Darek Mihocka, Emulators.com
Stanislav Shwartsman, Intel Corp.

June 21 2008

Virtualization Without Direct Execution: Designing a Portable VM

Agenda

- Introduction
- Gemulator
- Bochs
- Proposed ISA Extensions
- Conclusions and Future Work
- Q & A

Introduction

- A virtual machine is an indirection engine which redirects code and data inside of the “guest” sandbox.

- Three ways of virtual machine implementation:
  - Virtualization, direct execution (VMware, Virtual PC, Xen)
  - Dynamic (just-in-time) translation (QEMU)
  - Emulation (Bochs, Gemulator)

- Recent trend in x86 virtualization products to rely on hardware VT for hypervisor implementation on the “host” – requires use of very recent microprocessors.

- Other techniques like “ring compression” and dynamic recompilation – still very x86 or host specific.

A Portable VM

- A portable VM cannot rely on specific model of host CPU, or advanced features of CPU such as MMU.
- Interpretation based techniques can be used to implement portable VM, even using high level languages – C or C++.
- But we show that efficiently written emulation engine can be nearly as fast as a virtual machine implemented using dynamic translation.
- We choose portability over maximizing peak performance!

Benefits of Portable VM

- Instrumentation of memory accesses, flow control, and context switches becomes easier and performance efficient.
- Allows for simulation of future ISA extensions.
- Bounds memory overhead for memory constrained hosts.

Portability Means Isolation

- Most virtual machines today do NOT isolate the guest virtual machine from the host CPU due to use of direct execution or jitting.
- Information such as CPUID bits or ISA capability leaks through to guest.
- Only a truly portable virtual machine isolates everything, providing complete transparency.*

Overview of Presentation

- A look at implementation of Gemulator – a 68040 Macintosh emulator for x86
  - Efficient byte swapping
  - Efficient guest-to-host address translation
- A look at implementation of Bochs – a portable open source x86 PC emulator
  - Caching of decoded instructions
  - Lazy flags
- Proposed ISA extensions based on commonalities in Bochs and Gemulator
- Conclusions and future work

Gemulator Byte Swap

- Cannot rely on BSWAP functionality in C/C++ or for large data types.
- 68040 address space is thus stored *backwards* in x86 host address space.
- In most trivial implementation, entire 68000/68040 address space is allocated as one memory block.
- Guest address is negated to calculate the host access address.

Trivial Byte Swap Math

- Guest block of size M is allocated at host address B.
- Guest address G maps to host address:

$$H = B + M - 1 - G$$

- In general, guest access of N bytes maps as:

$$H = B + M - N - G$$

- Works for unaligned accesses!
- If B and M are large powers of 2, can use constant K: H = G XOR K

Page Based XOR Translation

- Using XOR for guest-to-host mapping, guest address space can be allocated in smaller discontiguous blocks.
- Each such block has a unique XOR constant.
- These XOR values may be stored in an array – one entry per guest page.

Software TLB using XOR

- Storing XOR values as small lookup table is software equivalent of a Translation Lookaside Buffer (TLB).
- 96%+ hit rate using 2048-entry table.
- Separating tables for code and data access catches guest self-mod code.
- Mapping granularity need not be 4K.
- Mapping function currently implemented using 10 x86 instructions, one branch!

Bochs Basics

- Highly portable open source IA32 PC emulator written purely in C++. Emulates x86 CPU and common I/O devices.
- Similar to QEMU, Xen, and VMware Workstation.
- But, does not use jitting or hardware virtualization.
- X86 Execution is purely interpreted.

