Jones Optimality and Hardware Virtualization

A Report on Work in Progress

Boris Feigin Alan Mycroft
Computer Laboratory, University of Cambridge
{Boris.Feigin, Alan.Mycroft}@cl.cam.ac.uk

Abstract

The growing popularity of hardware virtualization (VMware and Xen being two prominent implementations) leads us to examine the common ground between this yet-again vibrant technology and partial evaluation. A virtual machine executes on *host* hardware and presents to its *guest* program a replica of that host environment, complete with CPU, memory, and I/O devices. A virtual machine can be seen as a self-interpreter.

A program *specializer* is considered Jones-optimal if it is capable of removing a layer of interpretational overhead. We propose a formulation of Jones optimality which coincides with a well-known virtualization efficiency criterion.

A fully abstract programming language translation (an idea put forward by Abadi) is one that preserves program equivalences. We may translate a program by specializing a self-interpreter with respect to it. We argue that full abstraction for such translations captures the notion of *transparency* (whether or not a program can determine if it is running on a virtual machine) in virtual machine folklore.

We hope that this discussion will encourage wider exchange of ideas between the virtualization and partial evaluation communities.

Categories and Subject Descriptors D.3.1 [*Programming Languages*]: Formal Definitions and Theory—Semantics; D.3.4 [*Programming Languages*]: Processors—Interpreters; D.4.8 [*Operating Systems*]: Performance—Modeling and prediction

General Terms Performance, Languages

Keywords Jones optimality, virtualization, virtual machines, full abstraction

1. Introduction

Over the last few years, many academic and commercial hardware virtualization offerings have emerged (VMware [1] and Xen [5] are two prominent examples). A virtual machine monitor (VMM) is responsible for sharing out the hardware resources of the *host* system between several simultaneously running virtual machines (VMs). Each VM presents to its *guest* program a replica (with possible variations in the number and types of available I/O devices, amount of memory, *etc.*) of the host environment. In their seminal 1974 paper, Popek and Goldberg [20] described three requirements that a VMM must meet: *efficiency*, *equivalence*, and *resource control*. A VMM must not impose undue overhead, be faithful to the original hardware, and retain control over certain aspects of guest program execution (access to I/O devices, for example).

Since we are not concerned with issues of concurrency, in the remainder of this article we shall assume that only a single VM is running and use "VM" in preference to "VMM" henceforth.

1.1 Virtualization as self-interpretation

Let p be a program in language L (machine code) and d and s range over the inputs and outputs of p respectively. Let $[\![\cdot]\!]$ be a valuation function on L as implemented by the hardware. Recall that for a self-interpreter $sint$ and a program specializer mix, the following equalities hold by definition:

$$\begin{aligned} s &= [\![p]\!](d) && (1)\\ &= [\![sint]\!](p, d) && (2)\\ &= [\![[\![mix]\!](sint, p)]\!](d) && (3)\end{aligned}$$

Conceptually, a VM plays the role of $sint$. A common technique of avoiding the associated interpretational overhead relies on hardware support (or "assists") and is known as "trap-and-emulate". Instructions of the guest program are executed directly by the hardware; however, in keeping with the resource control requirement, privileged instructions (reading from a device, for example) are *trapped* and the VM is allowed to *emulate* them. Popek and Goldberg [20] derived formally the conditions that a trap-and-emulate hardware architecture must meet in order to support virtual machines.

Techniques based on binary translation are used when implementing VMs on hardware architectures that do not provide virtualization assists. One may argue that Equation 3 is the essence of this approach.

Equation 3 captures the spectrum from Equation 1 (*cf.* Equation 3 with a Jones-optimal mix; see next section) to Equation 2 (*cf.* Equation 3 with a trivial mix). Thus, even though mix has no immediate analogue in virtualization, we rely on Equation 3 pervasively.

1.2 Overview

We attempt to relate hardware and software techniques of efficient virtual machines to the notion of interpretational overhead known from partial evaluation.

To capture Popek and Goldberg's efficiency requirement for VMs, in Section 2 we propose a variant of Jones optimality based on program traces. Section 3 defines UAL, an assembly language for an idealized von Neumann machine with I/O devices. We show a meta-circular interpreter for UAL and argue that it fails to make

PEPM'08, January 7–8, 2008, San Francisco, California, USA.

an efficient VM. Virtualization assists after trap-and-emulate are presented in two alternative extensions to UAL (UAL/`step` and UAL/`eval`) in Section 4. The trivial specializer is trace Jones optimal with respect to a self-interpreter for UAL/`eval`.

