A Proposal for 80386 VM86 Performance Enhancements

David A. Butterfield

Locus Computing Corporation
3330 Ocean Park Blvd., Santa Monica, CA 90405
(213) 452-2435

ABSTRACT

The *Virtual 86* mode of the Intel 80386 is an operating mode that allows 8086 programs to execute in a protected, paged, multitasking environment under a host operating system containing a *Virtual Machine Monitor*. Programs and operating systems written for the 8086 can execute in this mode without modification or recompilation. This paper reports empirical studies of the behavior of some 8086 application programs and operating systems, describes the impact of the 80386 architecture on the performance of those programs, and proposes certain architectural changes to the 80386 processor that would allow these programs to execute with higher performance. A thorough understanding of the 80386 processor is assumed.

1 Characteristics of 8086 programs

Locus Computing Corporation has implemented a *Virtual Machine Monitor (VMM)* as part of its *Merge 386* product that supports MS-DOS and other operating systems in *Virtual 86 (VM86)* mode under UNIX on 80386-based computers. During the development of this product, we took measurements of the performance of DOS programs running in the virtual machine environment.

We found that the user-perceived performance of the system was significantly influenced by the time spent executing instructions that trap to the VMM. The important instructions for performance are *CLI, STI, PUSHF, POPF, INT n*, and *IRET*, which we will call the *sensitive instructions*, and the I/O instructions.

Under the current 80386 architecture, we took two approaches in this area to improve the performance of the VM86 programs. The first was to minimize the time spent in each VMM trap, by minimizing the overhead time to enter and exit the VMM and by optimizing the code that emulates the sensitive instructions. The second was to reduce the number of VMM traps that actually occur, by modifying the code running in the VM86 environment to perform fewer sensitive instructions.

The first approach has theoretical limits which we have reached. The second approach can be taken for software under our control, but is difficult to implement when running software that an end user buys off the shelf. Nonetheless, by making these improvements, we were able to dramatically improve the user-perceived performance of the system, thus validating the hypothesis that the performance of sensitive instructions is an important factor in overall system performance.

The remainder of this section reports measurements made on unmodified PC environments. For the purpose of these measurements, no special optimizations were made to these environments, either to the BIOS, the operating systems, or the applications, to reduce the number of sensitive instruction traps that occurred in the measured operations.

The instruction mix was measured by keeping a count of all sensitive instruction traps and total real time. Separate counts were kept for each sensitive instruction. Other measurements were used to compute average instruction times with no trapping, and the emulation time for each sensitive instruction. From these it is possible to calculate the instruction mix, that is, the percentage of instructions that trap. It is also possible to calculate the percentage of real time spent emulating the sensitive instructions.

Since instructions that trap take much longer to emulate than the non-trapping instructions take to execute, even an instruction mix with a very small percentage of sensitive instructions can spend a very large portion of its time performing instruction emulation. This proportion of time depends partly on the overhead time to enter and exit the VMM, and so depends on whether the transition to the VMM is through a task gate or through a trap gate. The task gate transition takes about three times as long as the trap gate transition; therefore we give the percentage of real time spent doing emulation for both the trap gate transition method and the task gate transition method.

1.1 Behavior of MS-DOS

We took two types of measurements of instruction traps for the MS-DOS operating system coupled with the system BIOS. We measured an “idle” DOS waiting at the command interpreter prompt. We also measured the system running typical DOS operations, such as copying and typing files.

We found that in an idle DOS environment 1.9% of the instructions trapped, resulting in 66% of the time being spent in emulation for the trap gate method and 78% for the task gate method.

For an active DOS performing simple commands, we found that 0.12% of the instructions trapped, with 17% of the time spent in emulation (23% for task switch method).

1.2 Behavior of Typical MS-DOS programs

For DOS application measurements, we chose two common DOS applications, Lotus 123 and WordStar. These programs were operated over a period of several minutes performing a mix of operations including file I/O, screen I/O, and computation.

During the execution of Lotus 123, 0.53% of the instructions trapped, resulting in 36% of the time being spent in emulation (50% for the task switch method).

For WordStar, 0.62% of the instructions trapped, resulting in 37% of the time being used for emulation (52% for the task switch method).

1.3 Behavior of Other 8086 Systems

Since our virtual machine implementation is designed to run all 8086 software, not just MS-DOS, we also ran measurements using the CPM-86 operating system. For the measured operations, the instruction mix and time spent in emulation were similar to MS-DOS.

2 Effect of VM86 Mode Architecture on Performance

2.1 IOPL and Sensitive Instructions

The VM86 mode architecture provides two options for the behavior of the sensitive instructions. By setting the processor's IOPL to different values, the VMM can specify whether the sensitive instructions will execute ‘‘normally’’ or will trap to the VMM. The ‘‘normally’’ is in quotes because the behavior of the INT n instruction when running in VM86 mode is to trap through the protected mode IDT rather than through the VM86 virtual IVT, as would happen in real address mode.

