Virtualization Without Direct Execution or Jitting: Designing a Portable Virtual Machine Infrastructure

Darek Mihocka
Emulators
darekm@emulators.com

Stanislav Shwartsman
Intel Corp.
stanislav.shwartsman@intel.com

Abstract

A recent trend in x86 virtualization products from Microsoft, VMware, and XenSource has been the reliance on hardware virtualization features found in current 64-bit microprocessors. Hardware virtualization allows for direct execution of guest code and potentially simplifies the implementation of a Virtual Machine Monitor (or "hypervisor")[1]. Until recently, hypervisors on the PC platform have relied on a variety of techniques ranging from the slow but simple approach of pure interpretation, the memory intensive approach of dynamic recompilation of guest code into translated code cache, to a hardware assisted technique known as "ring compression" which relies on the host MMU for hardware memory protection. These techniques traditionally either deliver poor performance[2], or are not portable. This makes most virtualization products unsuitable for use on cell phones, on ultra-mobile notebooks such as ASUS EEE or OLPC XO, on game consoles such as Xbox 360 or Sony Playstation 3, or on older Windows 98/2000/XP class PCs.

This paper describes ongoing research into the design of portable virtual machines which can be written in C or C++, have reasonable performance characteristics, could be hosted on PowerPC, ARM, and legacy x86 based systems, and provide practical backward compatibility and security features not possible with hardware based virtualization.

Keywords

Bochs, Emulation, Gemulator, TLB, Trace Cache, Virtualization

1.0 Introduction

At its core, a virtual machine is an indirection engine which intercepts code and data accesses of a sandboxed "guest" application or operating system and remaps them to code sequences and data accesses on a "host" system. Guest code is remapped to functionally identical code on the host, using binary translation or sandboxed direct execution. Guest data accesses are remapped to host memory and checked for access privilege rights, using software or hardware supported address translation.

When a guest virtual machine and host share a common instruction set architecture (ISA) and memory model, this is commonly referred to as "virtualization". VMware Fusion [3] for running Windows applications inside of Mac OS X, Microsoft's Hyper-V [4] feature in Windows Server 2008, and Xen [5] are examples of virtualization products. These virtual machines give the illusion of full-speed direct execution of native guest code. However, the code and data accesses in the guest are strictly monitored and controlled by the host's memory management hardware. Any attempt to access a memory address not permitted to the guest results in an exception, typically an access violation page fault or a "VM exit event", which hands control over to the hypervisor on the host. The faulting instruction is then emulated and either aborted, re-executed, or skipped over. This technique of virtualization is also known as "trap-and-emulate" since certain guest instructions must be emulated instead of executed directly in order to maintain the sandbox.[6].

Virtualization products need to be fast since their goal is to provide hardware-assisted isolation with minimal runtime overhead, and therefore generally use very specific assembly language optimizations and hardware features of a given host architecture.

A more general class of virtual machine is able to handle guest architectures which differ from the host architecture by emulating each and every guest instruction. Using some form of binary translation, either bytecode interpretation or dynamic recompilation or both, such virtual machines are able to work around differences in ISA, memory models, endianness, and other differentiating factors that prevent direct execution.

Emulation techniques generally have a noticeable slowdown compared to virtualization, but have benefits such as being able to easily capture traces of the guest code

or inject instrumentation. Dynamic instrumentation frameworks such as Intel's Pin [7] and PinOS [8], Microsoft's Nirvana[9], PTLsim[10], and DynamoRIO[11] programmatically intercept guest code and data accesses, allowing for the implementation of reverse execution debugging and performance analysis tools. These frameworks are powerful but can incur orders of magnitude slowdown due to the guest-to-host context switching on each and every guest instruction.

Emulation products also end up getting customized in host-specific ways for performance at the cost of portability. Apple's original 68020 emulator for PowerPC based Macs [12] and their more recent Rosetta engine in Mac OS X which run PowerPC code on x86 [13] are examples of much targeted pairings of guest and host architectures in an emulator.

As virtualization products increasingly rely on hardware-assisted acceleration for very specialized usage scenarios, their value diminishes for use on legacy and low-end PC systems, for use on new consumer electronics platforms such as game consoles and cell phones, and for instrumentation and analysis scenarios. As new devices appear, time-to-market may be reduced by avoiding such one-off emulation products that are optimized for a particular guest-host pairing.

1.1 Overview of a Portable Virtual Machine Infrastructure

For many use cases of virtualization we believe that it is a fundamental design flaw to merely target the maximization of best-case performance above all else, as has been the recent trend. Such an approach not only results in hardware-specific implementations which lock customers into limited hardware choices, but the potentially poor worst-case performance may result in unsatisfactory overall performance[14].

We are pursuing a virtual machine design that delivers fast CPU emulation performance but where portability and versatility are more important than simply maximizing peak performance.

We tackled numerous design issues, including:

- Maintaining portability across legacy x86 and non-x86 host systems, and thus eliminating the use of host-dependent optimizations,
- Bounding the memory overhead of a hypervisor to allow running in memory constrained environments such as cell phones and game consoles,
- Bounding worst-case performance, and thus allowing for efficient tracing and run-time analysis,
- Efficiently dispatching guest instructions on the host,
- Efficiently mapping guest memory to the host, and,
- Exploring simple and lightweight hardware acceleration alternatives.

Our research on two different virtual machines – the Bochs portable PC emulator which simulates both 32-bit x86 and x86-64 architectures, and the Gemulator[15] Atari ST and Apple Macintosh 68040 emulator on Windows – shows that in both cases it is possible to achieve full system guest simulation speeds in excess of 100 MIPS (millions of instructions per second) using purely interpreted execution which does not rely on hardware MMU capabilities or even on dynamic recompilation. This work is still in progress, and we believe that further performance improvements are possible using interpreted execution to where a portable virtual machine running on a modern Intel Core 2 or PowerPC system could achieve performance levels equal to what less than ten years ago would have been a top of the line Intel Pentium III based desktop.

A portable implementation offers other benefits over vendor specific implementations, such as deterministic execution, i.e. the ability to take a saved state of a virtual machine and re-execute the guest code with deterministic results regardless of the host. For example, it would be highly undesirable for a virtual machine to suddenly behave differently simply because the user chose to upgrade his host hardware.

Portability suggests that a virtual machine's hypervisor should be written in a high level language such as C or C++. A portable hypervisor needs to support differences in endianness between guest and host, and even differences in register width and address space sizes between guest and host should not be blocking issues. An implementation based on a high level language must be careful to try to maintain a bounded memory footprint, which is better suited for mobile and embedded devices.

Maintaining portability and bounding the memory footprint led to the realization that dynamic recompilation (also known as just-in-time compilation or "jitting") may not deliver beneficial speed gains over a purely interpreted approach. This is due to various factors, including the very long latencies incurred on today's microprocessors for executing freshly jitted code, the greater code cache and L2 cache pressure which jitting produces, and the greater cost of detecting and properly handling self-modifying code in the guest. Our approach therefore does not rely on jitting as its primary execution mechanism.

Supporting the purely-interpreted argument is an easily overlooked aspect of the Intel Core and Core 2 architectures: the stunning efficiency with which these processors execute interpreted code. In benchmarks first conducted in 2006 on similar Windows machines, we found that a 2.66 GHz Intel Core 2 based system will consistently deliver two to three times the performance of a 2.66 GHz Intel Pentium 4 based system when running interpretation based virtual machines such as SoftMac[16]. Similar results have been seen with Gemulator, Bochs, and other interpreters. In one hardware generation on Intel microprocessors, interpreted virtual machines make a lot more sense.