Section 5 contains a discussion on the relevance of full abstraction to virtual machines. References to related work are given in Section 6. Section 7 concludes with an outlook to further work.

This paper captures a snapshot of work in progress. The interested reader is encouraged to consult the first author's website (`http://www.cl.cam.ac.uk/~bf241/virt/`) for future revisions.

2. Jones Optimality for Traces

A Jones-optimal specializer removes a layer of interpretational overhead. Let $sint$ range over self-interpreters, and p over programs, then a specializer mix is Jones-optimal ("strong enough" in the original nomenclature of Jones [11]) whenever

$$\exists sint\, \forall p \quad [\![mix]\!](sint, p) =_\alpha p \tag{4}$$

The definition above is usually endowed with a side-condition to the effect that mix may not "cheat" by textually comparing $sint$ against a known self-interpreter and emitting p on success. (To a cheating, Orwellian mix, "some self-interpreters are more equal than others".)

Let us generalise the definition in the style of Glück [10]. Let $\sim$ be a binary relation on programs capturing some equivalence or ordering. Then define $\sim$-Jones-optimality of mix with respect to $sint$ as follows

$$Jopt_\sim(mix, sint) \quad \triangleq \quad \forall p \quad [\![mix]\!](sint, p) \sim p \tag{5}$$

Then $\sim$-Jones optimality of mix on its own is defined as

$$Jopt_\sim(mix) \quad \triangleq \quad \exists sint \quad Jopt_\sim(mix, sint) \tag{6}$$

Let us now rewrite Equation 4 as

$$Jopt_{=_\alpha}(mix) \tag{7}$$

The original definition of Jones optimality disallows potential optimisations that mix might be able to perform, since, *e.g.* `1 + 2` $\neq_\alpha$ `3`. Consequently, Jones et al. [13, Definition 6.4] relaxed the ordering to $\leq_{time}$, defined as follows:

$$p \leq_{time} p' \quad \triangleq \quad \forall d \quad time_p(d) \leq time_{p'}(d)$$

where $time_p(d)$ is the execution time of running p on data d. Under a very reasonable assumption that α-equivalent programs consume equal execution time, this definition subsumes the original. The crucial change is that an intrinsic ("static") notion of equivalence of Equation 4 is replaced with an extrinsic ("dynamic") one. We now define variants of Jones optimality (based on program traces) intermediate between $Jopt_{=_\alpha}$ and $Jopt_{\leq_{time}}$.

2.1 Traces

Intuitively, a *program trace* is a sound abstraction of a program run. Usually, it is the sequence of states (registers, memory, *etc.*) that are encountered during the run, or the sequence of operations (and their inputs and outputs) that are executed. Since we are trying to characterise interpretational overhead (*i.e.* extraneous operations), the latter approach is better suited. Let $\mathcal{T}_d(p)$ denote the trace of program p on input d. Note, that values read from I/O devices are assumed to be part of the input d.

2.2 $Jopt_{=_\mathcal{T}}$ and $Jopt_{\leq_\mathcal{T}}$

Define trace equality of programs, $=_\mathcal{T}$, as follows:

$$p =_\mathcal{T} p' \quad \triangleq \quad \forall d \quad \mathcal{T}_d(p) = \mathcal{T}_d(p') \tag{8}$$

Two α-equivalent programs produce the same trace (since a trace only includes *runtime* values). Two programs with the same trace consume equal execution time, making $Jopt_{=_\mathcal{T}}$ an intermediate criterion between the two classical formulations of Jones optimality.

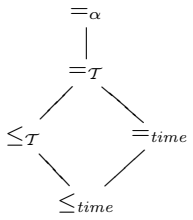
Let us also define a "subtrace" relation $\leq_\mathcal{T}$. Inspired by process calculi, first say that trace t *is simulated by* t', written $t \preceq t'$, iff t is a subsequence of t'. Formally

$$t \preceq t' \quad \text{iff} \quad t_i = t'_{f(i)} \tag{9}$$

for some strictly increasing function f on sequence indices. We define $\leq_\mathcal{T}$ by lifting this to programs:

$$p \leq_\mathcal{T} p' \quad \triangleq \quad \forall d \quad \mathcal{T}_d(p) \preceq \mathcal{T}_d(p') \tag{10}$$

To summarise, we have that



2.3 Self-interpretation

We expect that, *modulo* instructions with immediate operands (see Section 3.4), a self-interpreter implements instructions of the interpreted program using the corresponding machine instructions. Thus, we can assume that

$$\forall p \quad p \leq_\mathcal{T} [\![mix]\!](sint, p)$$

Intuitively, the self-interpreter always executes *at least* those operations that a program would on its own. With minor caveats, we view this as an issue of correctness rather than efficiency.

