

# Improved SSE3 emulation for the XNU kernel

Prashant Vaibhav  
[voodoo@mercurysquad.com](mailto:voodoo@mercurysquad.com)

Mike Byrne  
[turbo@0xfeedbeef.com](mailto:turbo@0xfeedbeef.com)

*October 2008*

## **Abstract**

The x86 instruction set includes several streaming single-instruction multiple-data opcodes to facilitate vector operations. Many closed source applications make use of these opcodes without checking for processor support, causing a hardware exception to be generated and the process shut down if any of the opcodes is not supported by the processor. There have been attempts to emulate SIMD opcodes included in Intel's SSE3 instruction set by performing the equivalent task using SSE2 in Apple's XNU kernel. However, such efforts have been fraught with problems ranging from poor performance to inability to run multiple concurrent threads. In this paper we describe the implementation of an SSE3 emulator designed from the ground up to be more robust, secure, faster and fully re-entrant. Compared to the previous generation of SSE3 emulator, our emulator has about 2.5 to 3 times faster performance for most opcodes; in some cases being hundreds of times faster. Full multi-thread capability was tested with more than 90 concurrent threads. Security was enhanced by avoiding read-write access to the kernel's commpage area. The emulator has been successfully deployed in an alternative kernel for use with Darwin or Mac OS X on SSE2 processors.

## Table of Contents

1	Introduction	3
1.1	The Maxxus emulator	3
1.2	Shortcomings and motivation	4
2	Design of the emulator	4
3	Implementation	5
3.1	Stage 1 interrupt handler	5
3.2	Stage 2 handler pre-processing	6
3.3	Instruction and operand decoding	6
3.3	Core emulation subroutines	7
3.4	Post-emulation clean up	7
4	Performance analysis methodology	8
4.1	Measurement of trap overhead	8
4.2	Measurement of emulation core performance	8
4.3	Measurement of multi-thread capability	8
5	Performance test results	9
5.2	Performance of emulation engine only	9
5.3	Relative performance	10
5.4	Comparison with native SSE3 CPU	10
5.5	Emulation stages breakdown	11
6	Future work	11
6.1	Opcode patching	11
6.2	In-situ recompilation	12
7	Conclusion and availability	12
	References	12

# 1 Introduction

The general purpose microprocessor market has seen increased interest in hardware support for vector operations in the form of SIMD (Single Instruction Multiple Data) functionality. Desktop and mobile processors from Intel, AMD and other manufacturers have included SIMD opcodes under the brand of MMX™ (MultiMedia eXtension), 3DNow™, SSE (Streaming SIMD Extensions) etc., the latter being available in several layers *viz.* SSE, SSE2, SSE3, SSE4 and further.

Applications that need to use these vector operations typically rely on compiler support to generate object code which makes judicious use of these instructions. However, since support for each sub-category of these opcodes (i.e. SSE2 or higher) is not uniform among currently available processors, most programs employ some sort of processor check before enabling usage of these instructions. In the x86 instruction set architecture (x86 ISA), attempting to execute an instruction not supported by the processor results in a hardware exception. On most operating systems, this causes the program to be abruptly shutdown. There exists poorly (either deliberately or inadvertently) written code which makes use of, e.g., SSE3 instructions without checking for actual support on the processor. Most often the said programs are closed-source, forbidding any (legal) modification to allow them to run on processors not supporting the instruction(s) they use.

It is however, possible to trap the hardware exception generated by the processor, perform the equivalent operation using the instructions available on the processor, and resume execution transparently. This approach has been used by Darwin/XNU kernel hackers *maxxus*, *semthex* and *oui*<sup>1</sup> to implement a functional emulator which emulates SSE3 opcodes via SSE2 operations. However the implementation has been limited in many ways.

## 1.1 The Maxxus emulator

The “previous” generation of SSE3 emulator (referred to as *Maxxus emulator*) was implemented for the XNU kernel – the kernel shipped with Apple’s Darwin operating system, which is the open-source foundation on which Mac OS X is based. The XNU kernel is a hybrid monolithic kernel built from a combination of three components: the Mach<sup>2</sup> microkernel (serving as the basis), the BSD subsystem based on FreeBSD 4 (which provides the network stack and POSIX support), and the I/OKit subsystem (providing hardware abstraction and device driver interface).

The Mach portion of the kernel receives hardware interrupts or exceptions from the processor, which are either handled in-kernel (e.g. page faults), or propagated to the relevant device driver or application (e.g. floating point exception). Many exceptions are propagated to the run time system generically and are thus defined as a generic exception. Invalid opcode exception (INT 6), in a stock XNU kernel, is defined as such. The addresses of the entry point for each interrupt/exception handler is stored in an area of the memory called the Interrupt Descriptor Table (IDT). During the boot process, the Maxxus emulator patches the entry point of interrupt 6 to point to its own specialized handler routine, *after* the IDT had been initialized already. This causes the processor to skip the kernel’s generic interrupt handler and instead, jump to the emulator routine.