Another important design goal is to provide guest instrumentation functionality similar to Pin and Nirvana, but with less of a performance penalty when such instrumentation is active. This requires that the amount of context switching involved between guest state and host state be kept to a minimum, which once again points the design at an interpreted approach. Such a low-overhead instrumentation mechanism opens the possibilities to performing security checks and analysis on the guest code as it is running in a way that is less intrusive than trying to inject it into the guest machine itself. Imagine a virus detection or hot-patching mechanism that is built into the hypervisor which then protects otherwise vulnerable guest code. Such a proof-of-concept has already been demonstrated at Microsoft using a Nirvana based approach called Vigilante[17]. Most direct execution based hypervisors are not capable of this feat today.

Assumptions taken for granted by virtual machine designs of the past need to be re-evaluated for use on today's CPU designs. For example, with the popularity of low power portable devices one should not design a virtual machine that assumes that hardware FPU (floating point unit) is present, or even that a hardware memory management is available.

Recent research from Microsoft's Singularity project [18] shows that software-based memory translation and isolation is an efficient means to avoid costly hardware context switches. We will demonstrate a software-only memory translation mechanism which efficiently performs guest-to-host memory translation, enforces access privilege checks, detects and deals with self-modifying code, and performs byte swapping between guest and host as necessary.

Finally, we will identify those aspects of current micro-architectures which impede efficient virtual machine implementation and propose simple x86 ISA extensions which could provide lightweight hardware-assisted acceleration to an interpreted virtual machine.

In the long term such ISA extensions, combined with a BIOS-resident virtual machine, could allow future x86 microprocessor implementations to completely remove not only hardware related to 16-bit legacy support, but also hardware related to segmentation, paging, heavyweight virtualization, and rarely used x86 instructions. This would reduce die sizes and simplify the hardware verification. Much as was the approach of Transmeta in the late 1990's, the purpose of the microprocessor becomes that of being an efficient host for virtual machines[19].

1.2 Overview of This Paper

The premise of this paper is that an efficient and portable virtual machine can be developed using a high-level language that uses purely interpreted techniques. To show this we looked at two very different real-world virtual machines - Gemulator and Bochs - which were independently developed since the 1990s to emulate 68040 and x86 guest systems respectively. Since these emulators are both interpretation based and are still actively maintained by each of the authors of this paper, they served as excellent test cases to see if similar optimization and design techniques could be applied to both.

Section 2 discusses the design of Gemulator and looks at several past and present techniques used to implement its 68040 ISA interpreter. Gemulator was originally developed almost entirely in x86 assembly code that was very x86 MS-DOS and x86 Windows specific and not portable even to a 64-bit Windows platform.

Section 3 discusses the design of Bochs, and some of the many optimization and portability techniques used for its Bochs x86 interpreter. The work on Bochs focused on improving the performance of its existing portable C++ code as well as eliminating pieces of non-portable assembly code in an efficient manner.

Based on the common techniques that resulted from the work on both Gemulator and Bochs and the common problems encountered in both – guest-to-host address translation and guest flags emulation - Section 4 proposes simple ISA hardware extensions which we feel could aid the performance of any arbitrary interpreter based virtual machine.

2.0 Gemulator

Gemulator is an MS-DOS and Windows hosted emulator which runs Atari 800, Atari ST, and classic 680x0 Apple Macintosh software. The beginnings of Gemulator date back to 1987 as a tutorial on assembly language and emulation in the Atari magazine ST-LOG[20]. In 1991, emulation of the Motorola MC68000 microprocessor and the Atari ST chipset was added, and in 1997 a native 32-bit Windows version of Gemulator was developed which eventually added support for a 68040 guest running Mac OS 8.1 in a release called "SoftMac". Each release of Gemulator was based around a 68000/68040 bytecode interpreter written in 80386 assembly language and which was laboriously retuned every few years for 486, Pentium Pro "P6", and Pentium 4 "Netburst" cores.

In the summer of 2007 work began to start converting the Gemulator code to C for eventually hosting on both 32-bit and 64-bit host machines. Because of the endian difference between 68000/68040 and 80386 architectures, it was a goal to keep the new C code as byte agnostic as possible. And of course, the conversion from 80386 assembly language to C should incur as little performance penalty as possible.

The work so far on Gemulator 9.0 has focused on converting the guest data memory access logic to portable code, and examining the pros and cons of various guest-to-host address translation techniques which have been used over the years and selecting the one that best balances efficiency and portability.

2.1 Byte Swapping on the Intel 80386

A little-endian architecture such as Intel 80386 stores the least significant byte first, while a big-endian architecture such as Motorola 68000 stores the most significant byte first. Byte values, such as ASCII text characters, are stored identically, so an array of bytes, or a text string is stored in memory the same regardless of endianness.

Since the 80386 did not support a BSWAP instruction, the technique in Gemulator was to treat all of guest memory address space - all 16 megabytes of it for 68000 guests - as one giant integer stored in reverse order. Whereas a 68000 stores memory addresses G, G+1, G+2, etc. in ascending order, Gemulator maps guest address G to host address H, G+1 maps to H-1, G+2 maps to H-2, etc. G + H is constant, such that G = K – H, and H = K – G.

Multi-byte data types, such as a 32-bit integer can be accessed directly from guest space by applying a small adjustment to account for the size of the data access. For example, to read a 32-bit integer from guest address 100, calculate the guest address corresponding to the last byte of that access before negating, so in this case guest address 100 + sizeof(int) – 1 = guest address 103. The memory read *(int *)&K[-103] correctly returns the guest data.

The early-1990's releases of Gemulator were hosted on MS-DOS and on Windows 3.1, and thus did not have the benefit of Win32 or Linux style memory protection and mapping APIs. As such these interpreters also bounds checked each negated guest offset such that only guest RAM accesses (usually guest addresses 0 through 4 megabytes) used the direct access, while all other accesses, including to guest ROM space, video frame buffer RAM, and memory mapped I/O, took a slower path through a hardware emulation handler.

Instrumentation showed that only about 1 in 1000 memory accesses on average failed the bounds check, allowing roughly 99.9% of guest memory accesses to use the "adjust-and-negate" bounds checking scheme, and this allowed a 33 MHz 80386 based PC to efficiently emulate close to the full speed of the original 8 MHz 68000 Atari ST and Apple Macintosh computers.

2.2 Page Table using XOR Translation

A different technique must be used when mapping the entire 32-bit 4-gigabyte address space of the 68040 to the smaller than 2-gigabyte address of a Windows application. The answer relies on the observation that subtracting an integer value from 0xFFFFFFFF gives the same result as XOR-ing that same value to 0xFFFFFFFF. For example:

```
0xFFFFFFFF - 0x12345678 = 0xEDCBA987
0xFFFFFFFF XOR 0x12345678 = 0xEDCBA987
```

This observation allows for portions of the guest address space to be mapped to the host in power-of-2 sized power-of-2-aligned blocks. The XOR operation, instead of a subtraction, is used to perform the byte-swapping address translation. Every byte within each such block will have a unique XOR constant such that the H = K –G property is maintained.