Unfortunately, if the VMM sets the IOPL to a level that allows the instructions to execute ‘‘normally’’, the VM86 program is allowed to affect the 80386 Interrupt Flag, with the possibility of disabling all system interrupts. Typically this cannot be allowed, so the VMM must set the IOPL so that all the sensitive instructions trap. As noted in previous sections, the time spent trapping and emulating the sensitive instructions is substantial.

2.2 Reflection of INT n Instruction

A correct VMM must reflect all INT n instructions back to the virtual machine for processing. This is because 8086 programs can and do intercept interrupts and INT n instructions by writing over the vector locations in low memory. For example, in the MS-DOS environment there are many examples of programs that add functionality to existing DOS and BIOS calls by intercepting interrupts in this way. Further, the definition of the semantics of a particular interrupt (above the first 32 vectors reserved by Intel) is entirely dependent on the operating system code (including any ‘‘BIOS’’ code) that exists in the system. Therefore it would be incorrect for a pure VMM to implement a system call for some particular 8086 operating system or BIOS, because that would preclude other systems from running under the VMM. And, as noted above, it is not even correct to implement system calls when the operating system is known, because of the many packages that extend the operating system by overwriting interrupt vectors.

It is still possible to achieve performance improvements in certain known VM86 environments (such as a DOS environment) by intercepting the INT n instructions from within the VM86 environment by overwriting the vectors as described above. By intercepting the interrupts early in the life of the virtual machine, before the other programs in the system add themselves to the head of the chain, higher-performance code that implements standard functions receives the interrupts *after* the off-the-shelf code that implements the functional enhancements, allowing those enhancements to work properly.

Since INT n instructions must be reflected for correct VMM operation, the utility of having the INT n instruction trap into the VMM is small. An automatic reflection to the VM86 process would provide a higher performance virtual machine.

3 Proposal for Modifications to VM86 Architecture

The reason that the 80386 must cause the important sensitive instructions (*CLI, STI, PUSHF, POPF, INT n*, and *IRET*) to trap to the VMM is that these instructions allow the VM86 program to read or modify the state of the interrupt flag, which must be virtualized for the VM86 environment to work correctly. With some additional support from the 80386 processor, however, the necessity of trapping on these instructions can be completely eliminated except when the trap is required to deliver a virtual interrupt from a virtual interrupt controller connected to an I/O device.

The proposal addresses (1) how to avoid the necessity for traps on the sensitive instructions, (2) how to cause traps when necessary to allow the delivery of virtual interrupts, (3) the possibility of continued support of the LIDT instruction in VM86 mode, and (4) the possibility of improved compatibility with realmode behavior when executing instructions that inhibit interrupts for one future instruction, such as *STI* and instructions that modify the stack segment register. Proposals (1) and (2) go together and could be of substantial benefit. Proposals (3) and (4) are less important to the performance of the system, but nonetheless are submitted for consideration.

In addition to the changes discussed below, there would presumably also have to be a way to enable or disable this new mode of operation for compatibility with older 80386 processors.

3.1 Virtual Interrupt Flag

In the current 80386, the EFLAGS register looks like this:

```
 31                             15               7             0
+---------------------------+-+-+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+
|0 0 0 0 0 0 0 0 0 0 0 0 0 0|v|r|0|n|io |o|d|i|t|s|z|0|a|0|p|1|c|
|                           |m|f| |t| pl|f|f|f|f|f|f| |f| |f| |f|
+---------------------------+-+-+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+
```

With the proposed change, the EFLAGS register would look slightly different in VM86 mode and 80386 protected mode. In 80386 mode it would look like this:

```
 31                              15              7             0
+-----------------------+-+-+-+-+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+
|                       |v|v| | | | |   | | | | | | | | | | | | |
|0 0 0 0 0 0 0 0 0 0 0 0|i|i|v|r|0|n|io |o|d|i|t|s|z|0|a|0|p|1|c|
|                       |r|f|m|f| |t| pl|f|f|f|f|f|f| |f| |f| |f|
+-----------------------+-+-+-+-+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+
```

whereas in VM86 mode it would look like this:

```
 31                              15              7             0
+-----------------------+-+-+-+-+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+
|                       |v| | | | | |   | | |v| | | | | | | | | |
|0 0 0 0 0 0 0 0 0 0 0 0|i|i|v|r|0|n|io |o|d|i|t|s|z|0|a|0|p|1|c|
|                       |r|f|m|f| |t| pl|f|f|f|f|f|f| |f| |f| |f|
+-----------------------+-+-+-+-+-+-+---+-+-+-+-+-+-+-+-+-+-+-+-+
```

It is not essential that the VIR and IF flags be visible from VM86 mode.

It is proposed that there be two new flags, the *Virtual Interrupt Flag (VIF)* and the *Virtual Interrupt Request (VIR)* flag. The bit positions in the flag word that the IF and the VIF flags occupy would be swapped while executing in VM86 mode. This swapping would only affect where PUSHF and INT get bit position 9 from, and where POPF and IRET put bit position 9. The swapping is important so that the VM86 task will see the correct virtual interrupt flag. Changing the real IF from the VM86 task would be IOPL sensitive, as it is now.