The emulator routine consists of two parts: the actual interrupt handler, which is run in the kernel/interrupt context, and the emulation routines which run as user-mode code within the commpage (described later). The interrupt handler is executed within a kernel interrupt context. As such it has its own exclusive stack, which is populated (among others) with the instruction pointer of the faulting opcode. The handler copies this instruction pointer into a writable area in the commpage, patches the return address to the entry point of the user-mode emulation routine, and returns from the interrupt. Once in user-mode, the routine can read the (invalid) opcode which generated the exception (by de-referencing the saved instruction pointer). Based on

---

<sup>1</sup> Said people have preferred to stay pseudonymous

<sup>2</sup> Originally developed at Carnegie Melon University

the opcode, a proper emulation sequence is run and control is returned to the program. Since an interrupt does not alter the processor registers, care must be taken to preserve the state of the registers before returning.

## 1.2 Shortcomings and motivation

While the overall design of the Maxxus emulator is straightforward, the actual implementation is fraught with several issues. These include the inability to emulate more than a fixed (small) number of threads simultaneously, the availability of a readily accessible, writable area in memory, unwieldy size of the emulation routines and inability to easily expand in the future.

Emulation of any opcode entails decoding its operands. In the x86 ISA, the operands are encoded in a ModRM byte immediately after the opcode, followed by an optional 'SIB' (Scale-Index-Base) byte and one of several different 'displacement' values. The total number of possible combinations of operand addressing modes is in the hundreds. Naturally, all such combinations of operands and opcodes cannot reasonably be coded statically. The emulation routine itself was written to be static initially, with place-holders for operands padded by NOP (no-operation) bytes. A separate area at the end of the emulation routine was designated as the temporary storage for opcodes and local variables. During runtime, these placeholders were patched with the actual operands, resulting in polymorphic, self-modifying code. The entire user-mode emulation routine, plus the storage area, was stored in the commpage.

The commpage is an area of memory which is initialized by the kernel during the boot process. This area consists of about 20 memory pages (each page being 4 KB in size), containing machine-specific implementations of several frequently accessed functions (e.g. `bcopy`, `gettimeofday`). The commpage is mapped onto the memory space of every process, giving the C library or other user-mode code automatic access to the platform-tuned version of each function. The entire commpage is ordinarily marked as read-only, for security and performance reasons.

Because the emulator's temporary storage area was in the commpage, it had to be marked read-write despite security concerns. Having to modify the running program several times meant slower performance (due to instruction cache misses). Further, since the commpage area was common to *all* running processes, and not individual threads, modifying it would make the entire scheme inherently non-reentrant. This was solved to some extent by having multiple copies of the emulation routine and its associated temporary storage area. The interrupt handler would check each 'slot' for availability and route the faulting opcode to the next free slot, and failing if none of the slots was free. Because each emulation routine in full consumed about half a page in memory (2KB), an emulator with only 4 slots consumed 2 pages. Increasing the number of slots further was impractical because of limited free space in the commpage. This limited the emulator to just a handful of concurrent threads, frequently overloading on vector-heavy applications. Thus a redesigned emulator which would be entirely statically coded and fully re-entrant was needed.

## 2 Design of the emulator

Our emulator is designed to be re-entrant. This means not relying on the commpage for data storage. Performance reasons dictated not using self-modifying code, and thus the emulator was designed to be statically coded.

These requirements meant the following changes had to be made :

- Use the calling thread's stack to store data. Since each thread has its own stack, synchronization was not needed.
- Decode the ModRM byte to access the operands statically instead of dynamically patching the emulation routines.

Guided by these constraints, our design is as follows :

- The generic INT6 handler is replaced with a ‘stage 1’ handler. This handler stores the faulting opcode address on the interrupted thread’s stack, patches the return address to the stage 2 routine and returns from interrupt context back to the user-mode thread. This routine is written in x86 assembler.
- The stage 2 handler pushes the processor registers on the stack and calls the emulation engine. The emulation engine is written in C (with some inline assembly) to aid future expansion.
- The emulation engine reads the faulting opcode and the following ModRM byte. It then decodes the operands. A few XMM registers are saved and the operands are fetched temporarily.
- Depending on the faulting opcode to be emulated, an appropriate subroutine is called which picks up the decoded operands, performs the emulation, and stores the result in one of the temporarily freed-up XMM registers.
- Once the emulation has been performed, the result is stored back in the actual destination, the temporary registers are restored to their original state, and control is returned to the instruction immediately following the faulting opcode. This results in the interrupted program continuing execution transparently.