A supremely efficient self-interpreter must also execute *at most* those operations that a program would on its own. Thus, our Jones optimality criterion is now simply $Jopt_{\leq_\mathcal{T}}$.

2.4 VM efficiency

According to [20, p. 417], efficiency in VMs is achieved when:

> *"All innocuous instructions are executed by the hardware directly, with no intervention at all on the part of the [VMM] control program."*

No overhead should be incurred when executing non-privileged instructions such as register movements and ALU operations. Privileged instructions, on the other hand, are exempt from this requirement as their implementations may need to emulate devices which are not physically present in the machine.

Several VMs may be *nested* forming a tower of self-interpreters. We annotate every operation in the trace of a program with a natural number n indicating the *virtualization nesting level* at which the operation was issued: $n = 0$ corresponds to direct execution on hardware. The nesting level can be easily added to every state in an operational semantics.

Let $p' = [\![mix]\!](sint, p)$. We expect a privileged operation in the trace of $[\![p]\!](d)$ to correspond to a sequence of (possibly privileged) operations in the trace of $[\![p']\!](d)$. Define a projection ϕ that removes interpretational overhead incurred by the VM as a result of emulating privileged instructions on behalf of the program. Suppose the VM is executing at nesting level l, then let $\phi_l(\mathcal{T}_d(p'))$ be the subtrace of $\mathcal{T}_d(p')$ consisting of only those operations where $n > l$.

p	::=	$([b{:}]\ \iota)^*$		Program.
ι	::=	v		Value v (data).
	|	`nop`		No-op.
	|	`movi`	$v,\ r$	Copy value v to register r.
	|	`mov`	$r_1,\ r_2$	Copy from r_1 to r_2.
	|	`ld`	$(r_1),\ r_2$	Load from location in r_1 to r_2.
	|	`st`	$r_1,\ (r_2)$	Store from r_1 to location in r_2.
	|	`beq`	$b,\ r_1,\ r_2$	Conditional branch.
	|	`add`	$r_1,\ r_2,\ r_3$	$r_3 := r_1 + r_2$
	|	`sub`	$r_1,\ r_2,\ r_3$	$r_3 := max(0, r_1 - r_2)$
	|	`halt`		Halt the machine.
	|	`in`	$\langle r_1\rangle,\ r_2$	Input from port in r_1 to r_2.
	|	`out`	$r_1,\ \langle r_2\rangle$	Output from r_1 to port in r_2.

Figure 1. Syntax of UAL.

$\sigma(pc, \ldots, pc+3) =$		Then $(\rho, \sigma, pc) \rightsquigarrow$
`nop`		$(\rho, \sigma, pc+4)$
`movi`	$v,\ r$	$(\rho\,[r \mapsto v]\,, \sigma, pc+4)$
`mov`	$r_1,\ r_2$	$(\rho\,[r_2 \mapsto \rho(r_1)]\,, \sigma, pc+4)$
`ld`	$(r_1),\ r_2$	$(\rho\,[r_2 \mapsto \sigma(\rho(r_1))]\,, \sigma, pc+4)$
`st`	$r_1,\ (r_2)$	$(\rho, \sigma\,[\rho(r_2) \mapsto \rho(r_1)]\,, pc+4)$
`beq`	$b,\ r_1,\ r_2$	(ρ, σ, b) iff $\rho(r_1) = \rho(r_2)$ $(\rho, \sigma, pc+4)$ iff $\rho(r_1) \neq \rho(r_2)$
`add`	$r_1,\ r_2,\ r_3$	$(\rho\,[r_3 \mapsto v]\,, \sigma, pc+4)$ where $v = \rho(r_1) + \rho(r_2)$
`sub`	$r_1,\ r_2,\ r_3$	$(\rho\,[r_3 \mapsto v]\,, \sigma, pc+4)$ where $v = max(0, \rho(r_1) - \rho(r_2))$
`halt`		$\top$ (stuck state)

Figure 2. Semantics of UAL (excluding I/O).

Correspondingly, ψ is a projection that simply removes all privileged operations from a trace.

