Memory Virtualization with Shadow Page-tables in Karma

Divij Gupta, 9th September 2010

1. Introduction

An increase in the number of errors arising from the software complexity of today's systems, has resulted in an active interest in micro-kernels as they show promise in making the problem of building trustworthy systems more tractable. Building such systems requires the trusted code base (TCB) to be as small as possible, and with it the number of defects that can potentially impair an application. As the kernel code of an OS runs in the most privileged mode by definition, it is crucial to minimize its size, which is the primary objective of micro-kernel based systems. As a result such systems can be better reasoned about their trustworthiness by:
a) Increasing trust in the correctness of the implementation, since a small kernel lends itself to thorough inspection or even formal verification techniques, which do not scale beyond a few thousand lines of code.
b) Providing increased isolation between components to avoid availability and confidentiality issues. That is, a misbehaving component should have a minimal impact on the others in the system.

Micro-kernels strive to achieve better component isolation by placing functionality that is more suited to be a user service, such as device drivers, file-systems and protocol stacks within user-land 'servers'. This is because such functionality is not required to be part of the TCB of the application, in fact it could prove to be detrimental if not verified properly. Hence, the kernel only handles aspects which need to consider the system as a whole, such as scheduling, page table management, memory protection and capability tracking. User-programs can then communicate with these 'servers', to carry out the requested functionality on their behalf. However, it is the responsibility of the micro-kernel to provide the mechanism for setting up of secure communication channels, according to the policy specified at user level. Although theoretically it is possible that such steps might prevent an application from being affected if say the file server it is communicating with crashes, in practice such a bug might prove to be fatal for the system. Hence, while micro-kernels might not be full-proof they do offer benefits in cases where the failed component could potentially be killed and restarted, which is very useful for some low level device drivers such as for network-interface cards (NIC).

Real-time tasks also greatly benefit from micro-kernel systems. By restricting the non real-time functionality to user mode, micro-kernels ensure that there are reduced chances of such functionality interfering with the operation of the system, in particular the scheduler. For example, some synchronization schemes in device drivers use IRQ disabling for exclusive access to resources for a certain amount of time, this could potentially interfere with the scheduling requirements for systems dealing with real-time tasks. As a result, components with such intricate synchronization schemes, the temporal behaviour of which might be unpredictable, can be accommodated as user servers.

By running most of the functionality in user-land servers, the micro-kernel allows developers to write their own servers which might better serve the needs of the system in question, instead of having to modify certain components directly in the kernel which could lead to spurious code becoming part of the TCB, as well as backwards compatibility issues for applications built on top of the micro-kernel application programming interface (API).

However, the stronger isolation of components in micro-kernel systems comes at the cost of reduced performance in comparison with monolithic kernels demonstrating similar functionality. The effects result from work-load characteristics such as privilege transitions and memory context switches, which are far more in number in micro-kernel based systems.

2. Micro-kernels vs Monolithic Kernels

The monolithic kernels of today such as Windows and Linux are in stark contrast to the principle of micro-kernel design. By including functionalities in kernel mode that a micro-kernel chooses to implement in user mode, they suffer from bloated TCB's, the negative effects of which become prominent during system crashes resulting from inaccurately verified applications that are incorporated into the kernel by unsuspecting users. However, monolithic kernels are in excessive use, despite their deficiencies. This is particularly due the immense functionality, support and compatibility with other applications that comes with them, having evolved over a considerable period of time. Hence, a solution to the problem of building trustworthy systems, which can prove to be both practical and feasible from an implementation standpoint as well as acceptable for the end-user, is to run a monolithic kernel as a component on top of a micro-kernel based system. This would not only provide the functionality of monolithic kernels, but also the security benefits of micro-kernels. There are two main ways one can go about this, the first is to run a re-hosted operating system as a user program, and the second to run a paravirtualized operating system in a virtual machine, monitored by a virtual machine monitor (VMM).

3. Running Guests on the Fiasco Micro-Kernel

The Fiasco micro-kernel which is part of the L4 micro-kernel family, already has implementations of the mentioned options. The re-hosted Linux based guest is known as L4Linux which runs directly as a user process in Fiasco. However re-hosting a commodity operating system to run on Fiasco demands a large number of changes to the OS. This is because the guest code which was meant to run directly on hardware, must be modified to make use of Fiasco's application binary interface (ABI) for security sensitive operations as well as the L4 run-time environment (L4Re), to communicate with user servers for services such as device access and memory allocation. For example, instead of manipulating the page-table directly and activating it by loading a pointer to it in the page table base register of the processor, thereby returning to the guest via the IRET instruction, L4Linux has to create tasks and populate them with Fiasco's secure address space construction scheme before it can resume guest execution with a syscall to Fiasco. Adapting the extremely large code base of such guests is a challenging task.