We shall now examine the implementation in detail.

## 3 Implementation

The actual implementation was written in a mix of x86 assembly language and C. The stage 1 handler was written in pure assembly. During an interrupt, all running threads are suspended and other interrupts are disabled. It is imperative to keep every interrupt handler as fast and short as possible. The stage 2 handler (and its emulation routines for each opcode) runs within a single user-mode thread’s context, and hence was written in C, with short pre- and post- emulation routines to save and restore the processor state. Discussion of each stage follows.

### 3.1 Stage 1 interrupt handler

The x86 platform generates interrupts or exceptions to signal an unusual condition during execution of a regular program. During an interrupt, the currently running program is suspended and the processor jumps to the interrupt handler of the corresponding interrupt. A total of 256 interrupts are available to system software and hardware, the first 32 of which are reserved by the processor. One of them is the Invalid Opcode exception (interrupt 6h), generated when the processor encounters an instruction it does not support. This is the case when SSE3 instructions are executed on an SSE2-only processor. The job of the first stage of the interrupt handler is to trap this exception and export it to the user-mode emulation engine.

The interrupt vector is initialized by the XNU kernel during the boot-up sequence, where INT 6h is assigned to a generic interrupt handler which propagates the exception to the runtime environment. In order for our replacement handler to be executed instead, the boot-up code (kernel source tree `osfmk/i386/start.s`) was modified. A check is made for SSE3 support (using the `CPUID` instruction). If the machine is found to be SSE3 capable, boot process continues normally. Otherwise, immediately after the IDT is initialized, the 6th vector is overwritten with the address of our stage 1 handler in the commpage (address `0xFFFF4000`).

When the processor enters the stage 1 handler, it is given its own stack<sup>3</sup>. The processor pushes the code segment, contents of the ESP register (containing a pointer to the interrupted thread’s stack), contents of the EIP register (which contains the instruction pointer on x86 platform) and the return instruction pointer (same

---

<sup>3</sup> Kernel-context (ring 0) handlers are isolated on the x86 architecture to prevent stack corruption of user threads

as the faulting instruction) on the stack. The stage 1 handler dereferences the stack pointer of the interrupted thread, and ‘pseudo-pushes’ the faulting EIP on the stack. Since the stack could originally have had any value on the top which must be preserved, the EIP is written above the address pointed to by the stack pointer. This ensures that the thread’s own stack is not corrupted. During this process, the stage 1 handler pushes the registers it uses (EAX and EFLAGS) to its own stack for later restoration. The top of interrupt context stack contains the return address (which points to the original faulting instruction). Once the EIP has been written on the interrupted thread’s stack, this return address is patched to the stage 2 handler’s entry point. When an IRET is then executed, the processor exits from the interrupt context, switches to user mode and jumps to the stage 2 handler instead of back to the faulting instruction. In addition, if a special opcode of 0xFFFF is found, the handler returns immediately to the instruction following the 2 bytes. This is designed to aid in finding the minimum trap overhead (ref. section 4. *Performance analysis*).

This stage is intentionally kept short so that only about 10 to 15 extra clock cycles<sup>4</sup> are spent within the interrupt context, and execution of other threads is interrupted only for a very short duration.

### 3.2 Stage 2 handler pre-processing

This stage is executed by the processor in the context of the original thread with the invalid opcode. As such, there could be multiple simultaneous threads all in an emulation stage, but executing in a single-threaded context. As long as all of the data is local to the thread, multi-thread synchronization is not needed. Because of this, we make exclusive use of the stack space only, which is guaranteed to be thread-local.

Once in the stage 2 handler, the processor’s registers are immediately saved by pushing them on the stack (using PUSHAX/PUSHF instructions). Since several of the operand addressing modes use the processor registers as a base, scale or index, access to the contents of these registers is required by the emulation engine. When the registers have been pushed on to the stack, the top of stack contains the faulting EIP and each of the registers’ contents. The C-language emulation engine is then called. This function is declared to have its arguments in the same order as the top of the stack. Because of the C calling convention, the function gets easy access to all the values exported from the stage 2 pre-processor, as its arguments.

### 3.3 Instruction and operand decoding

The first operation performed in the emulation engine is to decode the instruction and its operands. In the x86 ISA, each instruction is of variable length, ranging from 1 byte to 15 bytes. An instruction is composed of the opcode of 1 to 5 bytes (plus prefixes), followed by the operands. The operands are specified either as immediate values, or in encoded form. All of the SSE3 opcodes use the encoded form, where the operands are specified in terms of the registers and 8, 16 or 32 byte displacement. Every SSE3 opcode can work with operands in register-to-register, register-to-memory or memory-to-register form (memory-to-memory is not available). The operands are encoded by the ModRM byte which specifies which addressing mode is to be used. The ModRM byte is followed, in some cases, by an SIB byte which specifies a scaled-index + base addressing mode. This is then followed by a displacement value.