Once privileged operations are removed from the trace of p and their corresponding emulation from the trace of $[\![mix]\!](sint, p)$, we recover our previous intuitions. We say that $[\![mix]\!](sint, p)$ simulates p *correctly* whenever

$$\forall d \quad \psi(\mathcal{T}_d(p)) \preceq \phi_0(\mathcal{T}_d([\![mix]\!](sint, p))$$

Further, we say that $[\![mix]\!](sint, p)$ simulates p *efficiently* whenever

$$\forall d \quad \phi_0(\mathcal{T}_d([\![mix]\!](sint, p))) \preceq \psi(\mathcal{T}_d(p))$$

Our Jones optimality formulation states that, modulo emulation of privileged instructions, the self-interpreter (VM) executes those and only those operations that the program would if run directly on hardware, *i.e.*

$$Jopt_{=_{VM}}(\cdots) \tag{11}$$

where

$$p =_{VM} p' \quad \triangleq \quad \psi(\mathcal{T}_d(p')) \preceq \phi_0(\mathcal{T}_d(p)) \preceq \psi(\mathcal{T}_d(p'))$$

3. UAL

3.1 Introduction

UAL is an Untyped Assembly Language for an idealized RISC machine. The machine operates on natural numbers, has thirty-two registers, and an infinite store holding both instructions and data. *Viz.*

$$\begin{array}{rcl} v & \in & \mathsf{Value} = \mathbb{N}_0 \\ r & \in & \mathsf{Register} = \{\mathtt{r0}, \mathtt{r1}, \ldots, \mathtt{r31}\} \\ \rho & \in & \mathsf{File} = \mathsf{Register} \to \mathsf{Value} \\ l & \in & \mathsf{Location} = \mathsf{Value} \\ \sigma & \in & \mathsf{Store} = \mathsf{Location} \to \mathsf{Value} \\ \xi & \in & \mathsf{Port} = \mathsf{Value} \end{array}$$

Device I/O is performed via ports (which are akin to channels of process calculi). For example, the instruction

$$\mathtt{in} \quad \langle \mathtt{r0} \rangle,\ \mathtt{r1}$$

reads a value from the device attached to the port identified by register `r0` and stores that value in register `r1`.

Let b range over labels (alphanumeric identifiers in the assembly code) and, by a slight abuse of notation, the corresponding target locations (in machine code).

The syntax of UAL is shown in Figure 1. The `in`, `out`, and `halt` instructions are privileged: when issued by a guest program, they are to be emulated by the VM in keeping with the resource control property of Popek and Goldberg [20].

3.2 Translation to machine code

The translation $(\!|\cdot|\!)$ from instruction mnemonics of Figure 1 to binary machine code is straightforward; we omit a formal definition. Each instruction is assembled to four natural numbers—an opcode, and three operands. For instructions having fewer than three operands, the unused slots are zero-padded. Labels are translated to locations. Let $\#\iota$ denote the opcode corresponding to instruction ι ($\#\mathtt{nop} = 0$, $\#\mathtt{movi} = 1$, and so on), then *e.g.*

$$(\!|\mathtt{movi} \quad 42,\ \mathtt{r1}|\!)\,(l) = l \mapsto \begin{cases} \#\mathtt{movi} & l = 0 \\ 42 & l = 1 \\ 1 & l = 2 \\ 0 & l = 3 \end{cases}$$

3.3 Reduction

The reduction relation ($\rightsquigarrow$) of Figure 2 is defined over configurations. A configuration is a tuple of the form (ρ, σ, pc) where ρ is the register file, σ the store, and $pc \in \mathsf{Location}$ the program counter. The starting configuration for a program p is $(\rho_0, \sigma_0, 0)$ where ρ_0 maps all registers to zero and the store is initialised with values from $(\!|p|\!)$.

The reductions for the `in` and `out` instructions are respectively

$$\begin{array}{llll} \mathtt{in} \quad \langle r_1\rangle,\ r_2 \quad (\rho, \sigma, pc) & \overset{\xi?v}{\rightsquigarrow} & (\rho\,[r_2 \mapsto v]\,, \sigma, pc+4) & \\ & & \text{where } \xi = \rho(r_1) & \\ \mathtt{out} \quad r_1,\ \langle r_2\rangle \quad (\rho, \sigma, pc) & \overset{\xi!v}{\rightsquigarrow} & (\rho, \sigma, pc+4) & \\ & & \text{where } v = \rho(r_1) & \\ & & \text{and } \xi = \rho(r_2) & \end{array}$$

The labels *a la* CCS on the transitions above indicate that a value v is being read from ($\xi?v$) or written to ($\xi!v$) port ξ.

