# Advanced virtualization techniques for FAUmachine

Hans-Jörg Höxer Volkmar Sieh Martin Waitz

Institut für Informatik 3
Friedrich-Alexander-Universität Erlangen-Nürnberg
Germany

info@faumachine.org

***Abstract***

*This paper presents advanced virtualization techniques used to implement the virtual PC FAUmachine. We created a just-in-time compiler that can transform kernel mode code into code suitable for direct execution within a user mode simulator. This approach allows the efficient virtualization of standard PC hardware. To improve the performance of our simulator, we developed a small host kernel modification that simplifies system call redirection to the virtual machine. These approaches are described in detail and their performance is evaluated.*

## 1 Introduction

Virtualization of hardware is a topic of great interest both for the commercial and scientific sector. The main motivation for our work is to build a virtual machine that provides a realistic hardware simulator which is able to simulate hardware faults. Such a fault injector can be used in dependability benchmarks [1, 4].

Our team developed a virtual PC formerly known as UMLinux, now called FAUmachine. To ease virtualization, we used a specially modified Linux kernel as guest system. One of our main targets was, that the changes needed to port an original Linux kernel to our virtual environment should be minimal. We replaced certain assembler instructions with calls to the virtual PC, which simulates these instructions. This approach has proven to work very well and has the important benefit, that the virtual machine completely runs in user space. No special modules or extensions to the hosting Linux kernel are needed.

Nonetheless, two major drawbacks have shown up. First, we can not run system level binaries for which we have no source code. This includes binary-only Linux kernel modules and operating systems like Windows. Second, we use `ptrace(2)` to redirect system calls issued by user processes running on FAUmachine to the kernel running on FAUmachine. The introduced overhead degrades the performance of the virtual system significantly. To remedy these problems we implemented a just-in-time compiler and an extension to the hosting Linux kernel.

The just-in-time compiler which is able to automatically convert kernel mode code into user mode code is described in section 2. The kernel extention used to accelerate system call handling in the virtual machine is described in section 3. The performance of both methods is then evaluated in section 4.

### 1.1 Terminology

When working on virtualization of hardware, one always has to work with several systems: host and guest. The host system is the physical system which is used to run the virtual machine. The guest system is provided by the virtual machine. It is not build with physical hardware, but solely consists of virtual components. Of course, the software responsible for these components has to run on physical hardware (the host system), which is not directly available to the guest.

In our example, both host and guest systems are PC compatible computers (i386 architecture).

### 1.2 Overview over FAUmachine

Each CPU of the guest system is simulated with one user mode process on the host system. The virtual memory available for these processes is used to simulate the address space of the guest CPU. The MMU of the guest CPU is simulated with `mmap(2)` and `munmap(2)`. However, a normal process is only allowed to use three out of four GBytes address space (on Linux, values for other systems vary). Access to the last GByte is not possible efficiently so we use modified guest kernels that only access the lower three GByte. A solution for this limitation is being worked on (see section 5).

The simulator that is responsible for the virtual hardware and some special CPU instructions is mapped into the address space of the virtual CPU. It is responsible to simulate everything needed by the guest system and not available in a normal process. Signals raised by the host operating system are converted into exceptions of the guest CPU. The simulator handles the communication between the guest operating system and the virtual hardware. All the state machines for all that virtual hardware are implemented in the simulator.

A frontend process is responsible to provide interaction with the host system. It is able to display the virtual console in a GUI or terminal window. Keystrokes and mouse movements in those windows are transformed into events that are sent from the virtual mouse/keyboard to the guest operating system. The frontend and simulator processes are linked with several sockets to be able to communicate with each other.

For details on the FAUmachine hardware simulator see [3].

## 2 Just-in-time compiler

Virtualization is greatly simplified when host and guest architecture match. The host CPU can be used to directly execute the code that is to be executed by the virtual CPU. But this approach is not possible when the virtual CPU tries to communicate with peripherals – it has to talk to virtual peripherals and not to those connected to the host CPU.

This doesn't raise any problems as long as the virtual CPU is in user mode in which direct access to hardware is prohibited. The host CPU can simply execute those code parts on behalf of the virtual CPU.

If the virtual CPU switches to kernel mode, things get more complicated. The host CPU could also enter kernel mode to be able to execute those instructions [5]. However, the security of the host system can easily be compromised if the privileged code is not monitored carefully.

In order to execute virtual kernel mode code in a user process of the host machine, some modifications to the code are necessary. These modifications can be made at run time or at compile time.

Previous versions of FAUmachine used specially crafted guest kernels to be able to run inside the simulator. There are several problems with this approach. To be able to modify the guest operating system, it has to be modified in source code. But it is not always possible to obtain the source code of the operating system in question. Even if the source code is available, those modifications have to be repeated for every new version of the operating system.