For example, mapping 256 megabytes of Macintosh RAM from guest address 0x00000000..0x0FFFFFFF to a 256-megabyte aligned host block allocated at address 0x30000000 requires that the XOR constant be 0x3FFFFFFF, which is derived taking either the XOR of the address of that host block and the last byte of the guest range (0x30000000 XOR 0x0FFFFFFF) or the first address of the guest range and the last byte of the allocated host range (0x00000000 XOR 0x3FFFFFFF). Guest address 0x00012345 thus maps to host address 0x3FFFFFFF – 0x00012345 = 0x3FFEDCBA for this particular allocation.

To reduce fragmentation, Gemulator starts with the largest guest block to map and then allocates progressively smaller blocks, the order usually being guest RAM, then guest ROM, then guest video frame buffer. The algorithm used is as follows:

```
for each of the RAM ROM and video guest address ranges
 {
 calculate the size of that memory range rounded up to next power of 2
 for each megabyte-sized range of Windows host address space
 {
  calculate the XOR constant for the first and last byte of the block
  if the two XOR constants are identical
  {
   call VirtualAlloc() to request that specific host address range
   if successful record the address and break out of loop;
  }
 }
}
```

Listing 2.1: Pseudo code of Gemulator's allocator

This algorithm scans for host free blocks a megabyte at a time because it turns out the power-of-2 alignment need not match the block size. This helps to find large unused blocks of host address space when memory fragmentation is present.

For example, a gigabyte of Macintosh address space 0x00000000 through 0x3FFFFFFF can map to Windows host space 0x20000000 though 0x5FFFFFFF because there exists a consistent XOR constant:

```
0x5FFFFFFF XOR 0x00000000 = 0x5FFFFFFF
0x20000000 XOR 0x3FFFFFFF = 0x5FFFFFFF
```

This XOR-based translation is endian agnostic. When host and guest are of the same endianness, the XOR constant will have zeroes in its lower bits. When the host and guest are of opposite endianness, as is the case with 68040 and x86, the XOR constant has its lower bits set. How many bits are set or cleared depends on the page size granularity of mapping.

A granularity of 64K was decided upon based on the fact that the smallest Apple Macintosh ROM is 64K in size. Mapping 4 gigabytes of guest address space at 64K granularity generates 4GB/64K = 65536 different guest address ranges. A 65536-entry software page table is used, and the original address negation and bounds check from before is now a traditional table lookup which uses XOR to convert the input guest address in EBP to a host address in EDI::

```
; Convert 68000 address to host address in EDI
; Sign flag is SET if EA did not map.
    mov   edi,ebp
    shr   ebp,16
    xor   edi,dword ptr[pagetbl+ebp*4]
```

Listing 2.2: Guest-to-host mapping using flat page table

For unmappable guest addresses ranges such as memory mapped I/O, the XOR constant for that range is selected such that the resulting value in EDI maps to above 0x80000000. This can now be checked with an explicit JS (jump signed) conditional branch to the hardware emulation handler, or by the resulting access violation page fault which invokes the same hardware emulation handler via a trap-and-emulate.

This design suffers from a non-portable flaw – it assumes that 32-bit user mode addresses on Windows do not exceed address 0x80000000, an assumption that is outright invalid on 64-bit Windows and other operating systems.

The code also does not check for misaligned accesses or accesses across a page boundary, which prevents further sub-allocation of the guest address space into smaller regions. Reducing the granularity of the mapping also inversely grows the size of the lookup table. Using 4K mapping granularity for example requires 4GB/4K = 1048576 entries consuming 4 megabytes of host memory.

2.3 Fine-Grained Guest TLB

The approach now used by Gemulator 9 combines the two methods – range check using a lookup table of only 2048 entries - effectively implementing a software-based TLB for guest addresses. Each table entry still spans a specific guest address range but now holds two values: the XOR translation value for that range, and the corresponding base guest address of the mapped range. This code sequence is used to translate for a guest write access of a 16-bit integer using 128-byte granularity:

```
mov    edx,ebp
shr    edx,bitsSpan  ; bitsSpan = 7
and    edx,dwIndexMask ; dwIndexMask = 2047
mov    ecx,ebp       ; guest address
add    ecx,cb-1      ; high address of access
; XOR to compare with the cached address
xor     ecx,dword ptr [memtlbRW+edx*8]
; prefetch translation XOR value
mov   eax,dword ptr [memtlbRW+edx*8+4]
test    ecx,dwBaseMask
jne     emulate      ; if no match, go emulate
xor     eax,ebp      ; otherwise translate
```

Listing 2.3: Guest-to-host mapping using a software TLB

The first XOR operation takes the highest address of the access and compares it to the base of the address range translated by that entry. When the two numbers are in range, all but a few lower bits of the result will be zero. The TEST instruction is used to mask out the irrelevant low bits and check that the high bits did match. If the result is non-zero, indicating a TLB miss or a cross-block access, the JNE branch is taken to the slow emulation path. The second XOR performs the translation as in the page table scheme.

Various block translation granularities and TLB sizes were tested for performance and hit rates. The traditional 4K granularity was tried and then reduced by factors of two. Instrumentation counts showed that hit rates remained good for smaller granularities even of 128 bytes, 64 bytes, and 32 bytes, giving the fine grained TLB mechanism between 96% and 99% data access hit rate for a mixed set of Atari ST and Mac OS 8.1 test runs.

The key to hit rate is not in the size of the translation granularity, since data access patterns tend to be scattered, but rather the key is to have enough entries in the TLB table to handle the scattering of guest memory accesses. A value of at least 512 entries was found to provide acceptable performance, with 2048 entries giving the best hit rates. Beyond 2048 entries, performance improvement for the Mac and Atari ST workloads was negligible and merely consumed extra host memory.

It was found that certain large memory copy benchmarks did poorly with this scheme. This was due to two factors:

- 64K aliasing of source and destination addresses, and,
- Frequent TLB misses for sequential accesses in guest memory space.

The 64K aliasing problem occurs because a direct-mapped table of 2048 entries spanning 32-byte guest address ranges wraps around every 64K of guest address space. The 32-byte granularity also means that for sequential accesses, every 8th 32-bit access will "miss". For these two reasons, a block granularity of 128 bytes is used so as to increase the aliasing interval to 256K.

Also to better address aliasing, three translation tables are used – a TLB for write and read-modify-write guess accesses, a TLB for read-only guest accesses, and a TLB for code translation and dispatch. This allows guest code to execute a block memory copy without suffer from aliasing between the program counter, the source of the copy, or the destination of the copy.

The code TLB is kept at 32-byte granularity and contains extra entries holding a dispatch address for each of the 32 addresses in the range. When a code TLB entry is populated, the 32 dispatch addresses are initialized to point to a stub function, similar to how jitting schemes work. When an address is dispatched, the stub function calculates the true address of the handlers and updates the entry in the table.

To handle self-modifying code, when a code TLB entry is first populated, the corresponding entry (if present) is flushed from the write TLB. Similarly, when a write TLB entry misses, it flushes six code TLB entries – the four entries corresponding to the 128-byte data range covered by the write TLB entry, and one code "guard block" on either side are flushed. This serves two purposes:

- To ensure that an address range of guest memory is never cached as both writable data and as executable code, such that writes to code space are always noted by the virtual machine, and,
- To permit contiguous code TLB blocks to flow into each other, eliminating the need for an address check on each guest instruction dispatch.

Keeping code block granularity small along with relatively small data granularity means that code execution and data writes can be made to the same 4K page of guest memory with less chance of false detection of self-modifying code and eviction of TLB entries as can happen when using the standard 4K page granularity. Legacy 68000 code is known to place writeable data near code, as well as using back-patching and other self-modification to achieve better performance.