The other option of running a guest in a virtual machine (VM), requires fewer modifications to the guest. However, building efficient VM's was not feasible before the VM related deficiencies of x86 were remedied by the introduction of virtualization extensions. If at all, a basic solution was to run the guest via binary translation. Binary translation works on the basis of the VMM having to emulate the guest instruction on the host by means of translation from the guest instruction set to the host instruction set. However, this leads to a significant slowdown in the operation of the guest, primarily due to the overhead involved in code generation by the VMM as well as context switches into the host kernel to execute the emulated guest instructions. Hardware virtualization techniques by Intel (VT - Virtualization Technology)

and AMD (SVM - Secure Virtual Machine), offer a workaround to this problem by allowing a VM to run directly on hardware, which is setup using special assembly level instructions such as VMLAUNCH and VMRESUME in Intel. However, the VMM can set certain bits to correspond to instructions or events that need to be intercepted, such as device accesses, external interrupts and so forth. As a result the guest can run with minimal intervention from the VMM, which considerably speeds up its operation.

The Fiasco implementation consists of a micro-kernel extension responsible for world switches between the VMM and VM. It is specifically tailored to be compatible with the hardware virtualization provided by the platform it is running on. The VMM as a user-level component coordinates the VM's execution and acts a mediator to other micro-kernel services.

4. Karma VMM

Karma, a VMM implemented to run on Fiasco, operates on the basis of hardware virtualization extensions. It runs a paravirtualized flavour of Linux (where the CPU, memory and most platform devices are fully virtualized), requiring about 3500 lines of modifications to the code. These modifications are primarily required to make the guest aware that it is running in a VM. Both Intel and AMD have a special instruction (vmcall and vmmcall respectively), from which a guest can trap back into the VMM, this is analogous to syscalls being made in an operating system. These instructions plays an extremely important part in the para-virtualization of the guest as it allows the guest to make a call to the VMM (with arguments passed within general purpose registers) to handle certain functionalities such as device accesses or setting of model-specific registers (MSR's), which might need to be intercepted by the VMM in order to take appropriate action via the micro-kernel. This provides an alternative for the VMM to intercept certain functionality of the guest, since the hardware allows the VMM to only intercept specific instructions. Hence, a front end for certain functionalities is provided in the guest, which makes use of the vmcall/vmmcall instructions, with the back-end implemented in Karma to handle these calls based on the arguments passed along with it.

Karma can be thought of as dealing with three main aspects for a guest, a) register virtualization, b) memory virtualization and c) device virtualization.

Register Virtualization - For each guest, the set of registers which is architecturally visible to it when it has complete control of the system, such as control registers, general purpose register (eax, ecx, edx and so forth) and debug registers, are duplicated so that their values do not interfere with that of the host. Each guest has a data structure which hold the values of these registers, so that on launching/resuming a VM these are loaded into the physical registers, to carry on the execution of the guest from its previous state. On a VMEXIT, which returns control to the VMM, the register values of the guest are saved back to the data structure, so that the VMM can later inspect the latest execution state of the guest.

Memory Virtualization - To present the guest with the notion that it has complete control over its memory, the VMM allocates a data-space which is presented to the guest as the amount of RAM available to it. A data-space is an abstraction provided by Fiasco, which allows a user program to allocate a certain amount of physical memory which corresponds to a certain virtual address space. Fiasco tracks this address space, as well as any other tasks to which this address space might be mapped, so that any memory access is appropriately resolved to the

corresponding physical address. Hence kernel code as well as kernel data structures such as the Interrupt Descriptor Table (IDT) and General Descriptor Table (GDT), are all placed within this address space. While the guest views the memory as extending from 0 up-to the size of the data-space, in terms of the Fiasco kernel it extends from some base (virtual) address of Karma to base address plus size of the data-space. On making a world switch from the VMM to the guest, the host kernel's page table pointer is loaded, so that the host retains control over the page-table used for guest execution. This is because using the guest's page table would lead to a clash between the memory layout the guest expects and the actual layout since it is running on top of the host kernel. Hence, these privilege transitions do not modify the memory context of the guest, instead they just affect its accessibility.