Taking into account all the combinations, there can be close to 500 individual addressing modes for each opcode. It was imperative to keep the emulation core simple and thus insulated from the complexities of each addressing mode. Therefore, this stage of the emulation engine decodes the operand addresses using simple conditional logic and some calculation, before transferring control to the relevant emulation core. When the addressing mode refers to one of the registers, the saved value passed from the stage 2 pre-processor is used. Following this, two of the XMM registers are saved to prepare them to be used as temporary operands. The operands are then fetched and stored within the XMM registers before branching into the actual emulation subroutine. Since we also calculate the overall length of the opcode plus its operands, we also increase the saved return-EIP by the same number of bytes so the execution returns to the *next* instruction after emulation is completed. This step provides a static set of source and destination registers for the emulation subroutines to use, thus freeing them from including any conditional logic.

---

<sup>4</sup> Not counting ~430 cycles processor overhead for any interrupt

### 3.3 Core emulation subroutines

Once the operands have been decoded and the temporary XMM registers populated, we are ready to perform the actual emulation. The emulation core consists of an array of thirteen routines, each of which performs the corresponding function of their respective SSE3 opcodes.

These operations can be performed using regular non-vector x86 opcodes as well, which would make it possible to emulate them on any x86 hardware; however the speed benefit is lost in this case. Instead, we chose to build the emulation routines using SSE and SSE2 opcodes. In most cases, only a handful of cleverly thought-out operations are needed to fully emulate the SSE3 opcode.

To perform the emulation, the faulting SSE3 opcode is simply checked in a **switch ... case** statement, and the corresponding emulation routine is run. The comparison was a simple integer equality as all SSE3 opcodes (with the exception of **fsttp**) are 12 bits long. For **fsttp**, we made the comparison before the switch block for the other opcodes, and ran the emulation routine depending on the size of the operands. The emulation routines store their results in the XMM register we cleared as the temporary destination.

Note that this emulation ‘routine’ is not a C-function call — rather, we perform the emulation within the **case** block itself (but refer to these blocks as ‘emulation routines’).

A design decision was whether to find the most efficient methods to emulate a given SSE3 opcode using SSE2, or to simply perform the operation in the simplest manner even if it takes a few extra clock cycles. We chose the latter for clarity, as the trap overhead of ~400 cycles was already the defining factor of how fast the emulation of each opcode would run and a few extra clock cycles within the emulation core would not make any significant difference to the overall emulation speed.

### 3.4 Post-emulation clean up

Once the emulation core has finished, the result from the computation has been stored in the temporary destination XMM register. The post-emulation section of the code takes this value and stuffs it in the real destination as specified in the original SSE3 opcode (this had already been decoded in the operand decode stage, *cf. section 3.2*). After storing the results, the temporary XMM registers are restored to their original state by copying the values saved during the operand decode stage.

After this, the ‘stage 2’ C-function exits immediately into an assembly language epilogue. This epilogue performs the reverse operation of stage 2 preprocessing. During the execution of the C function, the processor registers were most likely clobbered by the compiler. Since the interrupted process is using these registers for its own purposes, we must restore these to maintain transparency. These values had been saved on the stack by the preprocess step, but because of the C calling convention, the stack pointer will have been decremented on exiting the C function. Thus we increment the stack pointer once again to the top of the last saved value so that a subsequent pop would work. The **POPA** and **POPF** instructions are then executed which pop the previously-saved values of the registers, restoring the original conditions of the registers.

At this stage, the top of the stack contains only the return-EIP (instruction pointer) which has already been appropriately incremented to point to the instruction following the faulting instruction. Hence, a **RET** instruction is executed which pops the EIP from the stack and returns execution to the interrupted thread. This step also restores the stack pointer of the interrupt thread to what it was before the emulation routine was called.

The emulation is thus considered complete and the interrupted thread/process continues with the next instruction, unaware of the fact that the previous instruction was emulated. Typically the next instruction is also an SSE3 instruction which is then emulated again.

The only way for the process to tell whether an instruction was emulated or ran natively is to measure the execution time and compare it with the time usually taken to execute the same SSE3 opcode natively. This is used in analyzing the performance of our emulator in the next section.

## 4 Performance analysis methodology

The emulator's performance was measured chiefly against the *Maxxus* emulator. Given the ~430 cycle trap overhead, it is impossible to achieve a reasonable performance against hardware SSE3 operation. Hence, the only useful metric for measuring the emulator's performance was the total number of cycles less the trap overhead.