In order to overcome these problems, code has to be converted automatically while running it. This section describes how this is done in the current version of FAUmachine.

### 2.1 Differences between kernel and user mode

Multitasking is a very important feature for todays computers. In order to be able to reliably run different processes in parallel, there has to be some program that arbitrates between all running processes. This special program is called the operating system kernel and is given special privileges by the CPU.

The CPU distinguishes between user and kernel mode. In kernel (privileged) mode, a program has complete control over the machine. It can access all the memory and all hardware resources. Processors have two very distinct modes: Kernel and user mode.

In user mode, only a subset of the functionality is available. All operations that are possibly harmful are not allowed by the CPU.

The operating system provides system calls that are used to provide a pre-defined set of safe operations that are executed on behalf of the user

process. When a system call is invoked, the processor will enter kernel mode and the operating system kernel is responsible for execution of this call.

We want to build a complete virtual CPU, including user and kernel mode. Yet the simulator is only allowed to run in user mode of the host CPU, like any other application. That way it is not possible for the virtual system to compromise the host operating system, but we have to find a way to create a fake kernel mode. The approach taken is described below.

## 2.2 Running user-mode code in a virtual machine

As the simulator runs in user mode itself, it is capable of running user mode code of the guest system directly. No special handling is necessary as the code in the guest system is not allowed to directly access hardware, anyway.

Leaving this virtual user mode has to be detected reliably. When an exception or system call occurs, control is given to the operating system. The simulator has to ensure that the guest operating system is used to handle that exception instead of the host operating system.

As normal user-space process, it is not possible for the simulator to inhibit the invocation of the host operating system. But it is possible for the simulator to be notified about these events. Exceptions are normally passed to the responsible process as a Unix signal. The simulator registers signal handlers for all possible exceptions (SIGSEGV, SIGILL, SIGFPE, ...) and simulates that exception in the guest CPU. Normally, only system calls are not converted into a signal so that other methods have to be used to catch those. System call handling is described in detail in section 3.

## 2.3 Simulation of individual instructions

Simulating kernel mode code is a bit more difficult as the virtual machine is not able to execute it directly. In user mode, only a small subset of the processor state is available. In order to be able to simulate a complete processor with kernel mode, a virtual version of the entire state is needed. This virtual processor state has to be stored in memory, in order to be accessible by user mode code. This state has to include all the CPU registers (general purpose, floating point, flags, various control registers and so on).

When an operation is simulated, it will work on this in-memory copy. All instructions are reimplemented as C code that mimics the functionality of the real CPU. All virtual hardware (e.g. hard disks, network and video cards) is also simulated with C code. A function of such a hardware simulator will be called when the virtual CPU tries to talk to it.

The simulated versions of the CPU instructions are much slower than the real ones. Therefore the simulator should only be used when necessary. Many instructions behave the same in user and kernel mode and can still be executed directly even when the virtual CPU is in kernel mode. Only those instructions that behave differently in user and kernel mode have to be simulated. Whenever the virtual CPU reaches code that has to be simulated, it has to activate the simulator. Whenever it reaches code that can be executed directly, it should switch to direct execution of that code part.

These transitions between direct execution and simulation are made possible by synchronizing the real and virtual CPU state. As instructions that are executed directly don't use the full CPU state, only the general purpose, flags and segment registers have to be synchronized. The registers of the real CPU are saved to the in-memory version just before simulating an instruction and are restored from there before starting direct execution again. As the C code of the simulator needs its own environment to be able to run, segment and flags registers and the stack pointer are set to special values while the simulator is running.

## 2.4 Running kernel code in a virtual machine

Most instructions that have to be simulated trigger an exception in the processor when they are executed from user mode. The simulator process will get a signal from the host operating system when this happens and can take appropriate actions to simulate that instruction instead.

However, this approach is slow because it requires a round trip to the host operating system and doesn't work for all instructions. Namely the `pushf` and `popf` instructions are problematic. They are used to save and restore the flags register, which holds information for both com-

parison results and for overall processor configuration, including privilege level, interrupt enable bit and so on. Some bits of this register are only available in kernel mode, some are always available. Exactly this dual use poses a problem: access to the register is always allowed, but it behaves differently in user and in kernel mode.

In kernel mode, the entire flags register may be saved and restored. However, user processes can only save and restore some bits of this register. All other bits are simply ignored. No exception is generated, so it is not possible to detect that an access to the flags register failed.

Another example is `iret`. The set of registers restored depends on the current privilege level.

As it is not possible to get notified about `pushf`/`popf`/`iret` usage, we have to check for it before issuing such an instruction. If every instruction gets checked before execution anyway, then it is possible to efficiently handle other special cases as well. Each instruction is compared against the list of instructions that have to be simulated. Exceptions are not used to detect to-be-simulated instructions. Only "real" exceptions which would also occur on a real system have to be handled (page and segmentation faults, divide-by-zero, etc.). Those are handled via a normal Unix signal handler.

