compatibility to a subset of programs is sometimes done in order to allow the implementation of a more efficient, better performing VM. If compatibility holds only for a subset of program binaries, we

refer to this as *extrinsic compatibility* because some additional conditions, external to the VM implementation, must hold in order for compatibility to be achieved. These conditions may include assurances that certain types of bugs are not present, that a certain compiler generated the binary, or only certain OS features are used, for example.

The standard for compatibility in most types of VMs is intrinsic compatibility because it frees the user from being concerned with which guest software will (or won't) give the same results when run on the VM. Hardware implementations typically adhere to the standard of intrinsic compatibility because they must be able to run all software correctly without assuming any software-specific behavior. Consequently, co-designed VMs such as the Transmeta processors adhere to intrinsic compatibility. Most system VMs such as the ones considered by Popek and Goldberg [2] also adhere to intrinsic compatibility; in their work, Popek and Goldberg essentially assume intrinsic compatibility, and then study the efficiency with which it can be achieved.

Intrinsic compatibility is also a good goal for process VMs, but, as we will show, there are a number of conditions which make intrinsic compatibility difficult to achieve while providing good efficiency. As we shall see, implementing intrinsically compatible process VMs is particularly difficult both because of the ways conventional ABIs are defined and the limitations of conventional ISA definitions.

3.1 Designs principles for intrinsically compatible process VMs

The basic VM structure that we consider was given in Section 2. To that basic structure, we now add two specific features, or design principles, that the emulation process satisfies as part of its compatible implementation. These two features are not sufficient by themselves, but are the ones that we focus on in the remainder of the report. As before, these features are commonly included in process VM implementations, so they do not significantly restrict the applicability of our results.

Process execution consists of the execution of a dynamic instruction sequence of finite length which causes a sequence of state transformations from an initial process state to a final state. This instruction sequence includes both user and system level instructions; it may include both normal user code and user-provided trap handling code. Let the dynamic instruction sequence be denoted by P and the process state be denoted by S. For simplicity, and without loss of generality, we consider that the only components

of S are the register memory state R and process mapped memory space A. During process execution a series of user instructions are executed, which modify the process state until some instruction, causes a trap from user mode to system mode. This terminates the user instruction sequence and initiates the processing of system code, normally part of an operating system. System code might perform privileged modifications to the process state and eventually transfer control back to user mode where user-level process instructions resume execution in either the normal guest code or a user-defined trap handler. We consider the user trap handler as part of the next user instruction sequence. The sequence of switches between user and system mode repeats until process completion or termination. Figure 3 illustrates these sequences; shown in the figure are an OS call, a trap with a user-level handler, and a trap that causes process termination.

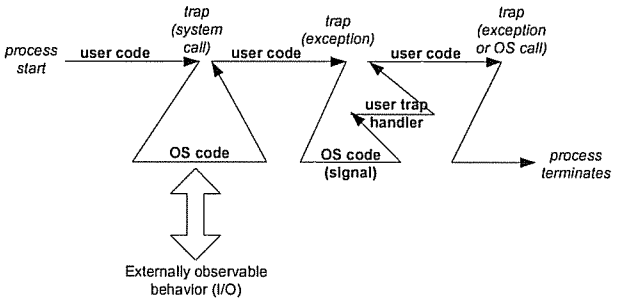

Figure 3. Process execution as a series of instruction sequences separated by traps

The complete process dynamic instruction sequence P can be represented as an ordered set of user instruction sequences (I1, I2,...In) alternating with system instruction sequences (T1, T2,...Tn), where the ith trap will initiate system code sequence Ti. Each I and T represents a sequence of user or system instructions and not just single instructions. Note that if two traps occur simultaneously they can be combined in the same trap sequence Ti for our model. Each instruction sequence takes a particular process state as its input, which it then transforms to the output process state.

$P = \{I1, T1, I2, T2...In, Tn\}$

or

$P = \{I_{S1\text{-}S2}, T_{S2\text{-}S3}, I_{S3\text{-}S4}, T_{S4\text{-}S5}, I_{S5\text{-}S6} \ldots\ldots\ldots I_{S\,(n\text{-}2)\,-\,S\,(n\text{-}1)}, T_{S(n\text{-}1)\text{-}Sn}\}$

Where Ii = $I_{Sj\text{-}Sk}$ is the user instruction sequence that transforms the process state from Sj to Sk, and Ti = $T_{Sj\text{-}Sk}$ is the system instruction sequence that transforms the process state from Sj to Sk with i =1,2,...n representing the order of the instruction sequence and j and k representing process states between the initial and the final state (including the two).