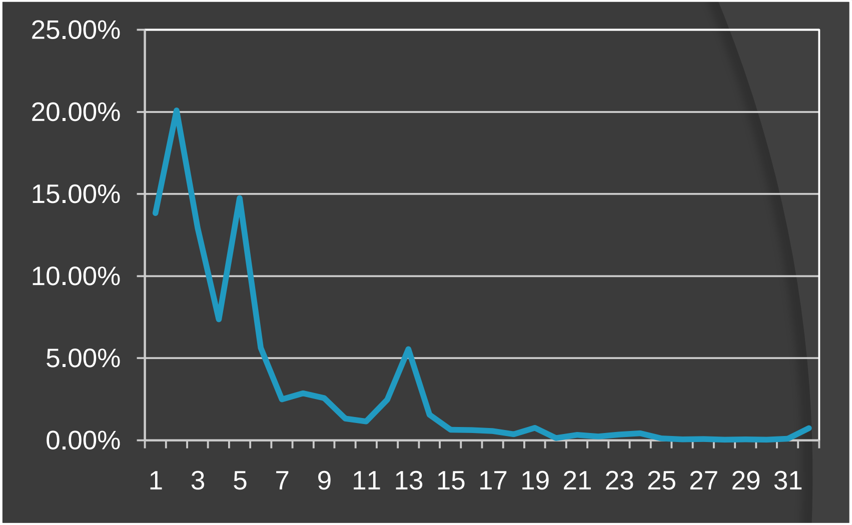
Bochs Trace Cache

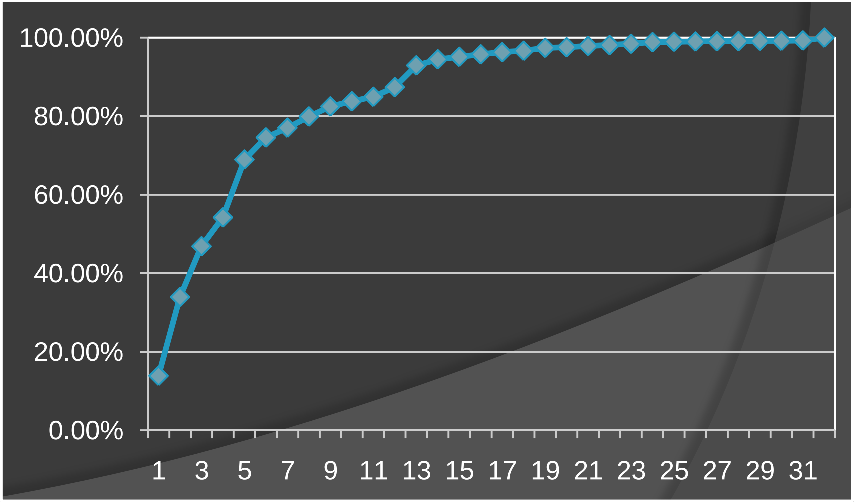
- Bochs 2.3.5 spent >50% of time in fetch-decode-dispatch CPU loop.
- Decoded instructions cached in simple direct mapped i-cache when single i-cache entry contains single decoded instruction.
- Every instruction should pay a price of i-cache lookup

But why not cache a decoded basic block at once?

Bochs Trace Cache

- 32K entries associated into direct map cache.
- Use fine tuned hash function to index cache entries.
- Trace length is virtually unlimited, traces allocated from static memory pool while optimizing for host data cache locality.





Reducing Misprediction

- A mispredicted branch can cost over 20 host clock cycles on modern CPUs.
- To reduce misprediction, Bochs tries to eliminate “if/else” statements that host hardware will not be able to predict, using techniques such as:
  - Replicating instruction execution handlers for register and memory forms of an instruction.
  - Moving effective address calculation out of main CPU loop and into the instruction handlers.
  - Merging similar effective address calculation code into common functions by using more general form of calculation.

Emulating EFLAGS in C++

- Most virtual machines resort to using inline PUSHF/POP or LAHF instructions to capture arithmetic flags – not portable!
- Bochs (and Gemulator) use lazy flags approach and calculate arithmetic flags values only when required, using only basic integer operations.
- Flags can be derived by caching the sign-extended values of input operands and the result.

Lazy Flags

- Based strictly on the cached result, can derive the ZF (Zero Flag), SF (Sign Flag), and PF (Parity Flag):
  - ZF = (result == 0);
  - SF = (result < 0);
  - PF = parity_lookup[result & 0xFF];
- This is *faster* than using inline ASM executing a PUSHF/POP or LAHF!

Lazy Flags II

- CF (Carry Flag), OF (Overflow Flag), and AF (Adjust Flag) are all derived from carry-out bits from different bit positions.
- AF is carry out of 4th LSB, thus:
  - *AF = ((op1 ^ op2) ^ result) & 0x10*
- OF and CF are based on sign changes between inputs and result:
  - *OF = ((op1 ^ op2) & (op1 ^ result)) < 0*
  - *CF = (result ^ (~(op1 ^ op2) & (op1 ^ result))) < 0*

Bochs Benchmarks

	1000 MHz Pentium III	2533 MHz Pentium 4	2666 MHz Core 2 Duo
Bochs 2.3.5	882	595	180
Bochs 2.3.6	609	533	157
Bochs 2.3.7	457	236	81

- Time (in seconds) to boot Windows XP guest using three different Intel host architectures.
- 2x improvement from Bochs 2.3.5 by using the techniques just described!

Proposed ISA Extensions

- In place of existing MMU, segmentation, and VT, we suggest some simple ISA extensions instead.
- The ISA extensions could be targeted as C++ compiler intrinisics or by jitters to achieve faster speeds for interpreters and binary translated code.
- ISA extensions aim at two goals – speed up guest-to-host mapping, and flags.