3.4 Self-interpretation

An interpreter is meta-circular if it *"defines each feature of the defined language by using the corresponding feature of the defining language"* [21]. Figures 3 and 4 together form the core of a self-interpreter for UAL which still qualifies as meta-circular but for two exceptions for immediate operands:

1. The `beq` instruction is implemented with a write to the register of the interpreter containing the program counter of the interpreted program.
2. The `movi` instruction is implemented with a memory write.

```
        movi   0, r0              Useful constants.
        movi   1, r1
        movi   L.data, r31
        mov    r31, r2
        movi   #nop, r10
        ...
        movi   #out, r20

L.eval: ld     (r2), r3           Opcode to r3.
        add    r2, r1, r2
        ld     (r2), r4           First operand to r4.
        add    r2, r1, r2
        ld     (r2), r5           Second operand to r5.
        add    r2, r1, r2
        ld     (r2), r6           Third operand to r6.

        add    r2, r1, r2

        beq    L.nop, r3, r10
        ...
        beq    L.out, r3, r20
L.crash: beq   L.crash, r0, r0    Unknown opcode.
```

Figure 3. Setup, decode, and dispatch.

The self-interpreter requires an array of thirty-two memory cells to hold the contents of the interpreted program's registers. For convenience, and due to the relative poverty of our instruction set, we choose to place this array at location zero in the interpreter. A sequence of eight `nop` instructions translates to exactly thirty-two zeros in machine code. Execution falls through the `nop` instructions to the self-interpreter proper, the first part of which is shown in Figure 3. Registers `r0` and `r1` hold useful constants: zero and one, respectively. Register `r31` is loaded with the location in memory of the interpreted program (which is labeled `L.data` for convenience). The program counter is kept in register `r2`. Registers `r10` through `r20` are populated with instruction opcodes. Finally, each instruction is fetched, decoded (the four values comprising each one loaded into registers `r3` through `r6`), and dispatched.

The machine code of the interpreted program is appended to the self-interpreter and labeled `L.data`.

3.5 Correctness

We argue, informally, that our self-interpreter is correct, that is

$$\forall d \quad [\![sint]\!](p, d) = [\![p]\!](d)$$

Define observational equivalence of UAL programs over sequences of values read from (or written to) I/O ports. Let $\mathcal{S}_d(p)$ be the abstraction of trace $\mathcal{T}_d(p)$ (Section 2.1) comprising the sequence of port number/value tuples (of the form $\xi?v$ and $\xi!v$) representing the interaction of program p with I/O devices when run on input d.

Two programs are IO-equivalent when they interact with I/O devices in the same way:

$$p =_{IO} p' \quad \triangleq \quad \forall d \quad \mathcal{S}_d(p) = \mathcal{S}_d(p')$$

From Figure 4 it is clear that `in` and `out` instructions are issued by the interpreter in the same order and with the same operands as would be the case had the program been executed directly.

3.6 Efficiency

Despite being correct, our self-interpreter would not make a good VM. In Figure 4, instructions that do useful work are boxed: every other instruction contributes to interpretational overhead.

```
L.nop:  [nop]
        beq    L.eval, r0, r0

L.movi: [st]   r4, (r5)
        beq    L.eval, r0, r0

L.mov:  ld     (r4), r4
        [mov]  r4, r7
        st     r7, (r5)
        beq    L.eval, r0, r0

L.ld:   ld     (r4), r4
        add    r4, r31, r4
        [ld]   (r4), r4
        st     r4, (r5)
        beq    L.eval, r0, r0

L.st:   ld     (r5), r5
        add    r5, r31, r5
        ld     (r4), r4
        [st]   r4, (r5)
        beq    L.eval, r0, r0

L.beq:  ld     (r5), r5
        ld     (r6), r6
        beq    L.t, r5, r6
        beq    L.eval, r0, r0
L.t:    [add]  r4, r31, r2
        beq    L.eval, r0, r0

L.add:  ld     (r4), r4
        ld     (r5), r5
        [add]  r4, r5, r7
        st     r7, (r6)
        beq    L.eval, r0, r0

L.sub:  ld     (r4), r4
        ld     (r5), r5
        [sub]  r4, r5, r7
        st     r7, (r6)
        beq    L.eval, r0, r0

L.halt: [halt]

L.in:   ld     (r4), r4
        [in]   ⟨r4⟩, r7
        st     r7, (r5)
        beq    L.eval, r0, r0

L.out:  ld     (r4), r4
        ld     (r5), r5
        [out]  r4, ⟨r5⟩
        beq    L.eval, r0, r0

L.data: ⟨Machine code of interpreted program⟩
```

Figure 4. A meta-circular self-interpreter.