The process VM implements a mapping between the guest's native states and host states. The guest's native states are the states that exist when the guest software is executed on its native platform. The host states are the process VM states when it is executing the guest software. Let the set of all guest native process states be C_{native} and the set of all host states be C_{host}. Then the virtual machine implements a mapping f: $C_{native} \rightarrow C_{host}$.

Given initial states that adhere to some mapping function, two instruction sequences are *equivalent* if their final states also satisfy the same mapping, i.e. they perform transformations that preserve the mapping from the guest state to the host state. An instruction sequence I′ (T′) on the host platform is equivalent to the corresponding instruction sequence I (T) on the native platform if

$I'_{S'j\text{-}S'k} \Leftrightarrow I_{Sj\text{-}Sk}$ if S′k = f (Sk) given that S′j = f (Sj)

Similarly, $T'_{S'j\text{-}S'k} \Leftrightarrow T_{Sj\text{-}Sk}$ if S′k = f (Sk) given that S′j = f (Sj) where f is the mapping function from the guest state to the host states.

A process VM is intrinsically compatible if its mapping function f from the VM host process state to a guest's native process state is a one to one homomorphism with respect to all the instruction sequences in the dynamic process sequence of any guest program. The process VM map f is a one to one homomorphism with respect to all the instruction sequences in the dynamic instruction sequence set P. For any state $S \in C_{native}$ and instruction sequence I, the following relationship holds: f (I (Sj)) = I′ (f (Sj)).

An intrinsically compatible process VM needs to execute equivalent instruction sequences, at its correct ordering point, for each guest process instruction sequence in P. However, note the mapping implied by the homomorphism does not have to be maintained on an instruction-by-instruction basis. The mapping must only be present at the beginning and ending instructions of the instruction sequences in P. Suppose the native guest process execution is started in an initial state and the VM process execution

is started in a homomorphic initial state. Compatibility differences between the two process executions can only be perceived if their observable states are different. The only way process state becomes observable is through a trap to the OS for I/O or some other OS-implemented function. The process state is also observable when the process terminates. Hence, we focus only on those points where traps to the OS occur. These are the points shown in figure 3. Note that we only include ABI-defined traps. Page faults (as distinct from memory permission traps) or any other traps and interrupts that the OS handles without affecting or terminating user process execution are generally not ABI-defined and are transparent to all user processes.

We assume it is always possible that user instructions can be emulated compatibly (disregarding performance). This implies that for any possible user instruction sequence $I_{sj\text{-}sk}$ on the native platform there exists an instruction sequence $I'_{S'j\text{-}S'k} \Leftrightarrow I_{Sj\text{-}Sk}$. This assumption is valid if both the host and the guest are Universal Turing machines [23].

Emulating the system code (T sequences) consists of two parts – identifying the correct sequence of T at its correct place in the ordered set of instruction sequences and constructing an equivalent T sequence and inserting it at the correct place (identified above). However, constructing the equivalent T sequences might involve not only the process VM but also host OS for privileged operations. Depending on the properties of the guest and the host OS, compatible emulation of all trap sequences might not be possible even for exactly same initial process state. For example, a process VM may not be able to support all the priority levels in a guest application if the thread priority levels that a host OS supports are at a coarser granularity than the priority levels supported by the guest OS.

To focus on ISA properties that affect the construction of equivalent trap sequences we assume that there exists an equivalent host OS operation for every guest operation. This assumption is valid for process VMs where the host and the guest OS are either identical (dynamic binary optimizers) or similar (Linux family) and only the ISA is different. Examples include like Dynamo, DynamoRIO, Rosetta, Mojo and IA-32EL. This implies that given the same initial state the host OS operation will produce the same output state and externally observable behavior as the guest OS operation. We note that being able to compatibly emulate every guest OS operation is also a necessary condition

for building an intrinsically compatible process VM but we do not consider it here as this is beyond the scope of this report and is a research topic on its own. Derek Bruening's thesis [17] on DynamoRIO illustrates some of the issues involved.

Given a system code sequence, $T_{Sj\text{-}Sk}$ there exists $T'_{S'j\text{-}S'k}$ if $S'j = f(Sj)$ where $S'k = f(Sk)$.

Then, in terms of the ABI-defined trap points, we define two properties that must hold for the class of VMs that we consider and refer to as intrinsically compatible process VMs. Hereafter, a reference to intrinsically compatible process VMs shall refer to this particular class of VMs unless stated otherwise. Given our assumptions about the emulation of user instruction sequences and OS operations these two principles are the major components of building the class of intrinsically compatible process VM in which we are interested.

Trap correspondence -- All the trap and return points in native execution correspond to explicitly identifiable points in the VM's emulation of the guest process.

State correspondence -- At each identified trap point, the homomorphism between VM state and guest state holds. That is, the mapping between the VM state and the guest's native state can be determined. For each Ti' where $Ti' \Leftrightarrow T_{Sj\text{-}Sk}$, $S' j = f(Sj)$.