This three-TLB approach gives the best bounded behavior of any of the previous Gemulator implementations. Unlike the original MS-DOS implementation, guest ROM and video accesses are not penalized for failing a bounds check. Unlike the previous Windows implementations, all guest memory accesses are verified in software and require no "trap-and-emulate" fault handler.

The total host-side memory footprint of the three translation tables is:

- 2048 * 8 bytes = 16K for write TLB
- 2048 * 8 bytes = 16K for read TLB
- 2048 * 8 bytes = 16K for code TLB
- 65536*4 = 256K for code dispatch entries

This results in an overall memory footprint of just over 300 kilobytes for all of the data structures relating to address translation and cached instruction dispatch.

For portability to non-x86 host platforms, the 10-instruction assembly language sequence was converted to this inlined C function to perform the TLB lookup, while the actual memory dereference occurs at the call site within each guest memory write accessor:

```
void * pvGuestLogicalWrite(
  ULONG addr, unsigned cb)
{
  ULONG ispan;
  ispan = (((addr + cb - 1) >> bitsSpan)
         & dwIndexMask);

  void *p;
  p = ((addr ^ vpregs->memtlbRW[ispan*2+1])
         - (cb - 1));

  if (0 == (dwBaseMask &
         (addr ^ (vpregs->memtlbRW[ispan*2]))))
  {
    return p;
  }
  return NULL;
}
```

Listing 2.4: Software TLB lookup in C/C++

This code compiles into almost as efficient a code sequence as the original assembly code, except for a spill of ECX which the Microsoft Visual Studio compiler generates, mandated by the __fastcall calling convention of preserving the ECX register.

On a 2.66 GHz Intel Core 2 host computer, the 68000 and 68040 instruction dispatch rate is about 120 to 170 million guest instructions per second, or approximately one guest instruction dispatch every 15 to 22 host clock cycles, depending on the Atari ST or Mac OS workload.

The aggregate hit rate for the read TLB and write TLB is typically over 95% while the hit rate for the code TLB's dispatch entries exceeds 98%.

For example, a workload consisting of booting Mac OS 8.1, launching the Speedometer 3.23 benchmarking utility, and running a short suite of arithmetic, floating point, and graphics benchmarks dispatches a total of 3.216 billion guest instructions of which 43 million were not already cached, a 98.6% hit rate on instruction dispatch.

That same scenario generates 3.014 billion guest data read and write accesses of which 132 million failed to translate, for a 95.6% hit rate. The misses include accesses to memory mapped I/O that never maps directly to the host.

This latest implementation of Gemulator now has very favorable and portable characteristics:

- Runs on the minimal "least common denominator" IA32 instruction set of 80386 which performs efficient byte swapping without requiring a host to support a BSWAP instruction,
- Short and predictably low-latency code paths,
- No exceptions are thrown as all guest memory accesses are range checked,
- Less than 1 megabyte of scratch working memory.

These characteristics are applicable not just to running 68040 guest code, but for more modern byte-swapping scenarios such as running PowerPC guest code on x86, or running x86 guest code on PowerPC.

The high hit rate of guest instruction dispatch and guest memory translation means that the majority of 68000 and 68040 instructions are simulated using short code paths involving translation functions with excellent branch prediction characteristics. As is described in the following section, improving the branch prediction rates on the host is critical.

3.0 Bochs

Bochs is a highly portable open source IA-32 PC emulator written purely in C++ that runs on most popular platforms. It includes emulation of the CPU engine, common I/O devices, and custom BIOS. Bochs can be compiled to emulate any modern x86 CPU architecture, including most recent Core 2 Duo instruction extensions. Bochs is capable of running most operating systems including MS-DOS, Linux, Windows 9X/NT/2000/XP and Windows Vista. Bochs was written by Kevin Lawton and currently maintained by the Bochs open source project[21]. Unlike most of its competitors like QEMU, Xen or VMware, Bochs doesn't feature a dynamic translation or virtualization technologies but uses pure interpretation to emulate the CPU.

During our work we took the official Bochs 2.3.5 release sources tree and made it run over than three times faster using only **host independent and portable** optimization techniques without affecting emulation accuracy.

3.1 Quick introduction to Bochs internals

Our optimizations concentrated in the CPU module of the Bochs full system emulator and mainly dealt with the primary emulation loop optimization, called the CPU loop. According to Bochs 2.3.5 profiling data, the CPU loop took around 50% of total emulation time. It turned out that while every instruction emulated relatively efficiently, Bochs spent a lot of effort for routine operations like fetching, decoding and dispatching instructions.

The Bochs 2.3.5 CPU main emulation loop looks very similar to that of a physical non-pipelined micro-coded CPU like Motorola 68000 or Intel 8086 [22]. Every emulated instruction passes through six stages during the emulation:

1. At prefetch stage, the instruction pointer is checked for fetch permission according to current privilege level and code segment limits, and host instruction fetch pointer is calculated. The prefetch code also updates memory page timestamps used for self modifying code detection by memory accesses.

2. After prefetch stage is complete the specific instruction could be looked up in Bochs' internal cache or fetched from the memory and decoded.

3. When the emulator has obtained an instruction, it can be instrumented on-the-fly by internal or external debugger or instrumentation tools.

4. In case an instruction contains memory references, the effective address of an instruction is calculated using an indirect call to the resolve memory reference function.

5. The instruction is executed using an indirect call dispatch to the instruction's execution method, stored together with instruction decode information.

6. At instruction commit the internal CPU EIP state is updated. The previous state is used to return to the last executed instruction in case of an x86 architectural fault occurring during the current instruction's execution.

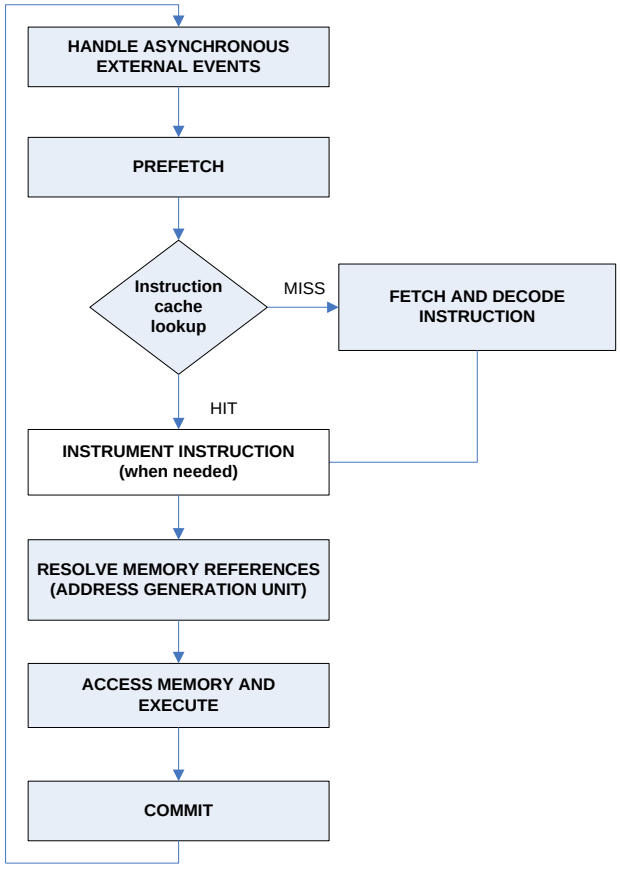

Figure 3.1: Bochs CPU loop state diagram

As emulation speed is bounded by the latency of these six stages, shortening any and each of them immediately affects emulation performance.