Device Virtualization - As discussed, the VMM implements a back-end to handle vmcall/ vmmcalls from the guest to emulate device functionality. On inspecting the arguments passed to it, the VMM can then emulate device reads by returning the appropriate values (such as from the real-time clock for time functionality) in the guest's registers, and device writes by communicating with the appropriate device servers using the L4 run-time environment. The VMM also delivers interrupts and events such as device interrupts and software exceptions (such as page faults), through a special field in the VM's data structure. These events are then delivered to the guest through the IDT before it begins its execution, as would happen if it were in complete control of the system.

Architecture - Apart from the data structure for register virtualization, each guest also has another data structure conceptually known as a VM control block (VMCB in AMD and VMCS in Intel). These data structures hold certain fields crucial to the execution of the VM, such as its instruction pointer and stack pointer. Along with that they also hold control bits that the VMM can set, which correspond to instructions/conditions that should be intercepted when the VM executes/encounters them, such as control register read/writes, tlb flushes, external interrupts, io port accesses etc. The other important fields are those that hold the reason for the VMEXIT, the exit codes for which can be interpreted to take the appropriate action, and event injection fields to be able to inject events to the VM such as hardware interrupts and software exceptions.

Fiasco provides an abstraction known as a VM task, which is used by the VMM to run the VM. The memory configuration of the VM is captured within this task. That is memory mapped into this task appears to the VM as its own. Fiasco has two main operations which play a major role in the memory mapping of tasks, the first is the 'map' operation and the second is the 'unmap' operation. The map operation allows a task to share a section of its virtual address space with another task, such that memory accesses by the descendants of the parent task, in the address space which has been shared with them, resolves to the appropriate physical address. The unmap operation reverses the effects of the map operation by revoking the rights of a descendant to the address space that was shared with it by the parent task. Since the VM uses the VMM's data-space for all its memory operations, the VMM must map its data-space into the VM task. By virtue of Fiasco's secure address space construction, the VMM can only pass on memory that it has access to, preventing misbehaving guests from reading/writing to memory beyond the VMM's dataspace. Similarly for device accesses, the VMM must map the appropriate address space allocated by the device servers into the VM task.

Hence, the VM task runs the VM by making a call to the micro-kernel and providing it with the VM control block and its duplicated registers. When a condition is encountered in the

execution of the VM for which an intercept is required, a VMEXIT takes place and a switch is made back to the VMM.

5. Nested Paging

The current version of Karma only has support to run guests on AMD machines, with the nested paging tables (NPT) feature enabled. Unfortunately, it cannot run guests on Intel with the extended page tables (EPT, the equivalent of NPT) feature since the Fiasco kernel itself does not have support for EPT yet. Nested paging is the mechanism by which the hardware handles memory management of the guest without intervention of the VMM. That is two levels of page tables are monitored, the first is the guest page table and the second is the host page table. The guest page table contains mappings from the guest virtual address (GVA) to the guest physical address (GPA). The host page table which is inaccessible to the guest, describes how a GPA is translated to the corresponding host virtual address (HVA). Hence, to access memory the hardware first traverses the guest page table to translate the GVA to the appropriate GPA (the page table pointer for the guest is accessible from its duplicated register state), and then the GPA is translated to the host physical address (HPA). Hence, all operations related to memory management such as the INVLPG instruction, control register writes and tracking of mapped pages do not need to be intercepted if nested paging is enabled.

If not, then a software implementation of of memory management must be done in the VMM, generally known as shadow page tables (SPT) or virtual tlb (VTLB). For SPT, the VMM must manually traverse the GPT in order to translate the GVA to the corresponding GPA which in turn resolves to the HVA in the VMM's dataspace, upon which the HVA is mapped to the GVA in the VM task. Hence, during the execution of the VM, the host provides the page table to be used, which is populated lazily by the above method on page faults.

In this case the aim was to implement SPT in Karma for both Intel and AMD, so that guests could be run on machines with hardware virtualization support but without NPT and EPT support. The different scenarios that arise during a software implementation of memory management are expounded upon below.