We briefly discuss the two properties and their implications.

Trap correspondence

This property is important because a process VM must emulate the effects of all traps that result in externally observable behavior, may change process state, or which terminate the process (which can be viewed as externally observable behavior). The identifiable point in the VM execution does not necessarily be at the point of a trap in the VM. It can also be a test followed by a jump to a runtime code sequence which emulates the guest OS behavior.

Note that here we consider traps that "change the process state" yet the trap correspondence principle refers to all traps. This expansion is done intentionally and for a practical reason. As the authors in [18] point out, it is extremely hard for a process VM to ascertain the effects of dynamic instructions in a generic system trap handler before the trap actually occurs. Consider the situation where a trap handler contains only a load

instruction whose address cannot be determined statically. Guaranteeing that the load will never cause a permission trap when there can be calls to change memory region permissions in the program is difficult to do prior to actually going to the trap handler. Also, the trap analysis would have to be done at least once even if it can be guaranteed to have no effect at any time in the future. Our assumption of VM behavior is consistent with the way trap analysis is done in all current process VM implementations such as Dynamo, Mojo, IA-32 EL, Rosetta etc.

Hence, we impose the property for all traps. We note that, if a certain VM does this analysis and can guarantee that a particular trap handler would not change process state under all execution conditions then it need not maintain trap correspondence for that particular trap or effectively the equivalent trap handler sequence can be NULL. This also extends to the case if the trap handler effects can be determined statically. Then the VM might construct the equivalent trap sequence as part of the user instruction sequence.

State correspondence

This property implies that at all trap points, the guest's precise native state can be completely re-constructed (via the homomorphism). This is important for correct OS or trap handler emulation. As with the trap correspondence principle, there may be cases where this is more than is required. In some cases, it might be possible to identify a proper subset of guest process state that a system call handler in the host OS is going to access or inspect. However, state correspondence does not require that the guest's native state actually be completely re-constructed, only that it *can be*. Furthermore, in general it will be extremely difficult to have fore-knowledge of the portions of the state that the OS or trap handler may access or inspect without extensive program analysis, so a conservative assumption is called for and is implemented in all current process VMs. This also helps shield the process VM implementation from being affected by changes to OS internals. In fact, most process VMs actually do construct the complete precise state at trap points.

4 Supporting the trap correspondence property

When implementing an intrinsically compatible process VM it is the trap correspondence property that tends to lead to the more constraining implementation requirements. Consequently, we focus on trap correspondence and assert necessary conditions for satisfying trap correspondence.

4.1 Traps

We divide instructions into two types – non-trappable and trappable. Non-trappable instructions are those that never cause an exception. Examples of such instructions may be logical or shift instructions. Trappable instructions are those that may either unconditionally transfer control to the OS (i.e., a system call instruction) or may do so conditionally in the event of exceptions.

Instructions that conditionally trap do so depending on their operands. That is, certain operand ranges will result in a trap, others will not. Arithmetic instructions overflow for certain operand values, for example. Memory access instructions (loads/stores) may cause permission traps for certain address values if the type of access is not permitted by the permissions in the page table; for example, a store to a read only region or access to a region that is not mapped to the application (an "out of bounds" access).

Static analysis of a given application binary may be able to determine that the operand values for a particular trappable instruction will never result in a trap for any dynamic execution of the binary. This analysis can be performed by VM software prior to execution or as part of the runtime translation process. For example, if the address of a load is generated using immediate values and is invariant, then static analysis may determine that it will never cause a permission trap.

In many cases, however, statically guaranteeing that a given instruction is not going to trap (especially if traps can be enabled or disabled at any time during program execution) is either not practical or not possible. We define such a guest instruction to be a *hazardous* instruction. Note that many hazardous instructions will never actually cause a trap; being hazardous only means that an instruction cannot be statically guaranteed never to trap. Examples of hazardous instructions typically include guest loads, stores, and indirect jumps whose target addresses are calculated dynamically at run time.

A hazardous instruction is one that can cause the insertion of a T sequence in the set of process instruction sequences P. A C program with pointers and pointer arithmetic would typically contain many hazardous instructions. For example, in the following code snippet, the load address pointed to by variable y depends on the set of values that an arbitrarily complex function foo() can return. Interprocedural static analysis is complex and might even be impossible (for example, if user input is involved).

```
y = foo();  z = *(int*) y;
```

4.2 Dynamic Exception Testing

Given that the hazardous instructions cannot be statically guaranteed to be free of exceptions, some form of dynamic checking must be implemented. In the process VMs that we consider, there are three methods for doing dynamic checking, two of them we define to be explicit methods and one is implicit.