3.2 Taking hardware ideas into emulation – using decoded instructions trace cache

Variable length x86 instructions, many different decoding templates, three possible operand and address sizes in x86-64 mode make instruction fetch-decode operations one of the heaviest parts of x86 emulation. The Bochs community realized this and introduced the decoded instruction cache to Bochs 2.0 at the end of 2002. The cache almost doubled the emulator performance.

The Pentium 4 processor stores decoded and executed instruction blocks into a trace cache[23] containing up to 12K of micro-ops. The next time when the execution engine needs the same block of instructions, it can be fetched from the trace cache instead of being loaded from the memory and decoded again. The Pentium 4 trace cache operates in two modes. In the "execute mode" the trace is feeding micro-ops stored in the trace to the execution engine. This is the mode that the trace cache normally runs in. Once a trace cache miss occurs the trace cache switches into the "build mode". In this mode the fetch-decode engine fetches and decodes x86 instructions from memory and builds a micro-ops trace which is stored in the cache.

The trace cache introduced into Bochs 2.3.6 is very similar to the Pentium 4 hardware implementation. Bochs maintains a decoded instruction trace cache organized as a 32768-entry direct mapped array with each entry holding a trace of up to 16 instructions. The tracing engine stops when it detects an x86 instruction able to affect control flow of the guest code, such as a branch taken, an undefined opcode, a page table invalidation or a write to control registers. Speculative tracing through potentially non-taken conditional branches is allowed. An early-out technique is used to stop trace execution when a taken branch occurs.

When the Bochs CPU loop is executing instructions from the trace cache, all front-end overhead of prefetch and decode is eliminated. Our experiments with a Windows XP guest show most traces to be less than 7 guest instructions in length and almost none longer than 16.

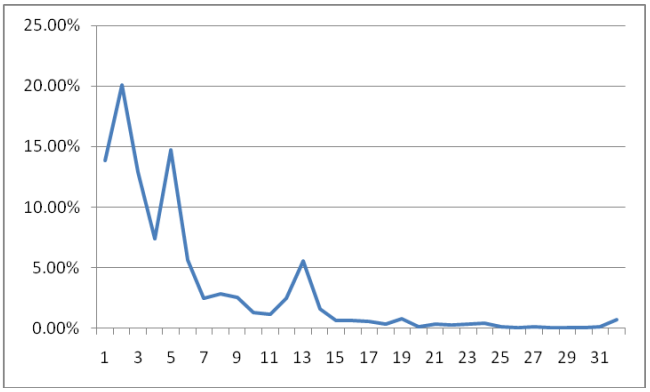

Figure 3.2: Trace length distribution for Windows XP boot

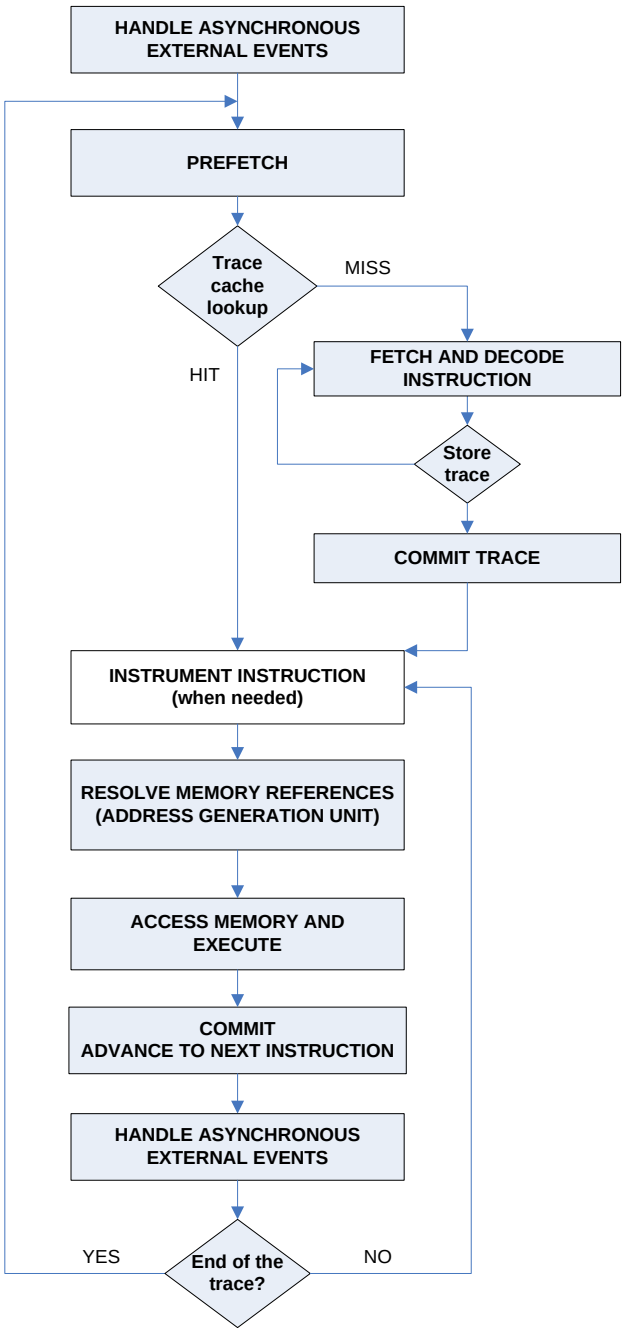

Figure 3.3: Bochs CPU loop state diagram with trace cache

In addition to the over 20% speedup in Bochs emulation, the trace cache has great potential for the future. We are working on the following techniques which will help to double emulation speed again in a short term:

- Complicated x86 instructions could be decoded to several simpler micro-ops in the trace cache and handled more efficiently by the emulator.

- Compiler optimization techniques can be applied to the hot traces in the trace cache. Register move elimination, no-op elimination, combining memory accesses, replacing instruction dispatch handlers, and redundant x86 flags update elimination are only a few techniques that can be applied to make hot traces run faster.

The software trace cache's primary problem is direct mapped associativity. This can lead to frequent trace cache collisions due to aliasing of addresses at 32K and larger power-of-two intervals. Hardware caches use multi-way associativity to avoid aliasing issues. A software implementation of a two- or four-way associative cache and LRU management can potentially increase branch misprediction during lookup, reducing cache gain to a minimum.

What Bochs does instead today is use a 65536-entry table. A hash function calculates the trace cache index of guest address X using this formula:

- *index := (X + (X<<2) + (X>>6)) mod 65536*

We found that the best trace cache hashing function requires both a left shift and a right shift, providing the non-linearity so that two blocks of code separated by approximately a power-of-two interval will likely not conflict.

3.3 Host branch misprediction as biggest cause of slow emulation performance

Every pipelined processor features branch prediction logic used to predict whether a conditional branch in the instruction flow of a program is likely to be taken or not. Branch predictors are crucial in today's modern, superscalar processors for achieving high performance.

Modern CPU architectures implement a set of sophisticated branch predictions algorithms in order to achieve highest prediction rate, combining both static and dynamic prediction methods. When a branch instruction is executed, the branch history is stored inside the processor. Once branch history is available, the processor can predict branch outcome – whether the branch should be taken and the branch target.

The processor uses branch history tables and branch target buffers to predict the direction and target of branches based on the branch instruction's address.

The Core micro-architecture branch predictor makes the following types of predictions:

- Direct calls and jumps. The jump targets are read from the branch target array regardless of the taken/not taken prediction.