Checking instructions is very slow compared to the time it takes to execute them. Several clock cycles are needed to determine whether an instruction has to be simulated or executed directly. Doing that for every instruction is prohibitive performance-wise as all kernel code would be slowed down extremely. Therefore, a method to reduce these checks is needed.

## 2.5 Caching compiled code

The decision whether an instruction has to be simulated or not does not depend on the current CPU state and thus only has to be done once for each instruction. Therefore, it is possible to cache information about how a given piece of code has to be run.

This is exactly the approach taken by our just-in-time (JIT) Compiler. It transforms a piece of arbitrary code into code that only consists of instructions that can be issued in user mode. This new code is a mixture of original code and calls to our simulator. There is no performance penalty for most instructions. Only those that have to be simulated anyway because they access hardware or processor flags have to go through the extra jump into the simulator.

Using this cache of native code, even kernel code can be efficiently executed in the virtual machine. But the cache still has to be filled before it can be used. Instructions are added to the cache one by one. When the virtual CPU is about to execute an instruction that is not yet part of the cache, it is compiled and added to the cache. Afterwards, the compiled version is executed and the next instruction is looked up in the cache. If more instructions would be fetched at once, an exception could be generated too early, for example when the page holding the following instructions is not available.

This compilation phase is rather expensive but only has to be done once for each instruction. Experiments have shown that our current cache size of about four megabytes is big enough to hold a complete Linux kernel. Once the kernel has booted in the virtual machine, compilation of new code is only necessary when the kernel executes new code paths, which is very infrequent.

## 2.6 Managing the cache

Efficient access to the modified instructions is crucial for the performance of the simulator.

Those modified instructions cannot be stored in the original code segment as the simulated process would be aware of the modifications if it reads its own code. There are several programs (mostly boot loaders) which use self-modifying code or which calculate a checksum of itself so that this approach cannot be used. The modified code has to be stored in extra memory as part of the simulated CPU.

If the code is not run in its original location, there is a discrepancy between the real (called `REIP`) and cached instruction pointer (called `CEIP`). This is not a problem for most instructions. It only matters for `ret` and `call` instructions. Those read and write the current instruction pointer (`%eip`) from/to the stack. Code that examines its own stack may break if a return address stored there contains unexpected values. Examples are all debuggers (they need the return address to identify the function called) and code that uses global variables in position independent code. In order not to break stack contents, `call` and `ret` instructions have to be simulated, too. The simulator correctly maps between cached and real addresses so that the stack always contains correct values.

Another problem is that the modified code is a little bit larger than the original code most of the time. An one-to-one mapping of original and modified addresses is not possible because of the resulting displacements inside the code segment. To address this problem, the code segment is split into many pieces called cache lines. Each of these cache lines consists of some unmodified code and only one modified instruction at the end. The delta between `REIP` and `CEIP` is always constant inside a single cache line. That allows to easily convert one to the other. The original address (`REIP`) of the first instruction in a cache line is stored explicitly, all others are calculated from that one.

As the cache gets split up into several lines, it is important to be able to quickly find the correct cache line. There are several situations that need different methods to find a cache line:

- Find the cache line holding the compiled code for a specific `REIP`.
- Find all cache lines that contain code from a common page. They have to be invalidated when the page gets modified after the code was compiled.
- Find all cache lines that contain jumps to a cache line. These jumps have to be adjusted if the target cache line is invalidated.
- Find the next free cache line.

The first two situations are solved by using hash tables. All cache lines that share the same hash value are linked in a list. To find a cache line, the hash value of the requested address is calculated and the corresponding list is linearly searched for the cache line in question.

```
unsigned long
cpu_jit_range_hash0(unsigned long eip)
{
        return (eip / CACHE_LINE_SIZE) % RANGE_HASH_SIZE;
}
unsigned long
cpu_jit_page_hash0(unsigned long page)
{
        return (page / 4096) % PAGE_HASH_SIZE;
}
```

The hash functions try to generate identical values for all instructions within a cache line. This is achieved by assigning one hash value to an entire block of addresses. As long as a cache line is inside such a block, it will get the same hash value for every instruction. A cache line can touch at most two hash value blocks as the maximum cache line size is smaller than the hash value block size. If the search for a cache line at a given index is not successful, then it is assumed that the cache line crosses a hash value boundary. The cache line is then looked up using the hash value of the preceding hash value block.

The other situations concerning cache line searching involve allocation and invalidation of cache lines. All cache lines are chained by a doubly linked list. When a cache line gets allocated, it is put to the tail of this list; if it gets invalidated it is moved to the head of the list. When a new cache line is needed, it is simply taken from the beginning of the list. That one is either free already or contains the least recently allocated cache line. A true least recently used strategy would be better but would require to instrument the code running in the cache line so that actual usage is detected. However, with a large enough cache size there is no need for advanced allocation strategies. Our tests have shown that a few megabytes are enough to hold the compiled version of a Linux kernel.