A second metric against the old emulator was the number of simultaneous threads that the emulator could support. Anything above 4 threads is an improvement, as practical considerations limit the usage of the old emulator to about 4 simultaneous emulation threads.

### 4.1 Measurement of trap overhead

Trap overhead is the number of cycles taken by the processor to switch to an interrupt trap context following attempted execution of an invalid (SSE3) instruction. To measure this, we included a dummy opcode (`0xFFFF`) in stage 1 of the emulator. This dummy opcode is not implemented on any Intel compatible processor, so encountering this in an executable would signal the invalid opcode interrupt. The stage 1 handler checks for this opcode and simply returns on encountering this. This allowed us to measure the time taken by the processor to switch into the interrupt context and exit, with minimal processing (only one comparison) in between.

### 4.2 Measurement of emulation core performance

To measure the emulation core performance, a simple test utility was developed. The utility ran a user-specified number of iterations of each of the SSE3 instructions on the processor, times them, and averages the result over the number of iterations. To measure the time with a high degree of accuracy, we used the TimeStamp Counter (TSC) found on modern Intel processors. The TSC, on SSE2 processors (and on SSE3 processors running at full rated clock frequency) increases with each processor clock cycle. We recorded the TSC's value before and after running the iterations, and divided the difference by the number of iterations to find the average number of clock cycles taken to emulate each instruction.

From this value, we subtracted the average trap overhead to arrive at the number of cycles taken by the emulation core. Note that this simple test would not account for the effects of processor caching. Indeed, it is entirely possible that the entire emulation loop would fit in one cache-line. Whether cached or non-cached performance is a more useful metric is open to debate.

We also measured the total number of cycles (including trap overhead) taken to emulate each instruction, to compare against the number of cycles taken on a real SSE3 processor.

### 4.3 Measurement of multi-thread capability

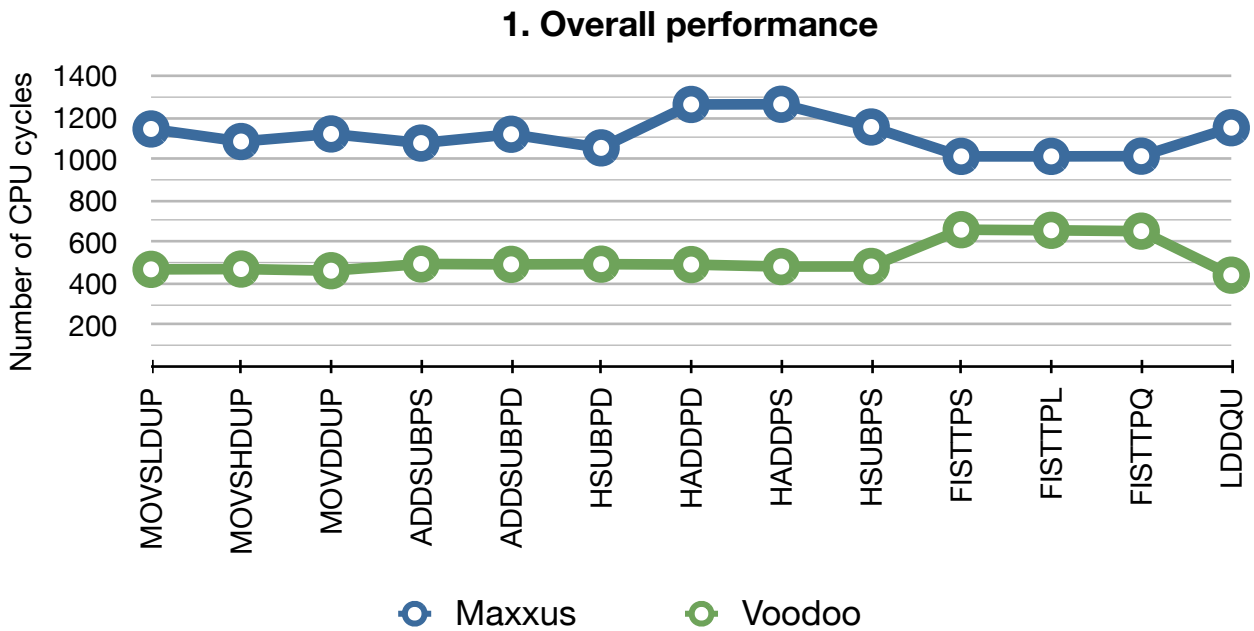
The design of the emulator enables it to support an unlimited number of threads at the same time. Nevertheless, we ran a concocted test to test its ability to emulate a large number of threads at a time. Tests were run in two ways: the first test was based on running several applications known to use SSE3 opcodes (e.g. iTunes, QuickTime Player, DVD Player, Front Row and Logic Pro), and performing generic tasks in each. This did not conclusively prove anything, however.

