Porting Fiasco to AMD64

Torsten Frenzel, TU Dresden

December 9, 2005

Contents

1 Introduction **7**
- 1.1 About this work . . . 7
- 1.2 About this document . . . 7

2 Fundamentals and Related Work **9**
- 2.1 Operating Systems and Microkernels . . . 9
- 2.2 AMD64 Architecture . . . 9
  - 2.2.1 Operating Modes and Transitions . . . 9
  - 2.2.2 Register and Instruction Set . . . 10
  - 2.2.3 Memory Management . . . 11
  - 2.2.4 Task Management . . . 12
  - 2.2.5 Interrupts and Exceptions . . . 13

3 Design **15**
- 3.1 Context Management . . . 15
  - 3.1.1 Context State . . . 15
  - 3.1.2 Context Switch . . . 15
- 3.2 Memory Management . . . 16
  - 3.2.1 Kernel Memory Management . . . 16
  - 3.2.2 User Memory Management . . . 22
- 3.3 Kernel Entry/Exit . . . 22
  - 3.3.1 Interrupts and Exceptions . . . 23
  - 3.3.2 System calls . . . 23
- 3.4 Boot up and Initialization . . . 23
- 3.5 Kernel Debugger . . . 25
- 3.6 User-level Programs . . . 26
- 3.7 Summary . . . 27

4 Implementation **28**
- 4.1 Context Management . . . 28
  - 4.1.1 Context State . . . 28
  - 4.1.2 Context Switch . . . 28
- 4.2 Memory Management . . . 29
  - 4.2.1 Kernel Memory Management . . . 29
  - 4.2.2 User Memory Management . . . 30

4.3 Kernel Entry/Exit 30
4.3.1 Exception and Interrupts 30
4.3.2 Syscalls 31
4.4 Bootup and Initialization 31
4.5 Summary 32

5 Measurements and Evaluation 33
5.1 Performance for short IPC 33
5.2 Performance for Long IPC 34
5.3 Summary 40

6 Conclusions, Open Topics and Future Work 42

A Interrupt handling 44

B System call conventions 46

C Compiler calling convention 49

D Kernel address space 50

E Glossary 51

Bibliography 52

List of Tables

3.1 Page table consumption of small application scenario 17
3.2 Page table consumption of big application scenario 18
3.3 Comparison of the size of mapping database objects 18

4.1 Measured TCB sizes of a 'hello world' application 28

5.1 short IPC within one address space with no shortcut 33
5.2 short IPC within one address space with C-shortcut 34
5.3 short IPC between two address spaces with C-shortcut 34
5.4 short IPC between two address spaces with no shortcut 34

A.1 Handling of interrupts in Fiasco 45

List of Figures

2.1 Operating Modes (source: [11]) 10
2.2 Segment registers in 64-bit mode (source: [11]) 12
2.3 Page translation in long mode (source: [11]) 12
2.4 TSS format in long mode (source: [11]) 13
2.5 Interrupt gate descriptor in long mode (source: [11]) 13
2.6 Interrupt-handler stack in long mode (source: [11]) 14

3.1 Shared page tables in Fiasco/IA32 17
3.2 Shared page table 20
3.3 Shared page directory 21
3.4 Shared page directory pointer 22
3.5 bootup sequences 25

5.1 Comparison of direct long IPC between Fiasco/IA32 and Fiasco/AMD64 with warm caches 35
5.2 Comparison of direct long IPC between Fiasco/IA32 and Fiasco/AMD64 with cold caches 36
5.3 Comparison of direct long IPC between Fiasco/AMD64 variants 37
5.4 Comparison of indirect IPC with 1 string between Fiasco/IA32 and Fiasco/AMD64 38
5.5 Comparsion of indirect IPC with 4 strings between Fiasco/IA32 and Fiasco/AMD64 38
5.6 Comparison of indirect IPC between Fiasco/IA32 and Fiasco/AMD64 with 1 string and cold cache 39
5.7 Comparison of indirect IPC between Fiasco/IA32 and Fiasco/AMD64 with 4 strings and cold cache 39
5.8 Comparison of indirect IPC between the Fiasco/AMD64 variants with 1 string and warm cache 40
5.9 Comparison of indirect IPC between the Fiasco/AMD64 variants with 4 strings and warm cache 40

D.1 Shared page table 50

1 Introduction

Since the first days of the x86 architecture designed by Intel several extensions have been made to improve emerging shortcomings. The last years have shown that 32-bit address spaces are no longer sufficient for some application's needs. The 64 bit extension AMD64 proposed by AMD targets this problem and solves it while trying to achieve a balanced compromise with respect to flexibility and compatibility.