- Indirect calls and jumps. May either be predicted as having a monotonic target or as having targets that vary in accordance with recent program behavior.

- Conditional branches. Predicts branch target and whether or not the branch will be taken.

- Returns from procedure calls. The branch predictor contains a 16-entry return stack buffer. It enables accurate prediction for RET instructions.

Let's look closer into at the Bochs 2.3.5 main CPU emulation loop. As can be seen the CPU loop alone already gives enough work to the branch predictor due to two indirect calls right in the heart of the emulation loop, one for calculating the effective address of memory accessing instructions, and another for dispatching to the instruction execution method. In addition to these indirect calls many instruction methods contain conditional branches in order to distinguish different operand sizes or register/memory instruction format.

A typical Bochs instruction handler method:

```
void BX_CPU_C::SUB_EdGd(bxInstruction_c *i)
{
  Bit32u op2_32, op1_32, diff_32;

  op2_32 = BX_READ_32BIT_REG(i->nnn());

  if (i->modC0()) {    // reg/reg format
    op1_32 = BX_READ_32BIT_REG(i->rm());
    diff_32 = op1_32 - op2_32;
    BX_WRITE_32BIT_REGZ(i->rm(), diff_32);
  }
  else {               // mem/reg format
    read_RMW_virtual_dword(i->seg(),
        RMAddr(i), &op1_32);
    diff_32 = op1_32 - op2_32;
    Write_RMW_virtual_dword(diff_32);
  }
  SET_LAZY_FLAGS_SUB32(op1_32, op2_32,
        diff_32);
}
```

Listing 3.1: A typical Bochs instruction handler

Taking into account 20 cycles Core 2 Duo processor branch misprediction penalty [24] we might see that a cost of every branch misprediction during instruction emulation became huge. A typical instruction handler is short and simple enough such that even a single extra misprediction during

every instruction execution could slow the emulation down by half.

3.3.1 Splitting common opcode handlers into many to reduce branch misprediction

All Bochs decoding tables were expanded to distinguish between register and memory instruction formats. At decode time, it is possible to determine whether an instruction is going to access memory during the execution stage. All common instruction execution methods were split into methods for register-register and register-memory cases separately, eliminating a conditional check and associated potential branch misprediction during instruction execution. The change alone brought a ~15% speedup.

3.3.2. Resolve memory references with no branch mispredictions

The x86 architecture has one of the most complicated instruction formats of any processor. Not only can almost every instruction perform an operation between register and memory but the address of the memory access might be computed in several ways.

In the most general case the effective address computation in the x86 architecture can be expressed by the formula:

- *Effective Address := (Base + Index * Scale + Displacement) mod* $2^{AddrSize}$

The arguments of effective address computation (*Base*, *Index*, *Scale* and *Displacement*) can be encoded in many different ways using ModRM and S-I-B instruction bytes. Every different encoding might introduce a different effective address computation method.

For example, when the *Index* field is not encoded in the instruction, it could be interpreted as *Index* being equal to zero in the general effective address (EA) calculation, or as simpler formula which would look like:

- *Effective Address := (Base + Displacement) mod* $2^{AddrSize}$

Straight forward interpretation of x86 instructions decoding forms already results in 6 different EA calculation methods only for 32-bit address size:

- *Effective Address := Base*
- *Effective Address := Displacement*
- *Effective Address := (Base + Displacement) mod* 2^{32}
- *Effective Address := (Base + Index * Scale) mod* 2^{32}
- *Effective Address := (Index * Scale + Displacement) mod* 2^{32}
- *Effective Address := (Base + Index * Scale + Displacement) mod* 2^{32}

The Bochs 2.3.5 code went even one step ahead and split every one of the above methods to eight methods according to which one of the eight x86 registers (EAX...EDI) used as a Base in the instruction. The heart of the CPU emulation loop dispatched to one of thirty EA calculation methods for every emulated x86 instruction accessing memory. This single point of indirection to so many possible targets results in almost a 100% chance for branch misprediction.

It is possible to improve branch prediction of indirect branches in two ways – reducing the number of possible indirect branch targets, and, replicating the indirect branch point around the code. Replicating indirect branches will allocate a separate branch target buffer (BTB) entry for each replica of the branch. We choose to implement both techniques.

As a first step the Bochs instruction decoder was modified to generate references to the most general EA calculation methods. In 32-bit mode only two EA calculation formulas are left:

- *Effective Address := (Base + Displacement) mod* 2^{32}
- *Effective Address := (Base + Index * Scale + Displacement) mod* 2^{32}

where *Base* or *Index* fields might be initialized to be a special NULL register which always contains a value of zero during all the emulation time.

The second step moved the EA calculation method call in the main CPU loop and replicated it inside the execution methods. With this approach every instruction now has its own EA calculation point and is seen as separate indirect call entity for host branch prediction hardware. When emulating a guest basic block loop, every instruction in the basic block might have its own EA form and could still be perfectly predicted.

Implementation of these two steps brought ~40% emulation speed total due elimination of branch misprediction penalties on memory accessing instructions.

3.4. Switching from the PUSHF/POP to improved lazy flags approach

One of the few places where Bochs used inline assembly code was to accelerate the simulation of x86 EFLAGS condition bits. This was a non-portable optimization, and as it turned out, no faster than the portable alternative.

Bochs 2.3.7 uses an improved “lazy flags” scheme whereby the guest EFLAGS bits are evaluated only as needed. To

facilitate this, handlers of arithmetic instructions execute macros which store away the sign-extended result of the operation, and as needed, one or both of the operands going into the arithmetic operation.

Our measurements had shown that the greatest number of lazy flags evaluations is for the Zero Flag (ZF), mostly for Jump Equal and Jump Not Equal conditional branches. The lazy flags mechanism is faster because ZF can be derived entirely from looking at the cached arithmetic result. If the saved result is zero, ZF is set, and vice versa. Checking a value for zero is much faster than calling a piece of assembly code to execute a PUSHF instruction on the host on every emulated arithmetic instruction in order to update the emulated EFLAGS register.

Similarly by checking only the top bit of the saved result, the Sign Flag (SF) can be evaluated much more quickly than the PUSHF way. The Parity Flag (PF) is similarly arrived by looking at the lowest 8 bits of the cached result and using a 256-byte lookup table to read the parity for those 8 bits.

The Carry Flag (CF) is derived by checking the absolute magnitude of the first operand and the cached result. For example, if an unsigned addition operation caused the result to be smaller than the first operand, an arithmetic unsigned overflow (i.e. a Carry) occurred.

The more problematic flags to evaluate are Overflow Flag (OF) and Adjust Flag (AF). Observe that for any two integers A and B that (A + B) equals (A XOR B) when no bit positions receive a carry in. The XOR (Exclusive-Or) operation has the property that bits are set to 1 in the result only if the corresponding bits in the input values are different. Therefore when no carries are generated, (A + B) XOR (A XOR B) equals zero. If any bit position b is not zero, that indicates a carry-in from the next lower bit position b-1, thus causing bit b to toggle.