Therefore, we ran to successful completion several simultaneous instances of the "stress-test" utility developed to measure core performance (described in section 4.2). Since each of these instances ran several million iterations of SSE3 opcodes, this proved conclusively that the emulator is capable of supporting several concurrent threads.

# 5 Performance test results

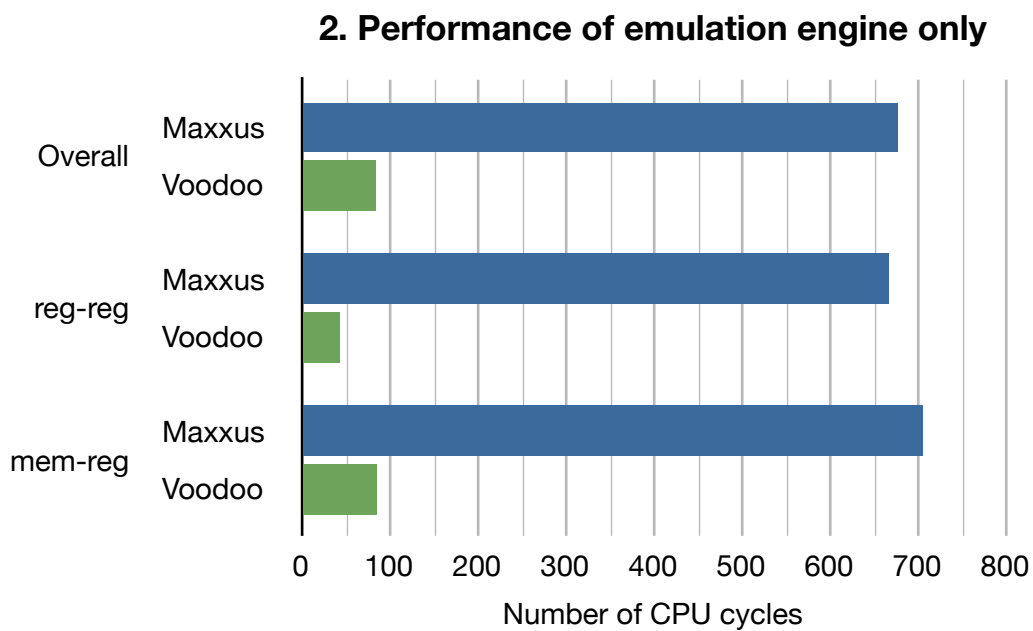
Figures below show results of our performance tests.

## 5.1 Overall performance



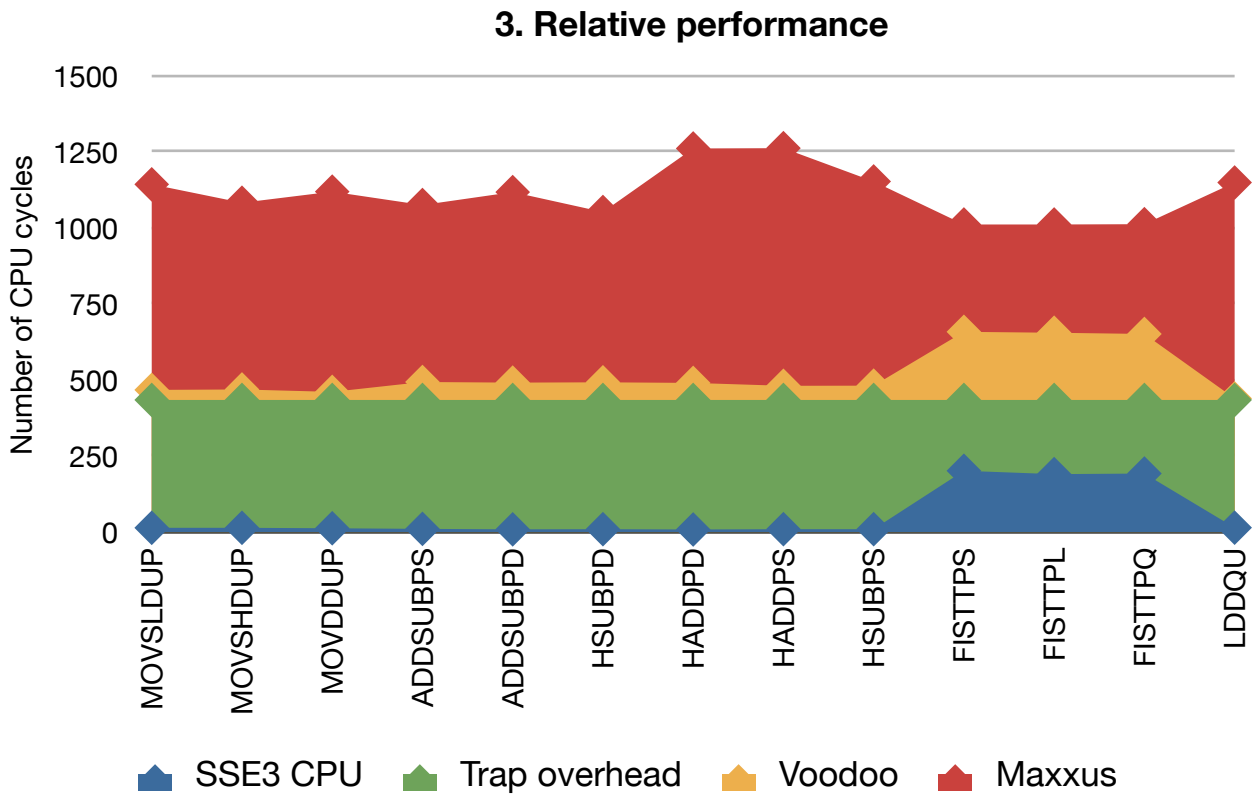
This graph describes the overall performance of the “Voodoo” emulator described in this paper, compared to the old “Maxxus” emulator. The comparison is made for each emulated op-code, and includes the trap overhead. As is evident, our emulator outperforms the Maxxus emulator by a factor of about 3.