Cache lines have to be invalidated if their original code changes or if there are no more free cache lines. To detect changes in the original code, pages are marked write protected when code contained in them is being compiled. If the content of the page is being modified, a page fault is triggered. The simulator catches this page fault and invalidates all corresponding cash lines, found by the hash table mentioned above. All jumps into such an invalidated cache line are now dangling and have to be removed in turn. To simplify the search for cache lines that contain such a jump, there is a linked list of referencing cache lines. This reference list is updated each time a jump is inserted into a cache line.

## 2.7 Code generation

Cache lines are filled one instruction at a time. Each cache line that is not yet complete (i.e. the cache line only consists of unmodified code and is shorter than the maximum length) is terminated with a special jump into the simulator. When the control flow of the virtual CPU reaches the end of the cache line, it will hit the jump and cause a fetch function to be called in the simulator. This function evaluates the next instruction of the original code and decides whether it can be executed directly or whether it has to be simulated. The appropriate code is then inserted into the cache line, replacing the jump to the fetch function. The cache line has to be terminated again, unless the new instruction already was an

unconditional jump. This is achieved by either jumping to another existing cache line which can be used as continuation or inserting a new jump to the fetch function.

Return from compiled code to simulator is not that straightforward:

- All register values have to be saved before being modified.
- The current stack must not be used.

The simulator must not modify the stack of the guest system; there may not even be a valid stack available. Therefore, normal call instructions cannot be used and another method is needed to save the return address. Other registers cannot be used as they would have to be saved in turn, too. As the simulator has to update its in-memory CPU state anyway, the in-memory copy of the instruction pointer is the natural target to store the instruction pointer. This is achieved by inserting an instruction that explicitly modifies the in-memory copy of the instruction pointer. Control is then transfered to the simulator which saves all other registers and executes the requested function.

When an instruction is simulated, then a decoded version of that instruction is saved in the cache line to simplify simulation. This instruction is read and executed by the simulator. Afterwards it restores all register values and jumps into the cache line holding the next instruction.

If an instruction doesn't have to be simulated, it is compiled into the cache. This compiled form may differ from the original instruction in some situations.

Jumps are always specified using relative addresses in the i386 architecture. To optimize code size, only a 8 or 16 bit offset will be used when the jump target is near enough. However, the compiled code does not have the same layout and it may not be possible for the jump target to be representable in the original instruction. This is a problem especially with conditional jumps which are only available using an eight bit offset. For example, a simple `je target` may have to be compiled as `jne else; jmp target; else:` in order to reach its target.

Code is always executed in 32-bit mode. If 16-bit code is to be simulated then the JIT has to insert or remove address and operand size prefixes accordingly. With those prefixes, it is even possible to execute BIOS code in the native 32-bit mode of the processor.

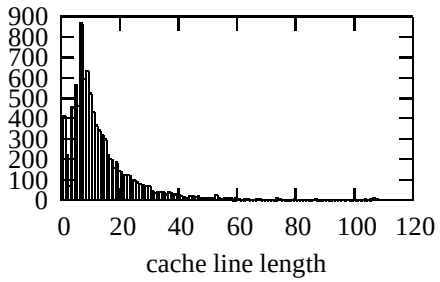


**Figure 1: Distribution of cache line sizes after booting Linux**

If the length of an instruction changes because of some modifications, then the current cache line is finalized and new code has to go to another cache line. A jump to the new cache line is appended in this case.

As seen in figure 1, most cache lines have to be finalized early and only contain very few instructions. Only a fraction of the cache memory can be used for code. That is impaired by the fact that cache lines are allocated for the target of a conditional jump even if it is not taken. About one fourth of the allocated cache lines were empty after booting a Linux kernel (not shown in figure 1 because it would dominate the graph). The JIT already works very well but optimizations still have to go on.

# 3 System call redirection

System calls issued by processes running on the virtual machine impose a special problem that is to be solved by FAUmachine. Both FAUmachine and guest processes use system calls to execute functions of the kernel they are running on. FAUmachine uses functions of the hosting kernel whereas a guest process has to use functions of the guest kernel. As the guest's userland code is executed by the CPU directly, system calls issued by the guest will trigger the system call handler of the hosting kernel.

To understand this issue we take a closer look in the next section on how system calls can be implemented on the i386. Section 3.2 summarizes the problem and outlines two methods solving this problem.

The first method uses standard mechanisms provided by Linux but has a noticeable performance overhead. It is introduced in section 3.3. The second method is based on a modified Linux host and is discussed in section 3.4.

We conclude this chapter with a description of how the simulator handles the redirection in a way common to both approaches.