Fiasco, a microkernel with realtime capabilities, developed at the TU Dresden, is the basis for the Dresden Realtime Operating System[5]. Fiasco is already ported to the 64-bit IA64 architecture[12]. The kernel has no support for the 64-bit AMD64 architecture. But as a basis for future research it seems necessary to follow this direction of hardware development.

1.1 About this work

The main goal of this work is to improve the Fiasco microkernel to run in a 64-bit environment. The basis for this port was the Fiasco implementation for the x86 architecture with the L4-v2 ABI, shortly referred to as Fiasco/IA32.

Besides functional correctness and stability for the AMD64 architecture as the obvious goals of the port, there are some more desired results. The 64-bit implementation should show at least the same performance as the 32-bit one. Also a firm integration of the new code into the main Fiasco build tree seems reasonable, because of the extensional character of the AMD64 architecture.

Fiasco as a mature and fully configurable microkernel has many extensions which can not be considered in the scope of this work. We will only deal with the fundamental system: inter-process communication (IPC) without further extensions (like small address spaces (SMAS)[6], IO/Protection[1] and the assembler IPC shortcut).

In addition to this main work of porting the kernel, there are other things to be done: Implementing a sufficient debugging environment and leveraging some user level programs to 64-bit for testing purposes. Altough this adaption has consumed a significant amount of time, it's not part of this document.

1.2 About this document

In the following chapter we discuss the new set of features introduced by the AMD64 architecture. Whereupon we focus only on points relevant for this port. After knowing

[1]an access rights enforcement for IO-ports

the hardware given prequisites we can deduce in chapter 3 a design for parts were changes might occur. We will show, that most work is done around the paging and bootstrap code related to paging. Chapter 4 deals with some implementation issues. We show the relevant changes in the code basis and which parts of the code required no changes. In chapter 5 we derive from the implementation some performance estimates. Therefore we tested Fiasco with the quasi standard performance tool for IPC 'pingpong' and spot on some special code fragments to get more information about execution times. In the last chapter we recall what we have achieved in this work. We also try to derive some conclusions and give a short outlook for later developments.

2 Fundamentals and Related Work

2.1 Operating Systems and Microkernels

The development of microkernels is one direction in the development of operating systems. Contrary to monolithic kernels like Linux or Windows it follows the concept of minimality. The microkernel paradigm proves to be the kernel as the smallest set of features and builds a running full featured operating system on top of it. This minimal set comprises an address space abstraction, threads as executional units and some sort of IPC mechanism.

Early microkernels failed due to performance penalties and lead to a new and considerably faster generation of microkernels. Fiasco as a member of this second generation microkernel family was first implemented with the L4v2 API [10]. This API was tailored to the x86 hardware architecture[7]. New APIs were designed to become more hardware independent, like L4-x2[3] and L4.sec[9].

At least I decided to use the L4v2 API as basis for this port because of the high level of maturity and stability of the used code and the available support of user level programs.

2.2 AMD64 Architecture

This section describes a set of new features of the AMD64 architecture related to the IA32 architecture. The content focuses mainly on differences between the two architectures. We assume that the reader is already familiar with the IA32 architecture.

2.2.1 Operating Modes and Transitions

In the AMD64 architecture two new operating modes are defined. Beside the legacy modes known from the IA32 architecture, AMD64 introduces the new long mode consisting of two submodes called compatibility mode and 64-bit mode (fig. 2.1). As the new name suggests the compatibility mode allows to run 32-bit programs under control of a 64-bit environment. In 64-bit mode all new features described in the following sections are available.