At this point, if we wish to improve the efficiency of our self-interpreter as a VM, we can either *(a)* leave the self-interpreter as is and build a good specializer for UAL, or *(b)* settle for a trivial specializer and achieve Jones optimality (in the sense defined in Section 2) by introducing virtualization assists to UAL. Conceptually, the latter path is the one taken by the virtualization community and we explore it in the next section.

4. Virtualization Assists

In this section we add support for trap-and-emulate virtualization to UAL.

4.1 UAL/`step`

Volume 3B of the *Intel 64 and IA-32 Architectures Software Developer's Manual*[1] states that (p. 18-12):

> *"The processor generates a single-step debug exception if (while an instruction is being executed) it detects that the TF flag in the EFLAGS register is set. [. . .] the exception is generated after the instruction is executed."*

To speed-up our self-interpreter/VM on non-privileged instructions without relinquishing control over execution of privileged ones, we introduce the `step` instruction into UAL. As the name suggests, the instruction allows us to single-step through guest program code:

$$\iota \quad ::= \quad \texttt{step} \quad r$$

The only operand to the `step` instruction is the location of a *VM Control Block (VMCB)*. Recall that our meta-circular self-interpreter uses the first thirty-two locations in memory to store the registers of the interpreted program. The VMCB is a variant of that structure and plays a role similar to a process control block in an operating system kernel. We leave the precise format of the VMCB and a formal semantics for UAL/`step` to future work. However, note that the VMCB contains a value for the program counter and a value for the *memory base*.

The `step` instruction saves the values of the registers in the VMCB and loads new values from the VMCB. It then executes a *single* non-privileged instruction at the location identified by the program counter taking care to offset all memory accesses (including branch targets) performed by this instruction with the value of the *memory base*. On completion, the current state (registers and program counter) is saved in the VMCB, the old one restored from the VMCB and execution continues with the instruction following `step`.

Note that the particular interpretational encoding of each instruction in the meta-circular interpreter is now uniformly captured (and one hopes efficiently implemented) by `step`. A self-interpreter for UAL/`step` is shown in Figure 5. It still fails to be efficient due to the overhead of instruction dispatch. We now introduce an alternative instruction to `step` closer to the spirit of trap-and-emulate virtualization.

4.2 UAL/`eval`

UAL/`eval` extends UAL with the `eval` instruction:

$$\iota \quad ::= \quad \texttt{eval} \quad r$$

The only operand to the `eval` instruction is the location of a VMCB. The `eval` instruction is similar to `step`, however, it executes any number of instructions rather than just a single one. If a privileged instruction is encountered, the state is restored and execution continues at the instruction following `eval` which may then determine the opcode and operands of the offending instruction via the program counter field of the VMCB.

```
          ⟨Setup as in top of Figure 3⟩
          movi  L.vmcb, r31

L.eval:   ⟨Load PC field of VMCB to r2⟩
          ld    (r2), r3          Opcode to r3.

          Check for privileged instructions:
          beq   L.halt, r3, r18
          beq   L.in, r3, r19
          beq   L.out, r3, r20
          step  r31
          beq   L.eval, r0, r0

L.halt:   ⟨Emulation of halt⟩
L.in:     ⟨Emulation of in⟩
L.out:    ⟨Emulation of out⟩

L.vmcb:   ⟨VMCB⟩
L.data:   ⟨Machine code of interpreted program⟩
```

Figure 5. A self-interpreter for UAL/`step`.

```
          ⟨Setup as in top of Figure 3⟩
          movi  L.vmcb, r31

L.eval:   eval  r31

          ⟨Retrieve opcode and operands from VMCB⟩

          beq   L.halt, r3, r18
          beq   L.in, r3, r19
          beq   L.out, r3, r20

L.halt:   ⟨Emulation of halt⟩
L.in:     ⟨Emulation of in⟩
L.out:    ⟨Emulation of out⟩

L.vmcb:   ⟨VMCB⟩
L.data:   ⟨Machine code of interpreted program⟩
```

Figure 6. A self-interpreter for UAL/`eval`.

A self-interpreter for UAL/`eval` is shown in Figure 6. The `eval` instruction increments the current virtualization nesting level. Therefore, in the trace, instructions that are executed by `eval` are marked with the new virtualization level. On encountering a privileged instruction, the virtualization nesting level is decremented prior to returning control to the callee. Thus, the sections of interpreter code which are responsible for emulating the privileged instructions are executed at the same virtualization level as the VM, whereas those of the program are executed at the higher level.

Recall from Section 2, that in our definition of Jones optimality (Equation 11) we strip out the emulation by means of the ϕ_0 projection.