Explicit methods: The VM implementation can explicitly check for exceptions either during interpretation or through explicit checks inserted in binary translated code. If an exception is detected it results in a jump to runtime software. It is these tests that mark the trap points given in the trap correspondence condition. A process VM that uses interpretation can do exception testing easily because it interprets each dynamic instruction. However, interpretation is very inefficient from a performance perspective. If the process VM uses binary translation to enhance performance, then the explicit dynamic checks would also add significant overhead.

Implicit method: The VM can check for exceptions implicitly by letting translated host instructions detect them as part of their normal execution. For example, the host's memory permission hardware can be relied upon to detect permission violations in guest code if the runtime performs the necessary OS calls to set the host memory permissions properly. An exception detected in this implicit manner results in a trap to the OS and then to a signal handler registered by the runtime. The runtime signal handler then hands control to a runtime exception emulator responsible for constructing the precise guest state and delivering the proper guest ABI signal. If the guest application has registered any signal handlers then the runtime schedules the proper guest signal handler for execution. Otherwise it will likely terminate the process (depending on the ABI specification).

4.3 Necessary Condition for Intrinsic Compatibility

Assertion 1: *In an intrinsically compatible process VM implementation, all hazardous instructions must be dynamically checked for exceptions during emulation.*

Argument: This assertion follows immediately from trap correspondence. If all trap points must be identifiable, and hazardous instructions cannot be statically guaranteed to be free of traps, then they must be dynamically checked in the class of VMs that we are considering.

Suppose the hazardous instruction is not dynamically checked and it causes a trap j in the guest's native execution. This would insert a trap sequence Tj in the guest's native execution sequence. But, during process VM emulation the trap would not be detected and by definition no equivalent trap sequence would be inserted.

The test may be either explicit or implicit, but it must be done. The test does not necessarily mean that the operation specified in the instruction must be performed – e.g. a load/store optimization may not actually perform the load/store, but the exception check must be performed.

5 Efficient Process VMs

Intuitively, a process VM is efficient if 1) binary translation is used, and 2) only host instructions necessary for carrying out intended guest instructions' operations are executed. We can identify two potential causes of inefficiency in process VMs – explicit analysis for trap detection and spurious traps during implicit detection. Both cause the VM to execute extra instructions that do not correspond to the intended (exception-free) execution of guest code.

We define an *efficient process VM* to be a process VM that uses binary translation and:

1) There is no explicit analysis for exception conditions.

2) There are no spurious memory permission traps when implicit exception detection is used.

We will elaborate on both of these in following subsections and also on how our assumptions are consistent with current process VM implementations.

Note (again) that we take a similar approach to Popek and Goldberg, this time regarding efficiency. That is, we do not quantify efficiency, but rather assume that certain emulation methods are inherently inefficient. Then, a VM emulation that does not use these methods is efficient. As pointed out earlier in the related work section (1.2), Popek and Goldberg define efficiency to be a characteristic of the VMM and state that a condition for efficiency is that "All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the control program."

5.1 Condition 1: No explicit analysis

The first condition for an efficient process VM implementation is that it does not accrue the overhead of explicitly analyzing hazardous instructions to detect exceptions. Current process VMs like DynamoRIO, Dynamo, FX!32, Rosetta etc are examples of process VMs that do not analyze hazardous instructions explicitly to avoid the inefficiency. Authors in [17] point out that explicitly analyzing memory loads would reverse the performance boost from DynamoRIO. A key aspect of the approach that current process VMs employ is to make use of the host platform memory permission subsystem to signal a trap if an emulation load/store accesses any region of memory other than the guest region. A process VM implementation needs to satisfy an extra set of conditions to maintain intrinsic compatibility when this is done.

In general, static analysis may only be able to exclude a subset of operand values (addresses) that can lead to an exception. Consider the range of excepting addresses that cannot be determined by static analysis. Then this range of addresses cannot include any of the readable/writeable runtime memory region. In general, however, the address that a hazardous load/store can access is not known statically and the entire host virtual address space must be considered.

Assertion 2: *If a guest binary can contain hazardous load/store instructions, then an intrinsically compatibility efficient process VM must not have any part of the runtime region readable/writeable in emulation mode.*

Argument: To be efficient, implicit trap checking is employed, by definition. However, if a hazardous load or store should access the readable/writeable runtime region then implicit checking will not catch this occurrence, i.e., a "rogue" load or store can read from or write to the runtime region without throwing an exception. On the other hand,

the guest's native platform would throw the exception. This violates trap correspondence.

This assertion places significant limitations on the VM implementation and on the relationships between the guest's native ISA and the host ISA. Following corollaries indicate these limitations.