The transition from 32-bit mode to 64-bit follows the way over compatibility mode. Compatibility mode is enabled with the following steps:

1. Enabling physical-address extension by setting CR4.PAE[7] to 1.

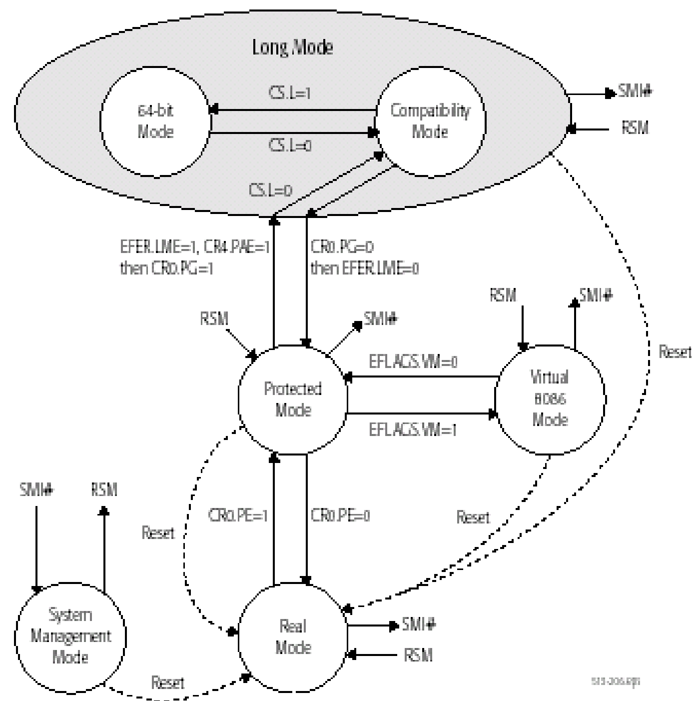

Figure 2.1: Operating Modes (source: [11])

2. Loading register CR3 with the physical base-address of the level-4 page-map-table[1] (PML4).
3. Setting the long-mode enable control bit EFER.LMEintel:ia32-v2 to 1.
4. Enable paging by setting CR0.PG to 1.

The processor is now running in the compatibility submode. Loading now a new 64-bit code segment with the L-bit[2] set, switches from compatibility submode to 64-bit submode.

2.2.2 Register and Instruction Set

In 64-bit mode all legacy registers are expanded to 64 bit and eight new general purpose registers named R8 to R15 are added.

The following list gives an overview of the changes in the instruction set:

[1] see section 2.2.3
[2] see section 2.2.3

operand size In 64-bit mode the default operand size is 32 bit. Exceptions are near branches and all instructions that reference the RSP (except far branches).

address size In 64-bit mode the default address size is 64 bit. The address size can be overridden by using the address-size prefix.

REX-prefixes These prefixes are a new set of instruction prefixes only valid in 64-bit mode. They are used for example to specify additional GPRs, a 64-bit operand size and additional debug registers.

RIP-relative addressing A new instruction-pointer relative addressing is implemented in 64-bit mode.

RSP-implicit addressing The stack pointer must be aligned to 64 bit and only pushes and pops of 64-bit values are allowed. The instructions `PUSHA` and `POPA` for saving all GPRs are not allowed in 64-bit mode.

Fast system call and return The instruction pair for fast control transfer, `SYSENTER` and `SYSEXIT`, is not allowed to be used in 64-bit mode. Instead the instructions `SYSCALL` and `SYSRET` has to be used.

2.2.3 Memory Management

While the segmentation mechanism is abandoned in long mode the paging mechanism is expanded to support 64-bit wide address spaces.

Segmentation

Segmentation was introduced with the 80286 processor to provide a way of isolation. However this mechanism was hardly used as the page-based protection mechanism become the prevalent. There still exists a global descriptor table (GDT) with segment descriptors but segmentation is more or less obsolete in 64-bit mode. This means the flat memory model is the only one supported. When loading a new segment descriptor the hidden parts of the segment registers are filled with default values. Segment limits are never checked and except for FS and GS the segment base is ignored for address calculation (fig. 2.2).

The architecture defines a new attribute for code segments. When loading a code segment with the L-bit attribute set to one the processor switches immediately to 64 bit mode, assuming it was in compatibility mode before [3].

Paging

As mentioned before the AMD64 architecture extends the legacy paging mechanism. The page translation hierarchy consists of four levels to support a 64-bit address space (fig. 2.3).