## 3.1 System calls and ABI

The i386 provides two mechanism for system call invocation: "Call gates" and "software interrupts." Both allow to transfer control from an unprivileged user context to a privileged system context in a secure and controlled manner. Which method is used and and how parameters are passed to the system call define the so called "Application Binary Interface" (ABI).

### 3.1.1 Call gates

To use call gates a gate descriptor is entered in the "Global Descriptor Table" (GDT) or "Local Descriptor Table" (LDT). The gate descriptor references a code segment descriptor and provides an offset into this segment. This segment can have a more privileged level than the current one. For details see [2].

When executing a far call instruction (`lcall <selector>, <offset>`) specifying this call gate selector, the processor loads the referenced code segment and continues execution at the defined offset. The privilege level is set appropriately. Additionally to the instruction pointer the previous selector is saved on the stack, so the previous context can be restored by executing a `lret` instruction. For example, this mechanism is used by Solaris/x86.

### 3.1.2 Software interrupts

Software interrupts are generated with the `int <number>` instruction which expects an interrupt number as argument. Similar to hardware interrupts and exceptions, the system has to provide an interrupt gate descriptor in the "Interrupt Descriptor Table" (IDT) to be able to handle a particular software interrupt. Like a call gate the descriptor for an interrupt gate references a code segment selector, an offset into this segment and the privilege level to use. When executing an `int <number>` instruction, the processor loads the corresponding code segment, sets the privilege level and continues at the specified offset. The previous process context is saved on the stack and is restored when executing the `iret` instruction. To name a few, Linux and the operating systems of the BSD family use the software interrupt 128 (`int $0x80`) to implement system calls. Again, for detials see [2].

### 3.1.3 Binary compatibility

FAUmachine is binary compatible to the i386. Thus user space applications are run on the guest system without any modification and are executed by the host CPU.

As a consequence applications will issue system calls natively as required by the ABI of the guest kernel. For example, an application running on a Linux guest uses `int $0x80` for system call invocation. This instruction is executed directly by the hosting CPU. If the hosting kernel has installed a handler for this interrupt – which is the case on a Linux host – the CPU traps into the hosting kernel and executes this interrupt handler and not the interrupt handler of the guest kernel.

## 3.2 The problem and possible solutions

We have to distinguish system calls issued by the simulator from those issued by the guest system. System calls of the simulator must be handled by the host, those of guest processes must be redirect to the guest kernel.

Note that system call redirection is only a special case of a more general problem: Actually there is a conflict between usage of gate descriptors on the guest and the host. As soon as guest and host are using the same descriptors we have to distinguish references to these descriptors by the guest from references by the FAUmachine simulator itself. In case of FAUmachine running on a Linux host we have to take care for the interrupt descriptor 128 ($0x80).

We came up with two solution for this problem: The first one uses the `ptrace(2)` debugging facility of Linux, the second one is a modification of the Linux kernel which provides a more efficient mechanism than `ptrace(2)`. The next section discusses the first approach using `ptrace(2)` and section 3.4 introduces the second solution.

## 3.3 Redirection using `ptrace(2)`

To make clear how we use `ptrace(2)` we give a short summary of its functionality. Then we describe in the sections 3.3.2 and 3.3.3 how we redirect system calls with `ptrace(2)`. We

conclude in section 3.3.4 with a short discussion of the overhead of this technique.

### 3.3.1 Synopsis of `ptrace(2)`

The `ptrace(2)` system call provides mechanisms for controlling and tracing the execution of a child process by its parent. It is used by debugging tools like `strace(1)`.

The child can explicitly request to be traced by specifying the request `PTRACE_TRACEME` as argument for `ptrace(2)`. Alternatively, the parent can use `PTRACE_ATTACH` to initiate tracing of an already existing process.

When traced, the kernel stops the child each time a signal is delivered. The parent – in the following called "tracer" – is notified on its next `wait(2)` system call and has several possibilities to manipulate the child. With the request `PTRACE_PEEKDATA` and `PTRACE_POKEDATA` the parent can read and modify data in the child's address space. The parent can also examine and modify the child's registers with the requests `PTRACE_GETREGS` and `PTRACE_SETREGS`.

There are three possibilities to resume the child. First, the tracer sends the request `PTRACE_CONT` and the child will again be stopped on receipt of the next signal. Second, the parent enables single stepping with `PTRACE_SINGLESTEP`: The child will only execute the next instruction and stops again. And third, the parent uses `PTRACE_SYSCALL`. Then child resumes execution and stops automatically at the next system call attempt or when a signal is deliverd. On return from a system call to user space the child is also stopped. Thus the tracer can examin the child before and after execution of a system call.

### 3.3.2 FAUmachine tracer

When using `ptrace(2)`, FAUmachine forks a tracer process on startup. This tracer process first forks the CPU process, the core of the simulator, and then suspends itself using `waitpid(2)`.