Corollary 1: *In an efficient, intrinsically compatible process VM, during emulation mode there can be no loads/stores to any data area in the runtime region (as defined in Section 2).*

Argument: The runtime region is not accessible during emulation mode according to Assertion 2. Hence no loads/stores (whether intentional or not) can access the runtime region in emulation mode.

A process VM implementation may attempt to deal with the rogue load/store problem by providing different memory permissions when in runtime mode and in emulation mode. For example, a VM implementation may provide the two sets of permissions as shown in Figure 4. The runtime can implement such a dual-permission scheme by using host page permission mechanisms implemented with OS calls to apply different permissions to the runtime region during the runtime and emulation mode [1].

However, certain VM implementations (not providing intrinsic compatibility), rely on runtime data region(s) that are read/writeable when in emulation mode. In particular, any register spill area must be read/writeable, and an indirect jump table must at least be readable.

Unfortunately, corollary 1 excludes the use of any type of scratch space, and this leads to the next corollary.

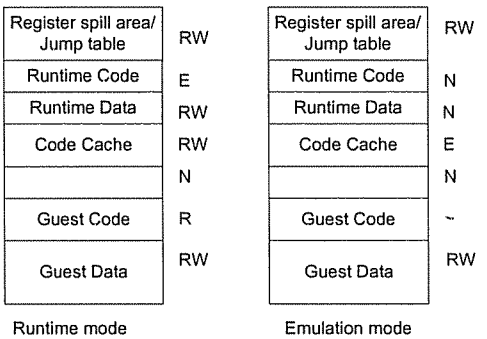

Figure 4. Permissions for different regions during runtime and emulation mode. R = read, W = write, E = execute, N = non-accessible, ~ = same as the native guest process.

Corollary 2: *In an efficient, intrinsically compatible process VM, there must be at least as many user level registers in the host ISA as in the guest ISA.*

Argument: The homomorphism that must be satisfied at trap points implies that the guest state during emulation must be mapped to host state. Hence, all the guest registers must be mapped to the host registers (unless they can be guaranteed to be dead even inside trap handlers) as there can be no scratch memory space for spilling guest application registers to the runtime region. This implies that there must be at least as many architected registers in the host platform as in guest platform given our earlier assumption that the registers are not compressed.

As the authors in [18] point out, a dead register from a compiler's perspective might not be dead from a trap handler's perspective. Unless deep dynamic trap handler analysis is done, the process VM implementation has to assume that all registers may be live inside a trap handler.

We observe that Corollary 1 also excludes the implementation of indirect jump tables. In particular, a hazardous, rogue load that reads from a jump table will not throw an address-out-of-range exception as would be the case if the instruction were executed on its native platform. However, at least for binary optimizers, the inability to implement a software jump table is not a fundamental problem. One can employ the technique used by ADORE [8] to patch the guest code instead of using a jump table.

Corollary 3: *In an efficient, intrinsically compatible process VM, the host ISA must support "execute only" permission.*

Argument: The runtime region must not be readable/writeable in emulation mode. But the code cache region must be executable in emulation mode. This implies that an execute-only permission is needed for the code cache.

Note that if the host ISA combines execute permissions with read/write permissions, as is sometimes done, then corollary 3 is violated.

5.2 Condition 2: No Spurious Traps

One could implement a method of testing for address exceptions that uses implicit checking, but causes a high number of traps to the runtime, which then filters out the "real" ones. In a sense this is a "loophole" to condition 1, which is closed by condition 2. We recognize that process VM implementations might benefit from the use of some speculative techniques that provide higher performance at the cost of very few spurious traps (if the speculation is highly accurate). However, that is a tradeoff which is dependent on the workload and therefore efficiency can not be guaranteed by the implementation alone. Hence, we stipulate that there are no spurious traps for efficiency.

Also, implicit checking relies on direct use of the host memory address translation and permission subsystems, and this can lead to spurious traps (and inefficiency) unless certain conditions are met. For example, if all the guest virtual addresses are not mapped to host virtual addresses, then there may be extra traps to the runtime when an unmapped guest address is referenced. This situation can occur in practice because the runtime and the guest software must share the same virtual address space. In this situation, the runtime may use techniques similar to traditional virtual memory to manage the mapping between the guest VA space and the host VA space. If the host VA space is much smaller than the combined size of the runtime software and the guest VA space then there would be a large number of extra traps to the runtime; analogous to thrashing in conventional virtual memory.

This leads us to our next assertion.

Assertion 3: *In an efficient intrinsically compatible VM the size of the host virtual address space must at least be equal to the size of guest virtual address space plus the size of runtime software.*

Argument: For intrinsic compatibility the VM must be able to accommodate a guest with maximum process size, which is equal to the size of the guest virtual address space. Note that we assume process VM implementations that do not compress the contents of guest memory. If the host virtual address space is not large enough to map the entire guest virtual address space and the runtime software simultaneously, then the runtime will only be able to map a proper subset of the guest process to host VA space (leaving

room to map itself in the remaining space). This will lead to spurious traps that would do not occur when the guest runs on its native platform.