[3] see section 2.2.1

CS (Attributes only)
DS (ignored)
ES (ignored)
FS (Base only)
GS (Base only)
SS (ignored)

Figure 2.2: Segment registers in 64-bit mode (source: [11])

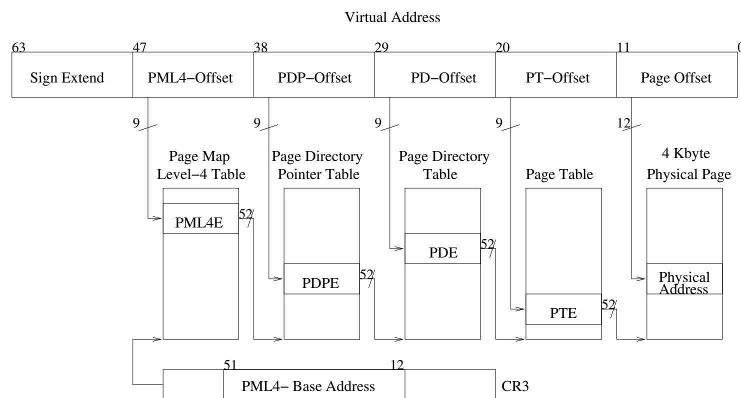

Figure 2.3: Page translation in long mode (source: [11])

The level-4 page-map-table (PML4) is the root of the paging hierarchy. PML4 entries point to a page-directory-pointer table (PDP), which in turn points to page directories.

A virtual address is now divided into five parts. The page offset is 12 bit wide to support 4-Kbyte size pages as in legacy mode. The following 9 bits are grouped as indexes in the respective page tables. All page tables have 64-bit wide entries and therefore hold 512 entries.

Beside 4-Kbyte pages long mode supports also 2-Mbyte superpages, 4-Mbyte superpages are not available.

Currently only 48-bit wide in virtual addresses are significant. Virtual address must be in canonical form, meaning non-significant bits must be either set all to zero or to one. The consequence is that a the address space is splitted up in a low and a high part.

2.2.4 Task Management

The legacy hardware task-switch mechanism is disabled in long mode. However, the system software must create a single task state segment.

A 64-bit TSS is defined for use in long mode. This new TSS format contains 64-bit stack pointers for all privileges, the interrupt stack table[4] (IST) pointers and the

[4]see section 2.2.5

I/O-map base address.

63		0	
I/O Permission Bitmap (8K)	I/O−Map Base	Reserved, IGN	+64h
Reserved, IGN			+5Ch
IST7			+54h
IST6			+4Ch
IST5			+44h
IST4			+3Ch
IST3			+34h
IST2			+2Ch
IST1			+24h
Reserved, IGN			+1Ch
RSP2			+14h
RSP1			+0Ch
RSP0			+04h
Reserved, IGN			−04h

Figure 2.4: TSS format in long mode (source: [11])

Software cannot use task gates and registers are not saved in the TSS (fig. 2.4).

2.2.5 Interrupts and Exceptions

The AMD64 architecture also expands the well known legacy interrupt-processing mechanism to support handling of interrupts in 64-bit mode.

	16	15	14 13	12	11 8	7 3	2 0	
Reserved, IGN								+12
Target Offset 63−32								+8
Target Offset 31−16		P	DPL	S	Type	Reserved, IGN	IST	+4
Target Selector	Target Offset 15−0							+0

Figure 2.5: Interrupt gate descriptor in long mode (source: [11])

When the processor is running in long mode, an interrupt or exception causes the processor to enter 64-bit mode, so all long-mode interrupt handlers must be 64-bit code.

The long-mode interrupt descriptor table IDT must contain 64-bit mode interrupt gates or trap-gate descriptors. As stated in the previous section task gates are not supported in long mode. A long mode IDT entry is expanded to 128 bits and contains a new field called IST (fig. 2.5), which references a stack pointer in the TSS.

The IST is a list of seven 64-bit addresses. These addresses point to memory that can be used as interrupt-handler stack. An interrupt-gate descriptor which contains a non-zero value in its IST field will load the RSP with the corresponding pointer indexed in the IST area of the TSS. This mechanism is a used in some situations where a valid stack pointer is not available like in double faults.

The size of interrupt stack-frame pushes is fixed at 64 bit. Long mode interrupts cause the stack pointer and stack segment to be pushed unconditionally, not only on a CPL change (fig. 2.6).