6. Shadow Page Table Memory Management

a) CR0 Tracking:

When the guest starts up, it does so in real mode (without paging), however since it is running in a VM we set the paging bit in the CR0 register to make it believe that it is in paged mode, in order to prevent direct memory accesses by the guest, so that it has to go through the paging mechanism of the VMM. We also intercept CR0 writes so that we can track when the guest actually enables paging and also reflect other changes made by the guest in the guest CR0 register.

b) CR3 Reloads:

A CR3 reload in the guest signifies the loading of a new guest page table pointer, which needs to be recorded by the VMM in order to traverse the guest page table later on. It implicitly also signifies the requirement of a tlb flush, so that all the prior memory mappings that were constructed from the guest virtual address space to the address space of the VMM, are unmapped.

c) Page Fault Handling:

When a guest tries to access a page not present in Fiasco's page tables, a VMEXIT is caused due to a page fault. The faulting linear address is made available to the VMM in a specific field. The VMM then traverses the guest page table to check if the corresponding page is present. If the page is present then the linear address is resolved to a guest physical address which is then resolved to a address in the VMM's dataspace (which is the base address of the dataspace plus the guest physical address). Using Fiasco's map operation a mapping is created between a page (either 4KB or 4MB in size) containing the faulting linear address to that of the page from the physical address space of the VMM, which Fiasco then tracks until that page is unmapped. If the corresponding page is not present in the guest page table then the VMM injects a page fault to the guest by which it supplies the faulting address and error code through an event injection field and resumes the VM. The page fault is then delivered to the guest through the Interrupt Descriptor Table (IDT) as if were running directly on the hardware, by which it takes the appropriate action to resolve the page fault by setting the page table entry. Once this is done, a VMEXIT will be caused immediately since a mapping is still not present in Fiasco's page table, yet now when the VMM traverses the guest page table, it will find the appropriate entry to resolve to a GPA.

d) Handling of Dirty Bits:

Each page in the guest page table (GPT), has a dirty bit associated with it which is set every time something is written to the page, so that the paging algorithms can write it back to disk when they are paged out if the need be, in which case the bit is then cleared. However, in the VMM we need to intercept when these writes happen so that the corresponding dirty bit can be set for the page in the GPT. Hence, the way we go about this is that when mapping a new page, if the access right of the page is 'write', we map it as read-only in Fiasco's page table so that when the guest tries to write to that page, we get a page fault with access right violation as the reason. We then check the GPT to see if the access right of the page was actually read-only, in which case it was a genuine page fault which needs to be injected to the guest. Else if the access write of the page in the GPT was write it means that we need to set its dirty bit and remap the page in Fiasco with 'write' access rights so that the guest can now write to the page without faulting.

e) TLB Flushes/INVLPG instruction:

Invalidation of page table table entries take place by primarily three mechanisms. The first is the INVLPG/INVLPGA instructions which are intercepted by the VMM and for which the linear address/ linear address, address space id, are provided in specific fields. The second is CR3 reloads which indicate the loading of a new page table pointer, implying a need to flush the tlb of stale mappings. The third is the clearing of the page global enabled bit in the CR4 register, which signifies that all global mappings must be flushed as they are not shared anymore. Hence, in these cases the VMM goes through the list of pages that it has mapped and using Fiasco's unmap operation removes the mapping of the corresponding sections of the guest's virtual address space from Karma's dataspace, which is reflected in Fiasco's memory management structures.

7. Multiprocessor Support

The Fiasco micro-kernel also has Symmetric Multi-Processing (SMP) support. This allows a user to request Karma for a certain number of virtual CPU's (vcpu's). As a result Karma allows the guest to execute in SMP mode by forking off a thread to be placed on each active CPU. Each thread or vcpu then has its own control block, set of general purpose registers and VM task which allows it to execute in parallel with the other CPU's. Inter-Processor Interrupts (IPI's), are injected in a similar manner to external interrupts and are handled in the guest via the IDT.