If the size of the host virtual address space is at least equal to the size of the guest virtual address space plus the size of the runtime software then the entire guest virtual address space and the runtime can be mapped simultaneously. It is then possible to map any guest application and eliminate virtual page faults (which appear as permission faults) associated with a guest virtual page not being mapped. For example, a position independent runtime can map guest addresses idempotently to the host address.

Assertion 4: *In an efficient intrinsically compatible VM the guest permission types must be a subset of host permission types.*

Argument: If there is a mismatch in the types of permissions available in the guest and the host platforms then the runtime software would need to emulate the guest permission in software. For example, if the guest supports separate 'Read' and 'Execute' permissions and the host only supports 'Execute' and 'Read and Execute' permissions, the runtime will have to mark a read-only guest page as non-accessible in the host process and keep track of all such pages. When an emulation load tries to access such a page, it will trap to a runtime trap handler that can determine if the load was to a read-only page and, if so, emulate the read. If the guest permission types are a subset of host permission types then it is possible to map the guest and host permissions idempotently.

Assertion 5: *In an efficient intrinsically compatible VM the memory permission granularity of guest ISA must be a multiple of that of the host ISA.*

Argument: If the memory permission granularity is equal to the page size (as it typically is) on both the guest and host, then this assertion places a restriction on the guest and the host page sizes. If the host page size is bigger than the guest page size then two guest pages with different permissions can be mapped to the same host page. For intrinsic compatibility, the runtime software would need to mark the host page with the more restrictive of the two permissions. This would cause extra traps to the runtime software when the emulated code accesses valid addresses on the other guest page. If the permission granularity of the guest ISA is a multiple of that of the host ISA then it is possible to map multiple host pages with the same permission.

5.3 Implications for VM Implementations

The above assertions, when applied to current process VM implementations severely restrict the transformations that may be performed when performing binary translation. In particular, we will show that safe optimizations are very limited in the next subsection. Then we will discuss the implications for compatible dynamic binary optimizers. Finally, we discuss combinations of guests and hosts that do satisfy the assertions, indicating that their implementations may be efficient.

Unsafe Optimizations

A number of code optimizations are unsafe and cannot be used in intrinsically compatible process VMs that do not explicitly test for exceptions. One of the most important restrictions is the necessity for maintaining trap correspondence. This restricts optimizations that remove hazardous instructions, such as dead code elimination. Removing an otherwise dead instruction that might trap can lead to non-equivalent results of execution on the host [18]. For example, the load instruction in the following code is dead in this basic block. However, if the load traps at run time and the trap handler modifies the value of register r2 then the instruction is not actually dead. If the runtime removes the load instruction and does not check for the trap during execution, then the value of r3 after the add instruction might differ from native execution.

```
lwz r3, 0(r2);
add r3, r2, r5;
```

Trap correspondence also constrains optimizations like code hoisting above loops because even if the instruction is not removed statically it does not execute all of its dynamic instances, any one of which might trap.

State correspondence at trap points restricts the reordering of load/store instructions or motion of code around the stores. This especially constrains stores because effects of any stores present in the instruction sequence following the trap point should not be visible to the trap handler. Note that compiler optimizers can perform these same optimizations safely because they generate a new binary after compilation and do not have to maintain compatibility with an existing binary.

The optimization safety issues just described are well-known [1] and we have repeated them here for completeness. However, the following issue is less widely known. Any transformations that result in register spills are potentially unsafe for dynamic binary optimizers because there can be no register spill area. We mention a few optimizations used in current optimizers that require spills in order to employ additional register values. The ADORE system requires extra register values for data cache prefetch transformations. Dynamo and DynamoRIO use extra register values and spill to scratch space for indirect branch adjustment, redundant load elimination, code motion, and the mode switches from emulation mode to runtime mode. In general, binary optimizers cannot guarantee intrinsic compatibility on ISAs that necessitate the use of a register to change permissions or hold the address of the jump location to the runtime code.

Nevertheless, a process VM can still perform highly localized optimizations in a region between two potentially trapping instructions and use any dead registers in that restricted region. For example, in the following code snippet, the (non-trappable) and instruction between the two load instructions is dead and can be removed and r2 can be used as a free register provided it can be restored in case the second load traps.

```
lwz r2, 0(r7);
and r4, r2, r1;
xor r4, r4;
....
lwz r2, 0(r6);
```

Dynamic Optimizers

The most important implications of the assertions we have made are for dynamic binary translators where there is no extra virtual or register address space. As explained above, this restricts code optimizations in a major way.