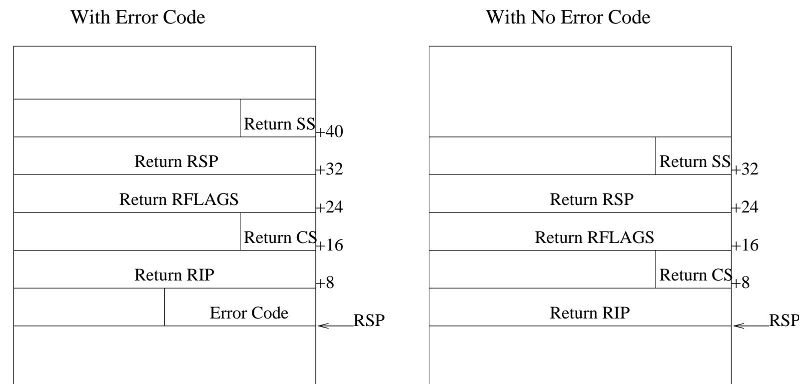

Figure 2.6: Interrupt-handler stack in long mode (source: [11])

3 Design

This chapter discusses the major changes for porting Fiasco. Because we use Fiasco/IA32 as the basis for the port, only hardware dependent aspects of the kernel are relevant.

Most changes will occur around the paging and CPU setup-code. Other parts in the kernel like scheduling, synchronization and high-level IPC-implementation need not to be modified.

3.1 Context Management

The Fiasco microkernel provides an abstraction for user-level execution contexts, called threads. Many of them may execute in the same address space. Address spaces provide a protection domain, so that threads in different address spaces are protected from each other.

3.1.1 Context State

The context state of a thread is maintained in a data structure, called thread control block (TCB). The TCB contains two parts of information, the thread state and the stack for kernel execution. The stack therefore is collocated to the TCB. The arrangement of this structure has the advantage of fast calculation of the TCB address from the kernel stack pointer, but the disadvantage of a potential collision between thread state and stack.

In Fiasco/IA32 the TCB size is fixed to 2 KByte. A TCB is mapped in the TCB area of the kernel address space. Every TCB has a unique address calculated by the id of the thread. The physical memory for them is allocated on demand on thread creation and triggered by a kernel page fault in the TCB area on the first write operation.

The thread state holds no hardware dependent information, all registers are saved on the stack, and therfore no changes are necessary.

3.1.2 Context Switch

If the kernel provides execution contexts, it also has to switch among them, for example to provide time sliced execution even on one CPU or to transfer control on IPC operations. At a context switch the state of the current thread must be preserved and the state of the next context to run must be restored, from their respective TCB.

A context switch is divided into a thread switch and a address space switch. A thread switch saves the register and stack pointer, loads the new register and stack pointer and resumes the execution at the restart point. An address space switch is done via reloading

the appropriate control register. It can be left out if source and target address space are identical.

Both parts are very architecture dependent. But due to the backward-compatibility and good high-level abstraction of code the porting effort is low. The changes comprise only to the expanded register set as part of the user program execution context. The address space switching mechanism needs not to be changed.

3.2 Memory Management

A kernel has to deal with user-level and kernel memory. The Fiasco/IA32 microkernel only provides mechanisms to implement user memory management safely outside the kernel. It sends user-level page faults as IPC to the pager.

3.2.1 Kernel Memory Management

The management of physical memory for the kernel is based on the amount of total physical memory available. Fiasco reserves at boot time usually 10% but not more than 64 MByte. This memory is used for all kernel and hardware data structures and can't be adapted dynamically to the application needs.

There are various aspects of the AMD64 architecture that influence memory consumption. The expanded machine word size, the 64-bit wide address space, the expanded register set and 4-level page tables contribute to the data structure growth. This might lead to an early kernel memory exhaustion. The next section therefore discusses relevant kernel objects and their impact on kernel resource usage. The goal is to figure out which data structures are most critical in this sense.

The easiest solution is to increase the size of the preallocated memory. But shrinking the size of available user-level memory seems not optimal, considering systems with few memory. So we have to search for possible reductions of the used kernel memory. For this reason a detailed analysis of memory usage scenarios seems worthwhile.

Kernel memory objects

The following list gives an overview, what kernel memory is allocated for and which impact can be expected in a 64-bit architecture.

1. Page tables

   Page tables are allocated per address space. Page table handling is different for the kernel and the user address space. The kernel address space is identical for all tasks. This means all page directory entries of the kernel address space reference the same page tables (fig 3.1).