The CLI, STI, PUSHF, POPF, INT, and IRET instructions would no longer trap when executed, but would read or modify the VIF flag rather than the IF flag. The INT instruction would automatically be delivered to the correct address in the VM86 task, without having to be reflected by the VMM.

The effect on system performance should be substantial, because the VMM traps taken due to sensitive instructions are eliminated.

3.2 Virtual Interrupt Request Flag

The Virtual Interrupt Flag described above is not sufficient to support a virtual machine. One of the jobs of the VMM is to deliver virtual interrupts from a virtual interrupt controller to the VM86 task. These virtual interrupts are delivered when a virtual interrupt is pending *and* the virtual interrupt flag is set.

If a virtual interrupt becomes pending while the VM86 task has its Virtual Interrupt Flag clear, the virtual interrupt cannot be delivered at that time, but must wait for the virtual interrupt flag to be set. When the VM86 task enables its virtual interrupt using the STI, POPF, or IRET instruction, the pending virtual interrupt must be delivered at that time. The processor needs a way to know that a trap should be taken into the VMM so that the VMM can deliver the virtual interrupt.

That is the purpose of the Virtual Interrupt Request (VIR) flag. When the VMM decides that a virtual interrupt is pending, it sets the VIR flag in the EFLAGS image of the appropriate VM86 task. While executing in VM86 mode, at the beginning of each instruction the processor checks the state of VIF and VIR. If both flags are set, the processor traps to the VMM so that it can deliver the virtual interrupt. Anytime the VIR bit is set and the VM86 task executes an instruction that enables its virtual interrupt, this turns on the VIF bit resulting in both VIF and VIR being set, causing the trap into the VMM before the execution of the next instruction. (An alternative to checking VIF and VIR at the beginning of each instruction would be to only check it when an instruction that changes the EFLAGS register is executed.)

Without the VIR flag, the processor would have to trap to the VMM anytime VIF transitioned from clear to set, so that the VMM could check for any pending virtual interrupts. With the VIR flag, the trap only occurs when it is needed.

3.3 Virtual Interrupt Vector Table Base

Currently, since the VMM must deliver all the interrupts and INT n instructions to the VM86 task, it has the option of emulating the 80206 realmode function of the LIDT instruction. When executed in realmode this causes the four-byte-per entry IVT table normally located at location zero to move to a new base. This is used by some 80286 software to intercept all interrupts before they are processed by the realmode operating system.

Since under the new proposal the INT n instructions would be delivered automatically through the VM86 virtual IVT, the processor would need a base register to know where the IVT was. Without the base register, the LIDT instruction would not be emulatable. This may not be considered very important, since this feature is not available on the 8086, and it is rarely used on the 80286.

3.4 Virtual Interrupt Inhibit Flag

Certain instructions on the 8086 inhibit interrupts for one instruction following their execution. Examples are STI and instructions that modify the stack segment register SS. Some software depends on this feature for correct execution. For example, the code sequence

```
mov     ss, ss_save
mov     sp, sp_save
```

depends on interrupts not being delivered between the two instructions, while the stack is inconsistent.

In VM86 mode, modification of SS also has this behavior, except in the case where a fault (such as a Page Fault) occurs during the execution of the subsequent instruction. If that occurs, a virtual interrupt could become pending while the page fault was being processed, and if the virtual interrupt were to be delivered by the VMM, it would be on an inconsistent stack.

There are multiple ways to solve this problem. One would be to not execute the move to SS instruction until it is determined that the next instruction will not fault. Another would be to undo the effect of the move to SS and reset the IP if the subsequent instruction faults. These ideas don't seem very easy to implement.

Another possibility is to define another VM86 mode flag, the Virtual Interrupt Inhibit (VII) flag that is set by any instruction that wants to prevent interrupts from occurring until one additional instruction is executed. This flag would be cleared by completion of all other instructions. The processor would use this flag in conjunction with VIF and VIR to decide whether to trap into the VMM. The VMM would also consult this flag (in the EFLAGS image) while deciding whether to deliver a virtual interrupt. If the flag is on when a virtual interrupt becomes pending, then the VMM will just set VIR and return to the VM86 process. When the VII bit resets upon execution of a normal instruction, the processor will see VII=0, VIR=1, VIF=1 and will trap to the VMM, which can then deliver the virtual interrupt.

Thus the VII flag would allow the efficient, correct emulation of instructions that temporarily inhibit interrupts.

4 Conclusions

The Intel 80386 processor provides a high degree of compatibility with the 8086 processor. With appropriate Virtual Machine Monitor software, multiple realmode operating systems can be supported simultaneously on a single processor. The 80386 currently provides a good environment to support such software. The architectural changes proposed in this paper would improve that environment to allow a higher performance virtual machine implementation.