Accelerating Software TLB

- Matching an entry in TLB involves a hashing operation and key match to retrieve correct value.
- Suggest a HASHLU (Hash Lookup) instruction of the form:

```
hashlu eax, dword ptr [ebp], flags
jne  no_match
```

- HASHLU is essentially a programmatic use of the hardware TLB.
- Propose an instruction SAFL (Store Arithmetic Flags) which stores just the arithmetic flags to a register or memory.
- Could be implemented as a complier intrinsic or automatically generated by compilers to accelerate interpreters and accelerate binary translated code.

Conclusions

- C++ based interpreter can achieve 100 MIPS execution speed today.
- Byte swapping, memory translation, arithmetic flags, and instruction dispatch can be implemented efficiently and in a portable way in C++.
- Benchmarks show that efficient emulation can be within 2x speed of dynamic translation implementations.
- Interpreter can do much of the work on a jitter – caching decoded instruction, constructing traces, etc. – but simply stops short of emitting new host code.
- This technique is known as a “threaded interpreter”.

Future Work

- Further research to try to achieve 200 MIPS.
- Porting Bochs and Gemulator to Sony Playstation 3 and PowerMac G5.
- Using Bochs as a general purpose instrumentation tool similar to DynamoRIO, Pin, and Nirvana, but possibly with less overhead.
- Using fine-grained mapping to efficiently compact a large guest into a small host – for example: Vista on an ASUS EEE or PS/3.

Q&A

Backup Slides

Properties of Portable VM

- Portable across x86 and non-x86 hosts.
- Bounds memory overhead for memory constrained hosts.
- Bounds worst-case performance for predictable execution speed.
- Efficiently dispatches guest code instructions, regardless of host ISA.
- Efficiently handlers data accesses, privilege checks, and byte swap issues.

Interpretation II

- Expensive operations such as division, interlocked memory operations, disk I/O and etc. really do not benefit from jitting or direct execution anyway.
- Jitting may add megabytes of extra memory overhead to the host, decreasing L1 and L2 hit rates.
- An interpreter already does the work of decoding an instruction. Adding instrumentation is minimal extra work.

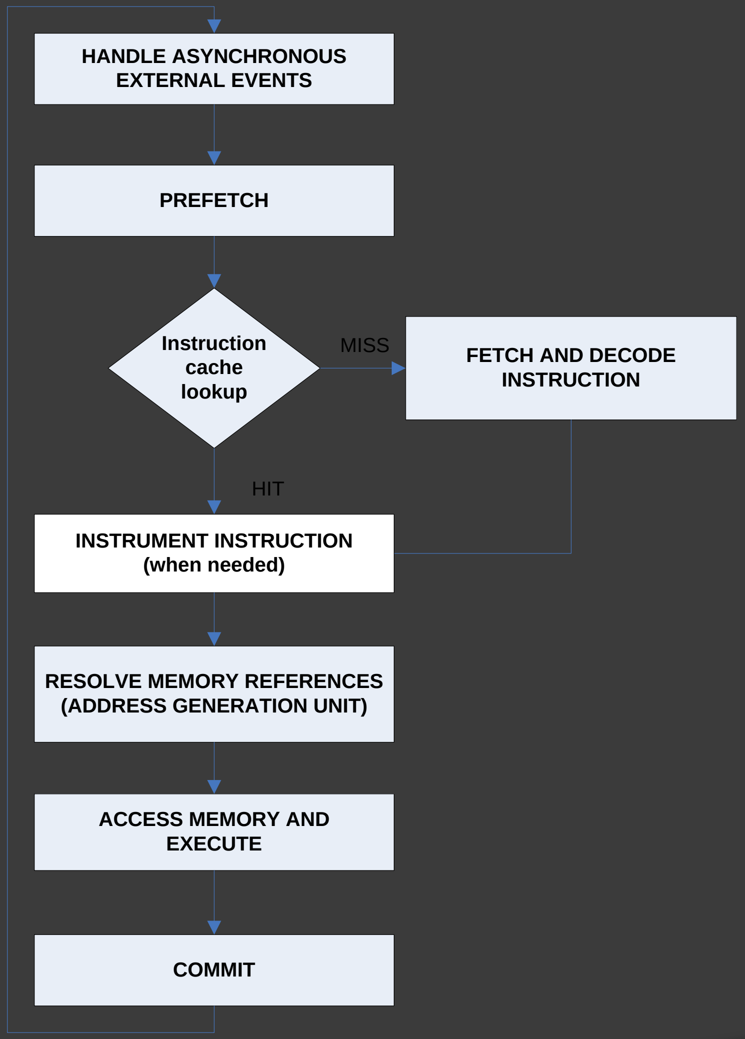
Bochs Internals

- Mimics everything the real CPU does

  Emulate CPU fetch-decode-execute flow

- Fetch:
  - At prefetch stage, check permissions and update page timestamps for self-mod code detection.
  - Fetch x86 opcode.
- Decode
  - Decode x86 instruction into internal representation.
- Execute
  - Calculate effective address of memory operands.
  - Indirect call to instruction execution method.
  - Update the register state and flags as necessary