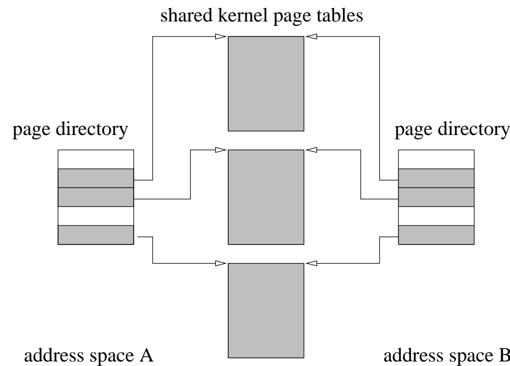

Figure 3.1: Shared page tables in Fiasco/IA32

Therefore in Fiasco/IA32 we need only a page directory for the kernel address space(, which of course also holds the entries to the user address space).

In two examples we try to determine the number of used page tables. In a first scenario we consider a user address space of a small application with two page tables at the lowest level, one for the code/data section and another for the stack section (table 3.1)

Because of the four-level paging hierarchy in Fiasco/AMD64 two more page tables int the mid level have to be allocated for the kernel address space and for the user address space. We allocate for both parts of the address space an intermediate PDP and page directory. So in this scenario of a small application we need more than twice the number of page tables for Fiasco/AMD64 than for Fiasco/IA32.

page tables	Fiasco/IA32	Fiasco/AMD64
kernel	1	3
user	2	4
total	3	7

Table 3.1: Page table consumption of small application scenario

In a second scenario the number of allocated page tables of the application grows up to ten (table 3.2). Now the ratio between Fiasco/IA32 and Fiasco/AM64 has improved. At least the absolute difference of four page tables is constant. This holds true as long as the address space of the application is limited to 1 GByte[1]. Further we can conclude that the additional number of allocated page tables is application dependent. The ratio between Fiasco/IA32 and Fiasco/AMD64 can be further improved, if superpages are used.

[1] after 1 GByte we have at least to allocate a new page directory in Fiasco/AMD64

page tables	Fiasco/IA32	Fiasco/AMD64
kernel	1	3
user	10	12
total	11	15

Table 3.2: Page table consumption of big application scenario

Some page table optimizations will be discussed below.

2. Context state (TCB)

   A TCB is allocated for every thread. There are various aspects that influence the size of the TCB. The most important one is the stack usage. But also the doubled machine word size, the enlarged register and the function calling convention used by the compiler are relevant. In theory, one could calculate the worst case stack depth. In practice, however, it is hard to determine because it depends on the deepest execution path in the kernel. Because Fiasco is fully preemptible, several execution paths can run interleaved on the same stack which complicates thid calculations. But in practice specific measurements can also show how big the size of the stack grows[2].

   The easiest way to avoid a stack overflow into the thread state part of the TCB is to expand the size of the TCB. Fiasco assumes the TCB always aligned to page boundaries and the size as a power of 2. So the stack size can only be doubled. This has direct impact on memory usage and address space layout. Unfortunately no easy approach for reduction can be seen, as every TCB holds unique information and is allocated on demand.

3. Mapping database objects

   The mapping database tracks the relationships between mapped pages. It is used for recursive revocation. The size of the mapping database depends on the amount of physical memory available and the mapping relationships of the running applications.

   The mapping database contains trees which hold mappings of virtual addresses in compressed form. All trees are arranged in an array indexed by physical addresses.

[2]see Implementation section

Object	Fiasco/IA32	Fiasco/AMD64
array element	8	16
tree with 4 elements	24	40
tree with 16 elements	84	144

Table 3.3: Comparison of the size of mapping database objects

From table 3.3 it can be deduced that the memory used for the whole mapping database nearly doubles in size. There is no obvious solution for a compression of mapping database objects.

4. Other data structures

   There are more memory objects in the kernel, but they are not important to this discussion, because thay have a neglible impact on memory consumption.

   Processor data structures, like GDT, TSS and IDT, are allocated once per processor. The size of this structures is processor defined and fixed.

   Slab caches hold empty objects of fixed size for various purposes. They are used for the mapping database and the kernel debugger objects. The size of such slabs might grow as the number of empty objects grows.