The CPU process executes the simulator core and represents the CPU of the virtual PC. This means, it is also running the guest's code. Having finished initialization the CPU process issues a `PTRACE_TRACEME` request to be traced by its parent, the tracer process. From now on every signal delivered to the CPU process will stop it and notify the tracer via `waitpid(2)`. To synchronize with the tracer, the CPU process sends itself a `SIGTRAP` to allow the tracer to start tracing of the CPU process.

When the tracer returns from `waitpid(2)`, it resumes the child with the `PTRACE_SYSCALL` request. From now on tracer and CPU process are synchronized, i.e. any signal delivered to the child or any system call attempted by the child will stop it and notify the tracer.

### 3.3.3 Redirection with `ptrace(2)`

As soon as the child attempts a system call, it is stopped and the tracer is notified. The tracer retrieves the registers of the child with `PTRACE_GETREGS` and examines the current instruction pointer, which points to the `int $0x80` instruction generating the system call software interrupt. As the simulator code resides at a well known virtual memory area, the tracer is able to distinguish a system call attempted by the guest from one by the simulator by looking at the EIP.

The procedure to deal with a system call attempt by the simulator is easy as it does not need to be redirected. The tracer just resumes the child with `PTRACE_SYSCALL`, which continues execution of the system call in the host kernel. On return from the system call it is again stopped and resumed by the tracer. So the simulator code of the child can issue regular system calls which are served by the host kernel.

If the software interrupt has been generated by code of the guest, then the tracer has to redirect this system call to the guest kernel. This procedure is more complicated. The child has already trapped into the host kernel and the initiated system call cycle must be completed. But there is no valid context for a system call from the guest to the host – from the guest's point of view it even might not be a system call at all.

However from the host's point of view this is a system call attempt and we have to finish it without causing further side effects. To accomplish this, the tracer modifies the number of the requested system call. This number is stored in the EAX register of the caller. The tracer saves the original value of EAX and sets it to the value of the `getpid(2)` system call, which has no side effects as it just returns the current process number. Then the child continues and finishes the `getpid(2)` system call and is stopped again on return to user space.

The again notified tracer restores the EAX registers of the child process with the original values

and continues the child with a SIGINT. Upon resumption, the child enters the signal handler of the hardware simulator which will arrange for the child to return to the guest's system call handler. This procedure is outlined in 3.5.

### 3.3.4 Tradeoff

System call redirection based on `ptrace(2)` is expensive. It involves four context switches between tracer and CPU process, several system calls issued by the parent (`ptrace(2)` requests to examine and modify registers of the CPU process) and a system call to the host kernel by the child just to consume the initiated software interrupt cycle.

Each system call attempted by the simulator code also involves four process switches and one system call to examine the child's registers.

## 3.4 Kernel redirection

This section describes our approach for a more generic redirection mechanism implemented in the host kernel. This mechanism can be used to easily redirect system calls with low overhead. In the next section we discuss our design decisions, in 3.4.2 we introduce the implementation and conclude with an overview of how FAUmachine uses this mechanism.

### 3.4.1 Design decisions

In the context of FAUmachine this mechanism has to meet the following requirements: First, system calls from the hardware simulator must be handled as usual by the host kernel, whereas system calls from outside the simulator – i.e. from the guest – need to be redirected. And second, the host kernel has to generate a signal to notify the process about the system call attempt.

The first requirement is necessary, as both the simulator and the guest are part of the same process and address space. Therefore our redirection mechanism has to split the user portion of the virtual address space in two distinct areas: One where system calls are served as usual and one where system calls are redirected. Thus we have to provide the host kernel with the address range in which the simulator code resides. This is similar to our approach using `ptrace(2)` where the tracer knows the simulators address range.

Using signals to notify the process about a system call attempt is quite natural, as the signal handling facility provides all the necessary means to do the actual redirection: The signal handler has full control of the interrupted process and can arrange for the process to continue on `sigreturn(2)` in the guest kernel (see 3.5). We decided to deliver the signal SIGINT.

There is no difference for the simulator between handling redirection done by the kernel or by the tracer. The simulator can use both mechanisms transparently and can run on both vanilla and patched hosts.

Note that there are only two traps into the host kernel involved: First, the system call attempted by the guest system and second the `sigreturn(2)` system call of the signal handler. There is no need anymore to switch to another process like the tracer from section 3.3.

To parametrize and control the redirection mechanism we decided to add a new system call, called `sys_faumachine_helper()`.

### 3.4.2 Kernel Implementation

This section describes the implementation of the kernel redirection mechanism. The code was originally written for 2.4 kernels but is now also running on 2.6 kernels. The redirection code for 2.4 differs only marginally from the 2.6 code. This paper focuses on the code for 2.6.