The Adjust Flag indicates a carry-out from the 4th least significant bit of the result (bit mask 0x08). A carry out from the 4th bit is really the carry-in input to the 5th bit (bit mask 0x10). Therefore to derive the Adjust Flag, perform an Exclusive-OR of the resulting sum with the two input operands, and check bit mask 0x10, as follows:

```
AF = ((op1 ^ op2) ^ result) & 0x10;
```

Overflow uses this trick to check for changes in the high bit of the result, which indicates the sign. A signed overflow occurs when both input operands are of the same sign and yet the result is of the opposite sign. In other words, given input A and B with result D, if (A XOR B) is positive, then both (A XOR D) and (B XOR D) need to be positive, otherwise an overflow has occurred. Written in C:

```
OF = ((op1 ^ op2) & (op1 ^ result)) < 0;
```

Further details of this XOR math are described online[25].

3.5. Benchmarking Bochs

The very stunning demonstration of how the design techniques we just described were effective shows up in the time it takes Bochs to boot a Windows XP guest on various host computers and how that time has dropped significantly from Bochs 2.3.5 to Bochs 2.3.6 to Bochs 2.3.7. The table below shows the elapsed time in seconds from the moment when Bochs starts the Windows XP boot process to the moment when Windows XP has rendered its desktop icons, Start menu, and task bar. Each Bochs version is compiled as a 32-bit Windows application and configured to simulate a Pentium 4 guest CPU.

	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

Table 3.1: Windows XP boot time on different hosts

Booting Windows XP is not a pure test of guest CPU throughput due to tens of megabytes of disk I/O and the simulation of probing for and initialization of hardware devices. Using a Visual C++ compiled CPU-bound test program [26] one can get an idea of the peak throughput of the virtual machine's CPU loop.

```
#include "windows.h"
#include "stdio.h"

static int foo(int i)
{
    return(i+1);
}

int main(void)
{
    long tc = GetTickCount();
    int i;
    int t = 0;

    for(i = 0; i < 100000000; i++)
        t += foo(i);

    tc = GetTickCount() - tc;
    printf("tc=%ld, t=%d\n", tc, t, t);

    return t;
}
```

Listing 3.2: Win32 instruction mix test program

The test is compiled as two test executables, T1FAST and T1SLOW, which are the optimized and non-optimized

compiles of this simple test code that incorporates arithmetic operations, function calls, and a loop. The difference between the two builds is that the optimized version (T1FAST) makes more use of x86 guest registers, while the unoptimized version (T1SLOW) performs more guest memory accesses.

On a modern Intel Core 2 Duo based system, this test code achieves similar performance on Bochs as it does on the dynamic recompilation based QEMU virtual machine:

Execution Mode	T1FAST.EXE time	T1SLOW.EXE time
Native	0.26	0.26
QEMU 0.9.0	10.5	12
Bochs 2.3.5	25	31
Bochs 2.3.7	8	10

Table 3.2: Execution time in seconds of Win32 test program

Instruction count instrumentation shows that T1FAST averages about 102 million guest instructions per second (MIPS). T1SLOW averages about 87 MIPS due to a greater mix of guest instructions that perform a guest-to-host memory translation using the software TLB mechanism similar to the one used in Gemulator.

This simple benchmark indicates that the average guest instruction requires approximately 26 to 30 host clock cycles. We tested some even finer grained micro-benchmarks written in assembly code, specifically breaking up the test code into:

- Simple register-register operations such as MOV and MOVSX which do not consume or update condition flags,
- Register-register arithmetic operations such as ADD, INC, SBB, and shifts which do consume and update condition flags,
- Simple floating point operations such as FMUL,
- Memory load, store, and read-modify-write operations,
- Indirect function calls using the guest instruction CALL EAX,
- The non-faulting Windows system call VirtualProtect(),
- Inducing page faults to measure round trip time of a __try/__except structured exception handler

The micro-benchmarks were performed on Bochs 2.3.5, the current Bochs 2.3.7, and on QEMU 0.9.0 on a 2.66 GHz Core 2 Duo test system running Windows Vista SP1 as host and Windows XP SP2 as guest operating system.

	Bochs 2.3.5	Bochs 2.3.7	QEMU 0.9.0
Register move (MOV, MOVSX)	43	15	6
Register arithmetic (ADD, SBB)	64	25	6
Floating point multiply	1054	351	27
Memory store of constant	99	59	5
Pairs of memory load and store operations	193	98	14
Non-atomic read-modify-write	112	75	10
Indirect call through guest EAX register	190	109	197
VirtualProtect system call	126952	63476	22593
Page fault and handler	888666	380857	156823
Best case peak guest execution rate in MIPS	62	177	444

Table 3.3: Approximate host cycle costs of guest operations

This data is representative of over 100 micro-benchmarks, and revealed that timings for similar guest instructions tended to cluster around the same number of clock cycles. For example, the timings for register-to-register move operations, whether byte moves, full register moves, or sign extended moves, were virtually identical on all four test systems. Changing the move to an arithmetic operation and thus introducing the overhead of updating guest flags similarly affects the clock cycle costs, and is mostly independent of the actual arithmetic operation (AND, ADD, XOR, SUB, etc) being performed. This is due to the relatively fixed and predictable cost of the Bochs lazy flags implementation.

Read-modify-write operations are implemented more efficiently than separate load and store operations due to the fact that a read-modify-write access requires one single guest-to-host address translation instead of two. Other micro-benchmarks not listed here show that unlike past Intel architectures, the Core 2 architecture also natively performs a read-modify-write more efficiently than a separate load and store sequence, thus allowing QEMU to benefit from this in its dynamically recompiled code. However, dynamic translation of code and the associated code cache management do show up as a higher cost for indirect function calls.

4.0 Proposed x86 ISA Extensions – Lightweight Alternatives to Hardware Virtualization

The fine-grained software TLB translation code listed in section 2.3 is nothing more than a hash table lookup which performs a "fuzzy compare" for the purposes of matching a range of addresses, and returns a value which is used to translate the matched address. This is exactly what TLB hardware in CPUs does today.

It would be of benefit to binary translation engines if the TLB functionality was programmatically exposed for general purpose use, using a pair of instructions to add a value to the hash table, and an instruction to look up a value in the hash table. This entire code sequence:

```
mov   edx,ebp
shr   edx,bitsSpan
and   edx,dwIndexMask
mov   ecx,ebp
add   ecx,cb-1
xor   ecx,dword ptr [memtlbRW+edx*8]
mov   eax,dword ptr [memtlbRW+edx*8+4]
test  ecx,dwBaseMask
jne   emulate
```

could be reduced to two instructions, based on the new "Hash LookUp" instruction HASHLU which takes a destination register (EAX), an r/m32/64 addressing mode which resolves to an address range to look up, and a "flags" immediate which determines the matching criteria.

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

Flags could be an imm32 value similar to the mask used in the TEST instruction of the original sequence, or an imm8 value in a more compact representation (4 bits to specify alignment requirements in lowest bits, and 4 bits to specify block size in bits). The data access size is also keyed as part of the lookup, as it represents the span of the address being looked up.

This instruction would potentially reduce the execution time of the TLB lookup and translation from about 8 clock cycles to potentially one cycle in the branch predicted case.

To add a range to the hash table, use the new "Hash Add" instruction HASHADD, which takes an effective address to use as the fuzzy hash key, the second parameter specifies the value to hash, and flags again is either an imm32 or imm8 value which specifies size of the range being hashed:

```
hashadd dword ptr [ebp],eax,flags
jne error
```

The instruction sets Zero flag on success, or clears it when there is conflict with another range already hashed or due to a capacity limitation such that the value could not be added.

The hardware would internally implement a TLB structure of implementation specific size and set associativity, and the hash table may or may not be local to the core or shared between cores. Internally the entries would be keyed with additional bits such as core ID or CR3 value or such and could possibly coalesce contiguous ranges into a single entry.