8. Implementation Differences between Intel and AMD

a) Multiple Event Injection Mechanism

One of the key implementation differences of hardware virtualization extensions in AMD and Intel is the mechanism by which the VMM can inject multiple events to the guest. These interrupts could be external interrupts such as device interrupts or exceptions such as page faults. In certain cases the guest is not ready to accept external events, which is the case if the interrupt flag (IF) bit of the RFLAGS register is cleared. Another scenario is if the guest is in an interrupt shadow, which is a phase during which the guest cannot accept interrupts even though the IF bit of the RFLAGS register is set to 1. In such scenarios multiple events might accumulate which need to be injected to the guest. However, since the guest handles only one event at a time after which it continues its normal execution, some mechanism is required by which a VMEXIT is caused when the guest is ready to accept new interrupts. In Intel this works by setting the interrupt-window-exiting bit in the control block of the VM, when interrupts are pending and the guest cannot accept them, so that a VMEXIT is caused as soon as the guest is ready to accept interrupts. This option handles both cases, namely the IF flags being set and the guest not being in an interrupt shadow. In AMD the mechanism differs a bit, although conceptually it is the same. If the guest can accept interrupts and only one interrupt is pending, then it is injected via the standard event injection mechanism, where a field is set with the interrupt type and number, so that on resuming the guest, before the execution of any instruction the interrupt is handled normally as through the IDT. Else if there are multiple interrupts pending then the first one is injected normally but the second one is marked as a virtual interrupt, with the virtual interrupt exiting bit set. This is done so that the guest immediately exits when it has handled the virtual interrupt and is ready to accept new ones. In AMD, the VMM must specifically check whether it can inject interrupts by confirming the following conditions: a) the guest is not in an interrupt shadow, b) the IF flag is 1 and c) the guest does not have its virtual interrupt pending bit set which implies that it still needs to handle the virtual interrupt that was passed to it.

9. Issues Encountered

a) AMD Instruction Analyzer

One of the main components that we had to incorporate into the VMM was an open-source instruction disassembler by the name of udis. This was because AMD does not provide the instruction length, and on certain intercepts the guest instruction pointer needs to be

incremented by the instruction length in order to make progress, as a result a disassembler was required.

b) AMD Qemu Bug

We encountered a bug in QEMU when trying to run the setup on it. When the guest tried to start up the initial user program, it constantly failed to load the shared libraries required for the operation. On further investigation, this was revealed to be a case of bogus syscalls being made by the user program. The events which led to the mentioned scenario played out as such: the user program made a syscall which was effectively completed, and the return value stored in the eax register. However before a switch could be made back to the user program from the kernel, the kernel assumed that the syscall had been interrupted and it needed to be re-issued so that it seems to the user as if the syscall only needed to be issued once. However, the sycall number in the eax register was already overwritten since the syscall had actually completed, as a result bogus syscalls were being made. A solution is still being worked upon and hopefully a patch will be submitted to the QEMU team soon enough.

c) Intel CR2 Register in Invalid State

Another issue that cropped up was that Intel does not explicitly set the CR2 register on VMEXIT's with the page fault address, leaving it up to the software to get the page fault address from a field in its control block. As a result the CR2 register must also be emulated as part of the guests general purpose register set and the host CR2 register should be set by Fiasco on resuming the VM, so that in the guest page fault handling, when it tries to get the page fault address from the physical CR2 register, it will not end up getting a bogus one.

10. Benchmarks

Benchmarks were run on AMD both with and without the vtlb implementation using the LMbench suite of tests. LMbench is standard set of tests meant to test certain aspects of the system such as memory, file-systems and networking on the basis of bandwidth and latency. A short description of each test is given below, the results of the tests with vtlb are presented, in cases where the results greatly differ without the vtlb, the results are also presented.

Latency Tests:

a) lat_syscall - times how long it takes to write one byte to /dev/null. This is useful as a lower bound cost on anything that has to interact with the operating system.

b) lat_pipe - uses two processes communicating through a Unix pipe to measure interprocess communication latencies. The benchmark passes a token back and forth between the two processes (this sort of benchmark is frequently referred to as a ``hot potato'' benchmark). No other work is done in the processes.

c) lat_proc - creates processes in three different forms, each more expensive than the last. The purposes is to measure the time that it takes to create a basic thread of control.

Options

Process fork+exit - The time it takes to split a process into two (nearly) identical copies and have one exit. This is how new processes are created but is not very useful since both processes are doing the same thing.

Process fork+execve - The time it takes to create a new process and have that new process run a new program. This is the inner loop of all shells (command interpreters).

Process fork+/bin/sh -c - The time it takes to create a new process and have that new process run a new program by asking the system shell to find that program and run it. This is how the C library interface called system is implemented. It is the most general and the most expensive.
---

d) lat_unix - The AF_UNIX sock stream latency

e) lat_fs - iss a program that creates a number of small files in the current working directory and then removes the files. Both the creation and removal of the files is timed.

f) lat_ops - measures the latency of basic CPU operations

g) lat_pagefault - times how fast a page of a file can be faulted in. The file is flushed from (local) memory by using the msync() interface with the invalidate flag set. The benchmark maps in the entire file and the access pages backwards using a stride of 256K kilobytes.

h) lat_sem - Measures semaphore latency

Bandwidth Tests:

a) bw_pipe - Creates a Unix pipe between two processes and moves 50MB through the pipe in 64KB chunks

b) bw_mem - allocates twice the specified amount of memory, zeros it, and then times the copying of the first half to the second half. Results are reported as megabytes moved, megabytes moved per second.