The virtual address space restriction is not as big a problem as the register space problem, however, because most applications do not use the entire virtual address space, leaving room for the runtime. Any applications, that do not leave room for the runtime, can simply "bail out" [1].

Dynamic optimizers satisfy some of the conditions for efficiently supporting intrinsically compatibility automatically. For example, the inefficiencies due to different page sizes and permission types do not arise. However, because the register file size is the same in the guest and host, there is no extra register space for the optimizer or runtime to use. Because the registers cannot be spilled to memory unless (inefficient) explicit analysis of all memory instructions is performed, a practical binary optimizer may only be able to maintain a level of extrinsic compatibility. For example, it may provide compatibility only for programs know to have no out-of-range load/stores, and for many applications, this will not be an obstacle.

A key point here is that research to date on dynamic binary optimizers is for experimental, prototype systems that may be incomplete. Consequently, it is unclear whether their lack of intrinsic compatibility is simply because they are incomplete, or for more fundamental reasons. Based on our work, and for the general implementation frameworks that are used, we claim that it is for fundamental reasons. For dynamic optimizers that follow the implementation structure that we have described; in order to achieve intrinsic compatibility, the allowed code optimizations are so severely restricted, that it is probably not worthwhile to implement dynamic optimization in an intrinsically compatible way. So we claim that, practically speaking, intrinsically compatible dynamic binary optimizers cannot be constructed for conventional ISAs without special support to remove the inefficiencies that we have cited.

Conforming Process VMs

From the discussion thus far, it might seem that the conditions we have laid out are so restrictive that most intrinsically compatible process virtual machines will be inefficient. However, there are some important practical cases where the conditions asserted above may be satisfied. The most important cases, perhaps, are virtualizations of the 32-bit IA-32/Windows and IA-32/Linux ABIs on a typical RISC platform or the Intel IPF (Itanium) platform. The IA-32 ISA has only eight registers and most RISC implementations have more than 32 registers. Also, the IA-32 address space is limited to 32 bits whereas the RISC address space is usually 64 bits. RISC page sizes are also

typically larger than the 4KB IA-32 page size. Finally, most RISC ISAs support an Execute-only permission mode.

High level language virtual machines such as a Java VM [13] or the Microsoft CLR [20] are similar to process VMs, but they do not face similar problems with compatibility because their architecture is not as rigid as a conventional ISA. For example, Java and the CLR do not specify a fixed size for the VA space, nor do they have registers. Even if a program runs out of memory at different points on different machines, it is still compatible with the ABI specification. Java has a stack based ISA with no architected registers to deal with for compatibility.

We note however, that these implications do not apply to process VMs that do not satisfy our assumptions. In other words, process VM implementers might consider explicit hardware or OS support to circumvent the implementation constraints that are assumed and practiced by current process VMs. For example, reserved load and store instructions that can only be used by the process VM to read or write the runtime region can be used to implement register spilling. Also, a process VM aware OS might be used to mark special pages for accessible to the runtime software during emulation mode with a trap being generated when there is an access by the guest code. However, this restricts the implementation of process VMs by third party vendors with no means of ensuring hardware or OS support.

6 Extrinsic Compatibility

An extrinsically compatible process VM that does not handle all the corner cases may be much more efficient and easy to design and implement than an intrinsically compatible one. However, it is extremely important to characterize exactly the type of application behavior that can lead to incompatibility.

The FX!32 system was designed to be IA-32/Windows compatible only for a specified list of Windows applications. The Rosetta [16] system guarantees correctness only for PowerPC programs that do not use the Altivec extensions and has some other restrictions on the types of applications that it can handle. Although it is not explicitly stated, both Dynamo and DynamoRIO are unlikely to be intrinsically compatible for

applications that have rogue reads and writes to the register spill area, or reads from the memory region holding the indirect jump table.

Omniware [15] requires recompilation of the guest application to restrict it from using all the registers and reserves some registers for efficient dynamic permission checking.

7 Conclusions

We conclude that although intrinsic compatibility is the standard for most VM implementations, it is more difficult to achieve in process VMs where the guest and host platforms implement conventional ISAs. A problem with conventional ISAs is that most of the "dimensions" are defined to have a fixed size; for example, there is a fixed-size register file, address space, and permission granularity. Then, for intrinsic compatibility, a process VM must fit ("squeeze" may be a better term) both the guest software and runtime software into the same fixed dimensions as a host platform's process would normally go. A primary difficulty with this "squeeze" is that any hazardous guest loads or stores cannot be allowed to reach outside the bounds of the guest process.