This programmable TLB would have nothing to do functionally with the MMU's TLB. This one exists purely for user mode application use to accelerate table lookups and range checks in software. As with any hardware cache, it is subject to be flushed arbitrarily and return false misses, but never false positives.

4.1 Instructions to access EFLAGS efficiently

LAHF has the serious restriction of operating on a partial high register (AH) which is not optimal on some architectures (writing to it can cause a partial register stall as on Pentium III, and accessing it may be slower than AL as is the case on Pentium 4 and Athlon).

LAHF also only returns 5 of the 6 arithmetic flags, and does not return Overflow flag, or the Direction flag.

PUSHF is too heavyweight, necessitating both a stack memory write and stack memory read.

A new instruction is needed, SXF reg32/reg64/r/m32/64 (Store Extended Flags), which loads a full register with a zero extended representation of the 6 arithmetic flags plus the Direction flag. The bits are packed down to lowest 7 bits for easy masking with imm8 constants. For future expansion the data value is 32 bits or 64-bits, not just 8 bits.

SXF can find use in virtual machines which use binary translation and must save the guest state before calling glue code, and in functions which must preserve existing EFLAGS state. A complementary instruction LXF (Load Extended Flags) would restore the state.

A SXF/LXF sequence should have much lower latency than PUSHF/POPF, since it would not cause partial register stalls nor cause the serializing behavior of a full EFLAGS update as happens with POPF.

5.0 Conclusions and Further Research

Using two completely different virtual machines we have demonstrated techniques that allow a mainstream Core 2 hosted virtual machine to reach purely interpreted execution rates of over 100 MIPS, peaking at about 180 MIPS today.

Our results show that the key to interpreter performance is to focus on basic micro-architectural issues such as reducing branch mispredictions, using hashing to reduce trace cache collisions, and minimizing memory footprint. Counter-intuitive to conventional wisdom, it shows that it is irrelevant whether the virtual machine CPU interpreter is implemented in assembly language or C++, whether the guest and host memory endianness matches or not, or even whether one is running 1990's Macintosh code or more current Windows code. This is indicated by the fact that both Bochs and Gemulator exhibit nearly identical average and peak execution rates despite the very different guest environments which they are simulating.

This suggests that C or C++ can implement a portable virtual machine framework achieving performance up to hundreds of MIPS, independent of guest and host CPU architectures. Compared to an x86-to-x86 dynamic recompilation engine, the cost of portability today stands at less than three-fold performance slowdown. In some guest code sequences, the portable interpreted implementation is already faster. This further suggests that specialized x86 tracing frameworks such as Pin or Nirvana which need to minimize their impact on the guest environment they are tracing could be implemented using such an interpreted virtual machine framework.

To continue our research into the reduction of unpredictable branching we intend to explore macro-op fusion of guest code to reduce the total number of dispatches, as well as continuing to split out even more special cases of common opcode handlers. Either of these techniques would result in further elimination of explicit calls of EA calculation methods.

To confirm portability and performance on non-x86 host systems, we plan to benchmark Bochs on a PowerPC-based Macintosh G5 as well on Fedora Linux running on Sony Playstation 3.

We plan to benchmark flash drive based devices such as the ASUS EEE sub-notebook and Windows Mobile phones. An interesting area to explore on such memory constrained devices is to measure whether using fine-grained memory translation and per-block allocation of guest memory on the host can permit a virtual machine to require far less memory than the usual approach of allocating the entire guest RAM block up front whether it ever gets accessed or not.

This fine-grained approach could effectively yield a "negative footprint" virtual machine, allowing the virtualization of a guest operating system which otherwise could not even be natively booted on a memory constrained device. This in theory could allow for running Windows XP on a cell phone, or running Windows Vista on the 256-megabyte Sony Playstation 3 and on older PC systems.

Finally, using our proposed ISA extensions we believe that the performance gap between interpretation and direct execution can be minimized by eliminating much of the repeated computation involved in guest-to-host address translation and computation of guest conditional flags state. Such ISA extensions would be simpler to implement and verify than existing heavyweight hardware virtualization, making them more suitable for use on low-power devices where lower gate count is preferable.

5.1 Acknowledgment

We thank our shepherd, Mauricio Breternitz Jr., and our reviewers Avi Mendelson, Martin Taillefer, Jens Troeger, and Ignac Kolenko for their feedback and insight.

References

[1] **VMware and CPU Virtualization Technology**, VMware, http://download3.vmware.com/vmworld/2005/pac346.pdf

[2] **A Comparison of Software and Hardware Techniques for x86 Virtualization,** Keith Adams, Ole Agesen, ASPLOS 2006, http://www.vmware.com/pdf/asplos235_adams.pdf

[3] **VMware Fusion**, VMware, http://www.vmware.com/products/fusion/

[4] **Microsoft Hyper-V**, Microsoft, http://www.microsoft.com/windowsserver2008/en/us/hyperv-faq.aspx

[5] **Xen**, http://xen.xensource.com/

[6] **Trap-And-Emulate explained**, http://www.cs.usfca.edu/~cruse/cs686s07/lesson19.ppt

[7] **Pin**, http://rogue.colorado.edu/Pin/

[8] **PinOS: A Programmable Framework For Whole-System Dynamic Instrumentation**, http://portal.acm.org/citation.cfm?id=1254830

[9] **Framework for Instruction-level Tracing and Analysis of Program Executions,** http://www.usenix.org/events/vee06/full_papers/p154-bhansali.pdf

[10] **PTLSim cycle accurate x86 microprocessor simulator**, http://ptlsim.org/

[11] **DynamoRIO**, http://cag.lcs.mit.edu/dynamorio/

[12] **DR Emulator**, Apple Corp., http://developer.apple.com/qa/hw/hw28.html

[13] **Rosetta**, Apple Corp., http://www.apple.com/rosetta/

[14] **Accelerating two-dimensional page walks for virtualized systems**. Ravi Bhargava, Ben Serebrin, Francesco Spadini, Srilatha Manne: ASPLOS 2008

[15] **Gemulator**, Emulators, http://emulators.com/gemul8r.htm

[16] **SoftMac XP 8.20 Benchmarks (multi-core)**, http://emulators.com/benchmrk.htm#MultiCore

[17] **Vigilante: End-to-End Containment of Internet Worms,** http://research.microsoft.com/~manuelc/MS/VigilanteSOSP.pdf

[18] **Singularity: Rethinking the Software Stack**, http://research.microsoft.com/os/singularity/publications/OSR2007_RethinkingSoftwareStack.pdf

[19] **Transmeta Code Morphing Software**, http://www.ptlsim.org/papers/transmeta-cgo2003.pdf

[20] **Inside ST Xformer II**, http://www.atarimagazines.com/st-log/issue26/18_1_INSIDE_ST_XFORMER_II.php

[21] **Bochs**, http://bochs.sourceforge.net

[22] **Intel IA32 Optimization Manual**: (http://www.intel.com/design/processor/manuals/248966.pdf)

[23] **Overview of the P4's trace cache**, http://arstechnica.com/articles/paedia/cpu/p4andg4e.ars/5

[24] **Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters** http://www.complang.tuwien.ac.at/papers/ertl&gregg03.ps.gz

[25] **NO EXECUTE! Part 11**, Darek Mihocka, http://www.emulators.com/docs/nx11_flags.htm

[26] **Instruction Mix Test Program**, http://emulators.com/docs/nx11_t1.zip