Options:
---
rd **-** measures the time to read data into the processor. It computes the sum of an array of integer values. It accesses every fourth word.

wr **-** measures the time to write data to memory. It assigns a constant value to each memory of an array of integer values. It accesses every fourth word.

rdwr - measures the time to read data into memory and then write data to the same memory

location. For each element in an array it adds the current value to a running sum before assigning a new (constant) value to the element. It accesses every fourth word.

cp - measures the time to copy data from one location to another. It does an array copy: dest[i] = source[i]. It accesses every fourth word.

bzero - measures how fast the system can bzero memory.
---

c) bw_unix - measures performance of data sockets

Results

Test	Shadow Page Tables	Nested Paging
bw_pipe	14.03 MB/sec	611.85 MB/sec
bw_mem (rd)	51.20 MB, 1129.12 MB/sec	
bw_mem (wr)	51.20 MB, 867.71 MB/sec	
bw_mem (rdwr)	51.20 MB, 709.61 MB/sec	
bw_mem (cp)	51.20 MB, 467.17 MB/sec	
bw_mem (bzero)	51.20 MB, 891.50 MB/sec	
bw_unix	7.31 MB/sec	756.51 MB/sec
lat_syscall (read)	0.3518 microseconds	
lat_syscall (write)	0.2987 microseconds	
lat_pipe	5267.1239 microseconds	10.2462 microseconds
lat_proc (fork+exit)	6931.0000 microseconds	78.7000 microseconds
lat_proc (fork+/bin/sh -c)	64181 microseconds	710.1250 microseconds
lat_unix	4377.000 microseconds	7.7433 microseconds
lat_fs (format for each row: size of the file, the number created, the creations per second, and the removals per second.)	0k 1482 261976 340613 1k 830 147818 53293 4k 747 131375 52728 10k 382 66527 202232	
lat_ops (ns - nanoseconds)	int bit: 0.8 ns int add: 0.8 ns int mul: 0.24 ns int div: 39.84 ns	

	int mod: 23.89 ns int64 bit: 0.81 ns uint64 add: 0.88 ns int64 mul: 0.84 ns int64 div: 74.85 ns int64 mod: 79.07 ns float add: 3.19 ns float mul: 3.19ns float div: 19.18 ns double add: 3.19 ns double mul: 3.18 ns double div: 19.17 ns float bogomflops: 18.41 ns double bogomflops: 18.34 ns	
lat_pagefault	10.8204 microseconds	1.0158 microseconds
lat_sem	1595.5145 microseconds	3.2946 microseconds

11. Conclusion

The aim of enabling Karma to run a commodity operating system such as Linux, using Intel and AMD's hardware virtualization techniques, without EPT/NPT support, was achieved by means of a shadow page table implementation. However, one of the drawbacks of the implementation being in the VMM itself, is that costly switches are required between the kernel and user-space for a majority of memory-management related operations. For example during page faults, a VMEXIT leads to a switch from the kernel to the VMM, which needs to make a switch back to the kernel when the 'map' operation is called. As a result some of the memory stress tests when run with the SPT functionality show slowdowns in the range of 10 to 100 times. To cut down on the number of context switches, future work could focus on integrating the shadow page table functionality directly within the micro-kernel, since it already has access to the same data structures as the VMM and can create the required memory mappings without the overhead of excessive context switches. Even though this approach will still not be as fast as running it with EPT/NPT, it will hopefully lead to a significant improvement over the current SPT implementation, while contributing marginally to the TCB of the system.

12. References

a) AMD Reference Manual Volume 2
b) Intel Reference Manual Volume 3a/b
c) LMbench manual pages. www.bitmover.com/lmbench/

13. Acknowledgements

I would like to thank Professor Jean-Pierre Seifert for placing me in the operating systems

group and giving me the opportunity to work on this project, which helped me learn an immense amount about micro-kernels and virtualization technology. Steffen Liebergeld, for his implementation of the Karma VMM and the paravirtualized Linux guest, as well as his advice on implementing shadow page tables. Matthias Lange, for his help in setting up Fiasco and Karma on both the Intel and AMD environments, as well as his initial explanations of micro-kernels and virtualization technology. Finally Michael Peter, for his countless hours of implementation/ debugging/writing help and knowledge on everything from operating systems to German history, without whom the completion of this project would not have been possible.