A dynamic binary optimizer, by definition, has exactly the same dimensions in the guest and host. There is no space (either register or memory) left over for optimizing; e.g. registers may not be spilled to make room for other register values. An additional problem is that a software indirect jump table cannot be used if intrinsic compatibility is to be maintained in the presence of hazardous loads. This leads us to conclude that an intrinsically compatible dynamic binary optimizer (which actually optimizes) is probably not practical.

On the other hand, when virtualizing a guest platform on an architecturally larger host (as when an IA-32 based ABI is the guest and a RISC or IPF platform is the host) then these obstacles are not present.

We should also emphasize that just because an intrinsically compatible VM is not efficient according to our definition, it may still perform well enough to be useful, especially if the goal is functional compatibility and not performance. For example, explicit range checks for all hazardous loads and stores may slow down guest performance by a factor of two (more or less), but this may be quite acceptable in many

situations. Also, there may be implementations that do not fit our defined model and hence are exempt from the implications presented in this report. We do note however, that current high performance VMs fit the model that we define.

We also suggest future research in adding "hooks" or architected features to a conventional ISA to enable it to more efficiently run process VMs. This is similar in concept to the new features added by Intel and AMD to implement system virtual machines efficiently. These features would likely include extensions to the memory permission architecture so that memory areas for register spills and jump tables can be implemented, while still being protected from ordinary, hazardous loads and stores.

8 References

[1] Vasanth, B., Duesterwald E., and Sanjeev, B.. Transparent Dynamic Optimization: The Design and Implementation of Dynamo. HPL-1999-1978.

[2] Popek, G. J., and Goldberg, R.. Formal requirements for virtualizable third generation architectures. Communications of the ACM, July 1974, pp. 412-421.

[3] Smith, J. E., and Nair, R.. Virtual Machines: Versatile Platforms for Systems and Processes. Morgan Kaufmann, 1st edition, 2005.

[4] Hookway, R. J., and Herdeg, M. A.. Digital FX!32: Combining Emulation and Binary Translation. Digital Technical Journal, Jan 1997.

[5] Seawright, L. H, and MacKinnon, R. A.. VM/370 – A Study of Multiplicity and Usefulness. IBM Systems Journal, 1979.

[6] Ebcioglu, K., and Altman E.. DAISY: Dynamic Compilation for 100% Architectural Compatibility. International Symposium on Computer Architecture, June 1997, pp. 26-37.

[7] Klaiber, A.. The Technology Behind Crusoe Processors. Transmeta Technical Brief, 2000.

[8] Chen H., Lu J., Hsu W.C., and Yew P.C.. Continuous Adaptive Object-Code Re-optimization Framework. ACSAC, 2004.

[9] Baraz L., Devor T., Etzion O., Goldenberg S., Skaletsky A., Yigal, W. Y.. IA-32 Execution Layer: a two-phase dynamic translator designed to support IA-32 applications on Itanium-based systems. International Symposium on Microarchitecture, 2003.

[10] http://www.winehq.com

[11] Chen W.K., Lerner S., Chaiken R., and Gillies D.. Mojo: A dynamic optimization system. 4th ACM Workshop on Feedback-Directed and Dynamic Optimization, 2000, pp. 81—90.

[12] http://www.vmware.com/products/desktop/ws_features.html

[13] Meyer, J., and Downing, T.. Java Virtual Machine. O'Reilly, 1997.

[14] Dragovic B., Fraser K., Hand S., Harris T., Ho A., Pratt I., Warfield A., Barham P., and Neugebauer R.. Xen and the Art of Virtualization. Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.

[15] Lucco S., Sharp O., and Wahbe R.. Omniware: A universal substrate for web programming. www.w3.org/Conferences/WWW4/Papers/165/

[16] http://www.transitive.com/technology.htm

[17] Bruening D.L.. Efficient, transparent and comprehensive runtime code manipulation. Ph. D. thesis, ECE dept, MIT, Sept 2004.

[18] Gschwind M., and Altman E.. Precise exceptions in dynamic compilation. Proceedings of the 11th International Conference on Compiler Construction, 2002.

[19] Kiriansky V., Bruening D., and Amarasinghe S.. Execution model enforcement via program shepherding. Proceedings of the 11th USENIX Security Symposium, 2002.

[20] Thai T., and Lam H.. NET Framework Essentials. O'Reilly, 2001.

[21] Gschwind, M., Ebcioglu, K., Altman, E., Sathaye, S.. Binary translation and architecture convergence issues for IBM System/390. In Proc. of the International Conference on Supercomputing, May 2000.

[22] Bala, V., Duesterwald, E., and Banerjia, S.. Dynamo: A transparent Dynamic Optimization System. SIGPLAN PLDI, June 2000, pp. 1–12.

[23] http://www.turing.org.uk/turing/scrapbook/machine.html