## 5.2 Performance of emulation engine only



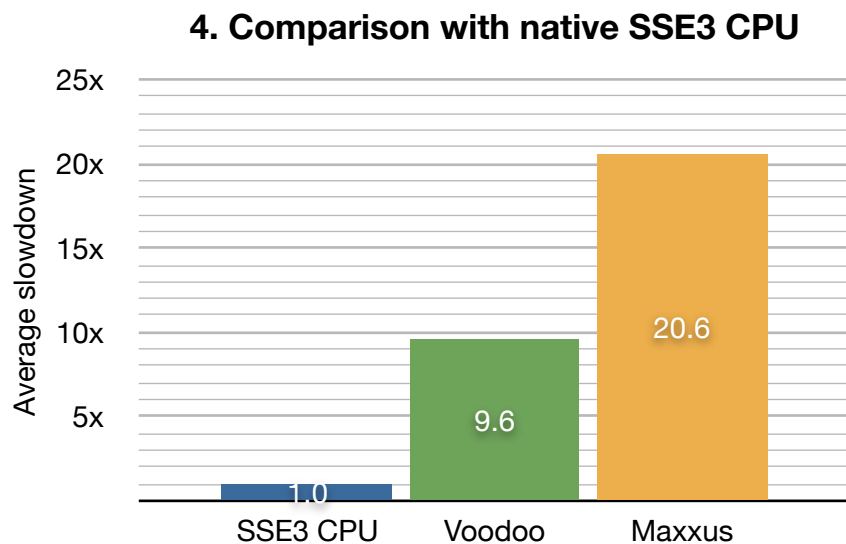
The figure above compares performance of the emulation engines. This does not include the 430-cycle hardware trap overhead (which is unavoidable). Memory-to-register operations take slightly longer than register-to-register operations, however, the Voodoo emulator still outperforms the Maxxus emulator by a factor of almost 7.

### 5.3 Relative performance



The graph above describes relative performances of various SSE3 implementations. The hardware trap overhead is included for perspective. Evidently, the Voodoo emulator emulates each instruction very efficiently compared to the Maxxus emulator. The bottleneck is clearly the trap overhead.