4.3 Jones optimality revisited

Starting with a meta-circular interpreter which has most control over guest program execution, we next have an interpreter using the `step` instruction which relinquishes some control in order to gain performance. Finally, the interpreter using `eval` does not

[1] `http://www.intel.com/products/processor/manuals/index.htm`

handle instructions directly but merely accepts "callbacks" from the hardware for the privileged instructions. Thus we finally invert the instruction dispatch loop: whereas before the interpreter was responsible for dispatch, now the hardware is.

Our observation is that this sequence of steps gradually eliminates interpretational overhead and yields an interpreter with respect to which even the trivial mix is Jones-optimal in the sense of our Equation 11.

5. Towards Full Abstraction for Virtual Machines

Ideally a program should not be able to detect whether or not it is executing within a virtual machine. This is not merely a matter of principle, but is necessary to allow, for instance, safe and reliable analysis of malware. An observable deviation between two known-equivalent programs is a litmus test for virtual machine presence. Garfinkel et al. [9] define VM *transparency* as: *"making virtual and native hardware indistinguishable under close scrutiny by a dedicated adversary"* and argue on pragmatic grounds that it is not an achievable goal on real-world, networked, x86 machines.

Full abstraction is best known as a notion of agreement between the denotational and operational semantics of a particular language. Abadi [2] suggested that a safe language *translation* should be *fully abstract*. That is given a translation $\mathcal{C}[\![\cdot]\!]$ from S to T, the following should hold:

$$e \sim_S e' \iff \mathcal{C}[\![e]\!] \sim_T \mathcal{C}[\![e']\!] \tag{12}$$

where $\sim_S$ and $\sim_T$ are observational equivalence relations over S and T respectively. Intuitively, the definition says that translation does not permit an adversarial T-context to glean more information than any S-context might. Notice that full abstraction enforces a notion of continuity in compilation: a malicious compiler, such as that of Thompson [25], must either miscompile all programs in an equivalence class or none at all.

The first Futamura projection lets us perform translation. Instantiate Equation 12 for a self-interpreter $sint$ and a program specializer mix:

$$p \sim_L p' \iff [\![mix]\!](sint, p) \sim_L [\![mix]\!](sint, p') \tag{13}$$

where p and p' are programs of language L and $[\![\cdot]\!]$ a valuation function on L. Now, if we let $\sim_L$ be $\leq_{time}$ (Section 2), we ensure that the relative speed of programs is preserved: that is a program q may not determine whether or not it is executing within a VM by running p and p' and comparing their execution times.

We intuit that full abstraction may prove to be a useful notion for reasoning about the safety of virtual machines. We plan to pursue this direction in future work.

6. Related Work

We give a few pointers to relevant material on virtualization and Jones optimality.

Virtualization. The landmark paper of Popek and Goldberg [20] (see also discussion of same in [24, ch. 8]) establishes architectural requirements for virtualization in a formal manner.

Robin and Irvine [22] investigate the feasibility of secure virtual machines on the Intel Pentium. Intel's VT virtualization extensions for the x86 architecture and the Itanium are described by Neiger et al. [18]. Adams and Agesen [3] discuss the pros and cons of these hardware assists compared with previously used techniques for x86 virtualization.

Jones optimality. Makholm [15] provides a good discussion of Jones optimality. More recently, Danvy and López [7] established a link between Jones-optimal specialization and higher-order abstract syntax. Glück [10] showed that for any Jones-optimal specializer in a particular class, for any given translation, there exists an interpreter that under specialization will yield programs "no worse" than the translation. Gade and Glück [8] give a formal argument of Jones optimality for the specializer Unmix.

'Trusting trust' and friends. Translation discontinuities were exhibited by Thompson's trojan compiler [25] which inserted spurious malicious code when compiling itself or the `login` program.

Mitchell [16] introduced "abstraction-preserving reductions" as a means of comparing the expressiveness of programming languages. Abadi [2] pointed out that full abstraction is generally useful as a benchmark of safety and correctness for translations. Kennedy [14] showed several cases where full abstraction fails for a C# to .NET bytecode compiler.

Instrumentation. Jones [12, Section 2.3.3], in describing practical concerns of constructing interpreters for specialization, notes that *"specialising an instrumented self-interpreter to a source program has the effect of inserting instrumentation code into the body of the source program"*. It is unclear precisely how modern instrumentation packages such as DTrace [6] and Valgrind [19] can be understood in terms of interpreter specialization. Jones optimality for traces seems a promising approach to addressing this.