The code is split in two parts, a machine dependent and a machine independent part. The machine dependent code for i386 resides in `arch/i386/kernel/faum.c`, the machine independent in `kernel/faumachine.c` respectively. We also modified `arch/i386/kernel/entry.S` to add the new system call and to alter the system call entry function `system_call()`. A context diff between the original and modified kernel is about 230 lines, not counting debugging code and experimental features.

The modified Linux kernel provides a new system call to control and parametrize the redirection mechanism:

```
int sys_faumachine_helper(int request, unsigned long lower,
                          unsigned long upper, unsigned long level)
```

The argument `int request` chooses the actual function to be used, as the system call also provides a debugging facility and other experimental features that we will not discuss further.

For system call redirection only the request `REGISTER` is relevant. When issued,

the parameters `unsigned long lower` and `unsigned long upper` specify the address range from where system calls are allowed. When the parameter `unsigned long level` is set to 1 the redirection mechanism is enabled. The request `REGISTER` is handled by the function `register_faumachine()` in `kernel/faumachine.c`.

To remember this address range we have added two variables `unsigned long lower` and `upper` to the task structure `struct task_struct` (see `include/linux/sched.h`). To monitor whether redirection is enabled or not we added a new `ptrace(2)` flag, `PT_TRACE_SIM`, which is also set by `faumachine_register()`.

The handler specified by the descriptor for the software interrupt 128 is the function `system_call()` in `arch/i386/kernel/entry.S`. On any system call attempt the flow of execution moves from userspace directly to `system_call()`. This function examines the system call number provided in the register EAX and calls the actual function implementing the system call. There are also checks, whether the current process is traced by another one. In this case, the system call is stopped and a signal is delivered to the traceing process. We hooked our redirection mechanism into `system_call()`.

The modifications to `system_call()` are simple. Similar to the already present checks for `ptrace(2)` being enabled we now check for `PT_TRACE_SIM`. This additional check imposes a small but negligible overhead on each system call, even when redirection is disabled. See section 4.1 for a discussion of this overhead.

When `PT_TRACE_SIM` is not set, we just continue with the system call. Otherwise, we call `test_faumachine_boundary()` in `kernel/faumachine.c`. This function checks whether the current instruction pointer is outside the range specified in `struct task_struct`. If yes, a SIGINT is delivered by calling `send_sig()` and the system call is aborted. The signal handler of the FAUmachine simulator will catch the SIGINT and proceed as described in 3.5. When the EIP is inside this range, the system call continues as usual.

This mechanism reduces the cost of system call redirection significantly. Redirecting a system call is now about as expensive as delivering a signal. For a performance comparison with the `ptrace(2)` approach see section 4.1.

### 3.4.3 Userland implementation

In this section we delineate how FAUmachine uses the kernel redirection mechanism.

On startup FAUmachine tries to execute the `sys_faumachine_helper()` system call. If this fails – the call returns with the value ENOSYS – FAUmachine uses `ptrace(2)` (see 3.3). Otherwise FAUmachine forks the CPU process directly without prior forking a tracer process. During the initialization phase the CPU process enables the system call redirection.

There is also a command line option to choose explicitly between the `ptrace(2)` and kernel redirection.

## 3.5 How the simulator handles redirection

This section describes how the FAUmachine simulator handles the redirection to the guest. The procedure described is common to both mechanisms, `ptrace(2)` and kernel redirection.

The simulator handles signals generate by exceptions – SIGFPE, SIGSEGV, SIGILL, SIGBUS – in the function `sigexception()` (in `node-pc/simulator/cpu_core.c`). The trap number is retrieved from the `struct ucontext`[1] which is passed to the signal handler by the host kernel. The simulator keeps the state of the guest's CPU in the `struct cpu` and the trap number is saved in `cpu->f_trapno` and the `cpu->f_type` is set to `CPU_FAULT_EXCEPTION` to indicate an exception.

The SIGINT delivered by the redirection mechanism is also handled by `sigexception()`. As it was not caused by a real processor exception we get no useful trap number. Thus we set `struct cpu->f_trapno` to $0x80 and `cpu->f_type` to `CPU_FAULT_SYSCALL` to indicate a software interrupt.

In case of a valid exception, `sigexception()` calls `cpu_core_call_exception()`. This functions retrieves the appropriate descriptor from the guest's GDT, LDT or IDT and derives the virtual address of the handler function. Then it adjusts the EIP and EFLAGS registers and segment selectors that are stored on the signal stack.

[1] The `struct ucontext` describes the whole state of the processor at the time the signal was generated. See `/usr/include/sys/ucontext.h` for details.

| machine | time |
|---|---|
| real machine | $141s$ |
| tracer and JIT | $1547s$ |
| kernel redirection and JIT | $635s$ |
| tracer without JIT | $1439s$ |
| kernel redirection without JIT | $514s$ |

**Table 1: Time for kernel compilation**

On return from the signal handler the modified registers and segments are loaded to the CPU which resumes execution in the specific handler function.