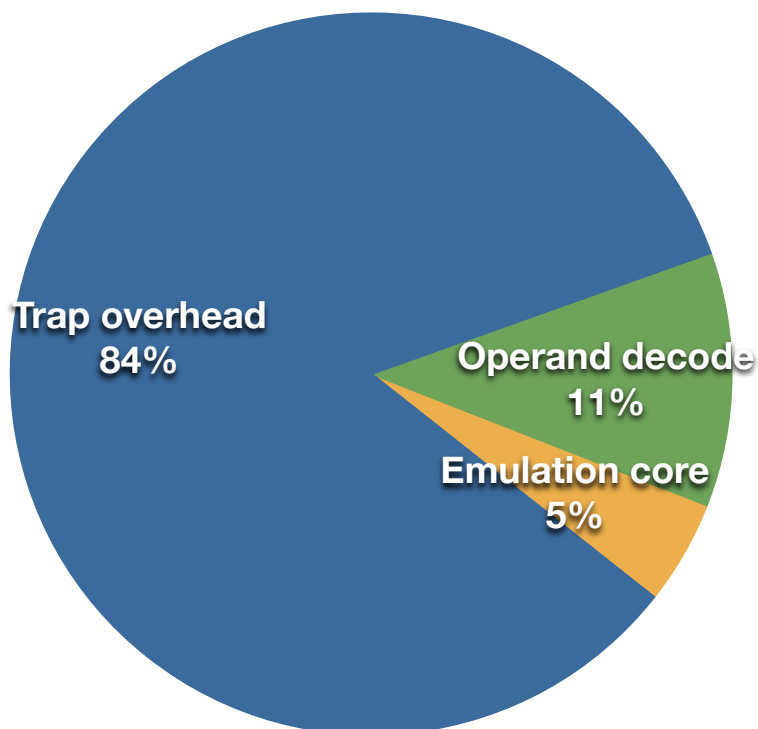
### 5.4 Comparison with native SSE3 CPU



The graph above gives a better perspective on how both these emulators compare to a native SSE3 processor. If we take a native SSE3 processor as taking 1.0 time unit to perform an SSE3 operation (on average), the then Voodoo emulator takes about 9.6 times longer, while the Maxxus emulator takes about 20.6 times longer.

## 5.5 Emulation stages breakdown

### 5. Emulation stages breakdown



The above chart provides an indication of where the time is being spent during an emulation iteration. The hardware trap overhead is by far the biggest time-consuming step, taking about 84% of the clock cycles during the emulation. This is followed by the operand decoding stage, which takes 11% of the clock cycles. Lastly, the actual emulation core which emulates an SSE3 operation via SSE2 opcodes, given the operands, takes only 5% of the total emulation time. This corroborates our design choice of not optimizing for the best emulation core strategy, but rather to focus on other orthogonal areas like avoiding cache pollution.

## 6 Future work

### 6.1 Opcode patching

Further work in this area includes the possibility of patching binaries loaded in memory, on first invocation of the emulator. We tried this with the `LDDQU` instruction which was patched to `MOVDQU` which performs the identical function, but does not avoid cacheline splits; and with the `FISTTPx` opcode group, which we replaced with their corresponding `FISTPx` instruction.

While the experiments were largely successful, for the final version of the emulator we chose to drop these optimizations for various reasons; the two chief reasons being: 1) Patching binaries from the emulator caused problems with the Unified Buffer Cache, resulting in seemingly random system freezes and kernel panics which we did not investigate further, and 2) The amount of space we had to patch-in a replacement instruction was not always enough to overwrite it with a functionally equivalent set of instructions. For example, replacing `FISTTPx` with `FISTPx` would cause incorrect results with edge-cases, which although seldom encountered in real-life applications, were still incorrect from a theoretical point of view.

## 6.2 In-situ recompilation

There is a lot of scope for expanding on our emulation strategy: it could, for example be used to emulate a completely different opcode set (e.g. AMD's 3DNow!). Indeed, any op-code not directly supported by an Intel x86 compatible processor can be transparently emulated through this method.

However, the biggest issue, as described above, is the trap overhead. Continuing from the opcode patching idea from section 6.1, we could take it one step further and dynamically recompile a section of the binary, upon the first invocation of the interrupt. This recompilation would entail recalculating all jump offsets, moving the text segment forward by the number of bytes required to slot the replacement routine in place, and modifying all jumps to point to the newly calculated locations. This is a lot of work, however, something similar is done in commercial software virtualizers like VirtualBox and VMWare, or solutions like Apple's Rosetta. With some work, it should be possible to include their recompilation engine into the 2nd stage "emulator" so that non-native binaries can be transparently recompiled and run on processors which might not support all (or any) instructions present in the binary.

## 7 Conclusion and availability

We have described our strategy to emulate SSE3 opcodes transparently on a native host, and shown that it is very efficient, secure and multi-threading capable. Any further improvement in the performance rapidly approaches the point of diminishing returns, as the biggest bottleneck is the processor's hardware trap overhead. Since the trap overhead is a fixed quantity, we believe our emulator achieves the optimum effort-vs.-performance gain in this particular emulation strategy. We have further discussed future possibilities of exploiting our strategy to include transparent emulation of non-native binaries via opcode patching and dynamic recompilation, which could alleviate the trap overhead bottleneck.

The Voodoo SSE3 Emulator is currently available as part of the Voodoo port of the XNU kernel<sup>5</sup>. It has been deployed on thousands of desktop and laptop systems, and is able to boot an entire Mac OS X desktop on SSE2 hardware. The emulator is capable of supporting multimedia-intensive applications like QuickTime and Logic Pro.

## References

- [1] Intel Software Developer's Manual, Volumes 1 - 3
- [2] Published source code for Maxxus/semthex/oui emulator

---

<sup>5</sup> <http://code.google.com/p/xnu-dev>