Siskind and Pearlmutter [23] introduced the `map-closure` construct which allows identifiers in the environment of a closure to be rebound, thus permitting non-standard interpretation. The relationship of `map-closure` to non-standard interpretation techniques developed by the partial evaluation community is, to our knowledge, unexplored.

Meta-circularity. The term "meta-circular interpreter" is due to Reynolds [21] who also demonstrated how a meta-circular interpreter can be disambiguated using defunctionalization and CPS conversion.

7. Conclusions and Further Work

We have described the current state of our efforts to relate well-established concepts from partial evaluation with an important practical application: virtualization. We showed how the well-known virtualization requirements of Popek and Goldberg [20] can be understood in terms of Jones-optimal specialization. We introduced UAL, an assembly language for an idealized RISC machine with I/O devices. We showed a self-interpreter for UAL as well as self-interpreters for two alternative extensions of UAL with hardware virtualization assists: UAL/`step` and UAL/`eval`. We argued that, in our framework, the trivial specializer is Jones-optimal with respect to the self-interpreter for UAL/`eval`. Finally, we suggested how the notion of full abstraction can be applied to virtual machines.

It would also be interesting to explore development of typed assembly languages with privileged instructions as a first step towards formalizing a minimal trusted virtual machine, much in the spirit of the work on proof-carrying code [17, 4].

Acknowledgments

We thank members and friends of the CPRG and the PEPM 2008 referees for helpful comments and pointers to related work. The first author gratefully acknowledges an EPSRC studentship.

References

[1] VMware Website. `http://www.vmware.com/`.

[2] Martín Abadi. Protection in programming-language translations. In *Proceedings of ICALP*, volume 1443 of *LNCS*, pages 868–883, 1998.

[3] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of ASPLOS*, pages 2–13, 2006.

[4] Andrew W. Appel. Foundational proof-carrying code. In *Proceedings of LICS*, 2001.

[5] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of SOSP*, pages 164–177, 2003.

[6] Bryan Cantrill, Michael W. Shapiro, and Adam H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX ATC*, pages 15–28, 2004.

[7] Olivier Danvy and Pablo E. Martínez López. Tagging, encoding, and Jones optimality. In *Proceedings of ESOP*, volume 2618 of *LNCS*, pages 335–347, 2003.

[8] Johan Gade and Robert Glück. On Jones-optimal specializers: A case study using Unmix. In *Proceedings of APLAS*, volume 4279 of *LNCS*, pages 406–422, 2006.

[9] Tal Garfinkel, Keith Adams, Andrew Warfield, and Jason Franklin. Compatibility is not transparency: VMM detection myths and realities. In *Proceedings of HotOS*, 2007.

[10] Robert Glück. The translation power of the Futamura projections. In *Perspectives of Systems Informatics*, volume 2890 of *LNCS*, pages 133–147, 2003.

[11] Neil D. Jones. Challenging problems in partial evaluation and mixed computation. *New Generation Comput.*, 6(2&3): 291–302, 1988.

[12] Neil D. Jones. Transformation by interpreter specialisation. *Science of Computer Programming*, 52:307–339, 2004.

[13] Neil D. Jones, Carsten K. Gomard, and Peter Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall, 1993.

[14] Andrew Kennedy. Securing the .NET programming model. *Theoretical Computer Science*, 364(3):311–317, 2006.

[15] Henning Makholm. On Jones-optimal specialization for strongly typed languages. In *Proceedings of SAIG*, volume 1924 of *LNCS*, pages 129–148, 2000.

[16] John C. Mitchell. On abstraction and the expressive power of programming languages. *Science of Computer Programming*, 21(2):141–163, 1993.

[17] George C. Necula. Proof-carrying code. In *Proceedings of POPL*, pages 106–119, 1997.

[18] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. Intel Virtualization Technology: Hardware support for efficient processor virtualization. *Intel Technology Journal*, 10(3):167–177, 2006. `http://dx.doi.org/10.1535/itj.1003.01`.

[19] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of PLDI*, pages 89–100, 2007.

[20] Gerald J. Popek and Robert P. Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[21] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998. Reprint of a 1972 paper.

[22] John Scott Robin and Cynthia E. Irvine. Analysis of the Intel Pentium's ability to support a secure virtual machine monitor. In *Proceedings of USENIX Security Symposium*, 2000.

[23] Jeffrey Mark Siskind and Barak A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *Proceedings of POPL*, pages 71–76, 2007.

[24] James E. Smith and Ravi Nair. *Virtual machines: versatile platforms for systems and processes*. Morgan Kaufmann, 2005.

[25] Ken Thompson. Reflections on trusting trust. *Communications of the ACM*, 27(8):761–763, 1984.