# 4 Performance benchmark

In this section we compare the performance of FAUmachine using `ptrace(2)` with FAUmachine using the kernel redirection. We also measured the overhead introduced by system call redirection and the JIT.

## 4.1 FAUmachine speedup

As benchmark, we use the compilation of a Linux kernel from source. It involves computing, disk I/O and memory usage and therefore gives a good estimate for the performance of a machine. We used an AMD Athlon XP 2100+ with 1 GByte of memory as host machine and compiled the sources of the Linux 2.4.18 kernel. For each measurement we extracted the source tree and performed the following commands: "`make oldconfig`" and "`make dep`" to prepare the compilation and then "`make bzImage`" to actually compile the kernel. We measured the duration of "`make bzImage`."

We applied this benchmark to five different targets: First to the host itself to get the performance of the real machine. Then second on FAUmachine using the tracer and JIT enabled, third on FAUmachine using the kernel redirection and JIT enabled, fourth FAUmachine using the tracer and JIT disabled and finally FAUmachine using kernel redirection and JIT disabled. The results are shown in table 1.

When not using the JIT, the kernel redirection improves the performance significantly: Compared to the real machine, FAUmachine is only about 3.6 times slower, whereas the tracer is about 10.2 times slower. Thus modifying the host kernel is a sensible approach which increases the performance of FAUmachine about the factor 2.8. When using the JIT, the overall performance is degraded, but kernel redirection increases the performance about factor 2.4.

| kernel | time |
|---|---|
| patched | $139.50s$ |
| vanilla | $139.41s$ |

**Table 2: Kernel redirection overhead**

## 4.2 Redirection overhead

As discussed in section 3.4.2, the redirection mechanism imposes a overhead for each system call even on processes, that do not use kernel redirection. To measure this overhead we used the same benchmark as in section 4.1 and ran it on a real machine with a vanilla Linux kernel and a patched kernel with redirection support. Again we used an AMD Athlon XP 2100+ with 1 GByte of memory.

As one can see in table 2, the patched kernel is only slightly slower than the vanilla kernel. At least with regard to our simple benchmark the overhead imposed by system call redirection seems to be negligible.

## 4.3 JIT overhead

Using the JIT instead of a pre-modified guest kernel imposes a performance overhead for the kernel code of the guest system as it has to be analyzed and modified. The compiled code is slower than the original as it contains more instructions and is spread over more pages, hurting processor cache performance. The effect on performance can be seen in table 1: the JIT version is about 14% to 24% slower, depending on the system call redirection method used. The cause for this dependency between compilation method and system call redirection method is not yet fully identified.

# 5 Current research

One topic of our current research is the use of the full 4 GByte address space for FAUmachine. As long as the code segment of a Linux kernel running on the virtual machine does not overlap with the segment of the hosting kernel, Linux can run on top of Linux easily. When they overlap, the virtual machine has to simulate not only special instructions, but also all instructions referencing

that overlapping segment. This has a huge performance impact. To get rid of this limitation, we are working on an enhanced version of the existing 4-GByte-patch, which will allow us to make use of the full 4 GByte address space.

# 6 Conclusion

We have presented details of two new techniques used in the FAUmachine virtual machine. We showed how arbitrary kernel code can be transformed into code suitable for direct execution by the simulator using a just-in-time compiler. This allows us to run a large number of operating systems in our virtual machine. We can already run the majority of Linux and OpenBSD systems inside FAUmachine. We are currently still working on improving the compatibility of the simulator to the real hardware, in order to run other operating systems like Windows. In order to increase performance, we created a small patch for the operating system kernel of the host machine. The modification allows to redirect system calls efficiently and is used by the simulator to detect systems calls issued by the guest system. The performance impact of both methods on the virtual machine has been evaluated.

# References


[1] K. Buchacker, Mario Dal Cin, H.-J. Höxer, R. Karch, V. Sieh, and O. Tschäche. Reproducible dependability benchmarking experiments based on unambiguous benchmark setup descriptions. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 469–478, 2003.

[2] Intel Corporation. Intel architecture software developer's manual (volume 3: System programming guide), 1999.

[3] H.-J. Höxer, K. Buchacker, and V. Sieh. Implementing a user mode linux with minimal changes from original kernel. In J. Topf, editor, *9th International Linux System Technology Conference, Köln, Germany, September 4-6, 2002*, pages 71–82, 2002.

[4] H.-J. Höxer, K. Buchacker, and V. Sieh. UMLinux - a tool for testing a linux system's fault tolerance. In *LinuxTag 2002, Karlsruhe, Germany, June 6-9*, 2002.

[5] McKee and A. Bret. System and method for monitoring execution of privileged instructions. U.S. Patent 6,694,457, Hewlett-Packard Development Company, March 6, 2001.