Bochs CPU Loop



HANDLE ASYNCHRONOUS EXTERNAL EVENTS

PREFETCH

Instruction cache lookup

MISS

FETCH AND DECODE INSTRUCTION

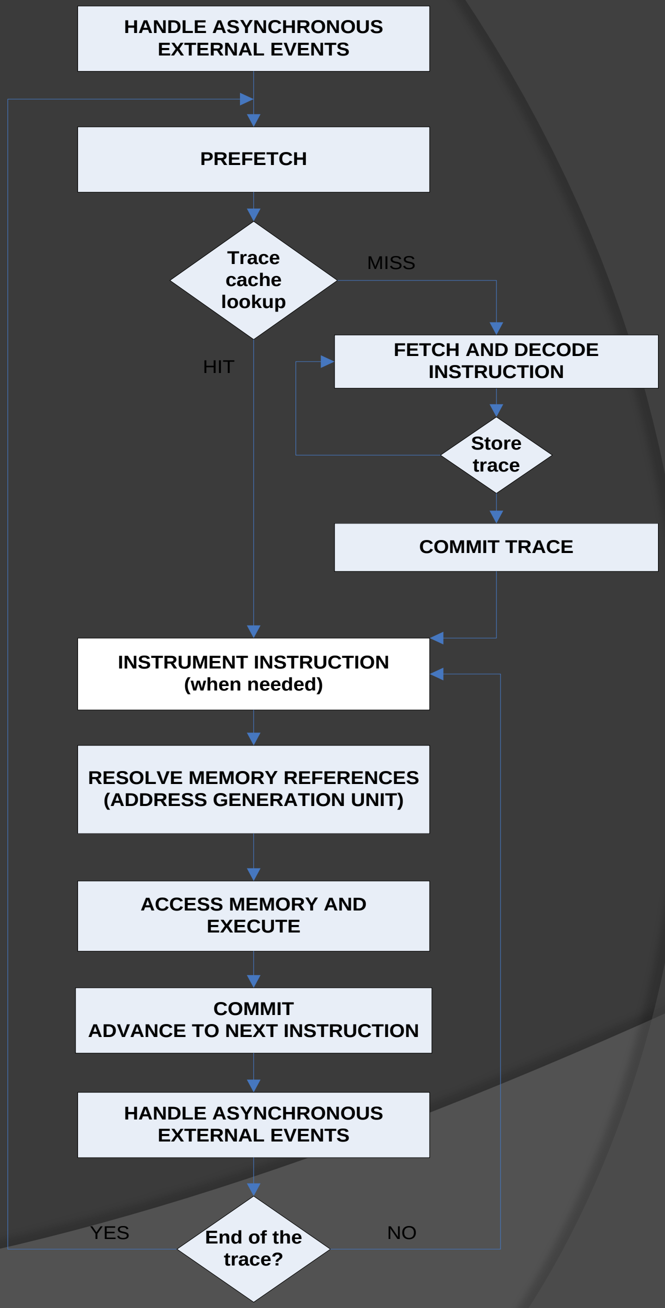
HIT

INSTRUMENT INSTRUCTION (when needed)

RESOLVE MEMORY REFERENCES (ADDRESS GENERATION UNIT)

ACCESS MEMORY AND EXECUTE

COMMIT



HANDLE ASYNCHRONOUS EXTERNAL EVENTS

PREFETCH

Trace cache lookup

MISS

FETCH AND DECODE INSTRUCTION

Store trace

COMMIT TRACE

HIT

INSTRUMENT INSTRUCTION (when needed)

RESOLVE MEMORY REFERENCES (ADDRESS GENERATION UNIT)

ACCESS MEMORY AND EXECUTE

COMMIT
ADVANCE TO NEXT INSTRUCTION

HANDLE ASYNCHRONOUS EXTERNAL EVENTS

YES

End of the trace?

NO

Gemulator Basics

- 68000/68040 interpreter for MS-DOS and Windows which emulates Apple Macintosh.
- Needs to handle 68040-to-x86 byte swap for all access sizes.
- Needs to handle mapping of up to 1GB of 68040 RAM to possibly fragmented host address space.
- Detect and handle self-modifying 68000 code, very common in older Macintosh applications.
- To run on MS-DOS, must not generate any host exceptions or faults!