   The kernel debugger itself allocates memory for various objects. The trace buffer contains sets of trace entries. This buffer has a predefined size and memory is allocated in static initialization code. As the size of the objects in the trace buffer grows, their number decreases. So if not explicitly expanded, the size of the trace buffer is constant and has no impact on kernel memory usage. But as the kernel debugger is not a functional component it needs not to be considered.

We can conclude that TCBs, page tables and the mapping database are the most relevant objects, in terms of kernel memory usage. But only page tables leave reasonable room for optimizations, which will now be discussed.

Page table optimization

As explained above Fiasco uses shared page tables for the kernel address space. This has the advantage that kernel page tables are only allocated once for all address spaces. The master page table holds the current version of the kernel address space. All other task must synchronize with the master address space.

There are three cases where the kernel address space between tasks might temporarily differ:

1. As stated out before TCBs are allocated on demand when a thread is created actually when it is accessed the first time. If there is only a read access for an previously unaccessed TCB (for example to check the existence) a zeroed page is mapped read only. Therefore the TCB area is synchronized at least before every system call.

2. Fiasco/IA32 uses an IPC-window to map a range of the user address space in the kernel address space when transferring data between threads. The IPC-window is mapped on long IPC system call of a thread in the sender's kernel address space. This requires only one copy operation instead of two.

3. The IO-bitmap is a hardware defined data structure and used to give tasks dedicated access to IO-ports. So address spaces using this feature, have mapped different pages on the address range for the IO-bitmap.

The following three approaches improve the sharing for four-level page tables and discuss the advantages and disadvantages in detail.

1. Approach A: Sharing kernel page tables

   At a first try we can use the sharing mechanism of Fiasco/IA32 for Fiasco/AMD64 as shown in figure 3.2. With this approach we have to allocate two more pages for a PDP and a page directory for every task, only to map the kernel address space. And even worse, only one entry in these page tables is used, which means wasting lots of memory.

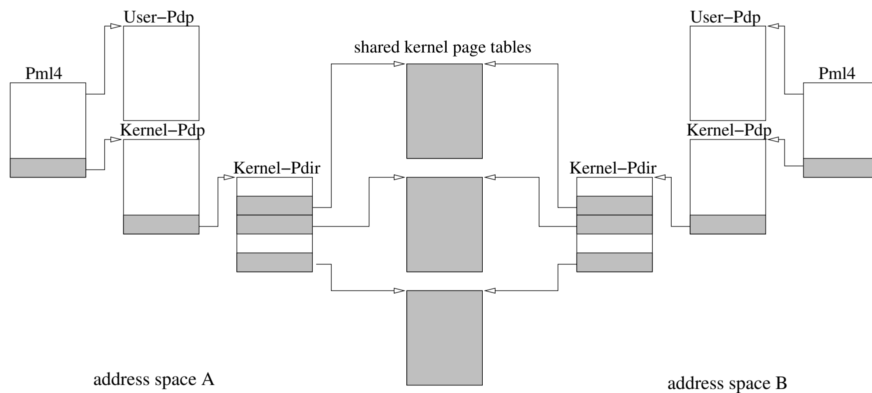

Figure 3.2: Shared page table

2. Approach B: Sharing the kernel page directory

   A reasonable approach is also to share the page directory (fig. 3.3). With this solution we save one page table, to map the kernel address space and reduce the synchronization overhead. The kernel address space extends to 1 GByte.

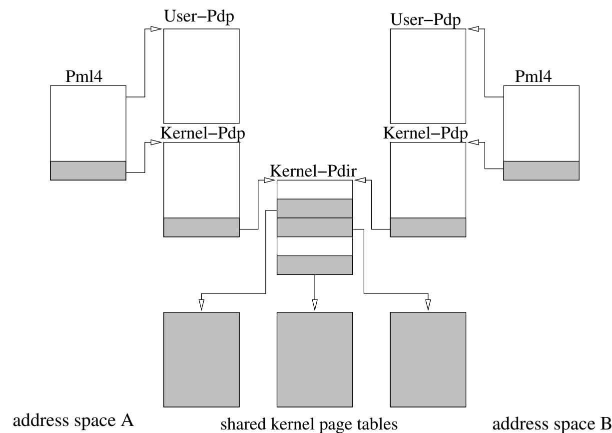

Figure 3.3: Shared page directory

3. Approach C: Sharing the kernel page directory pointer

   Going one step further than in approach B we can also share the kernel PDP. This means we reserve one PML4 entry for the shared kernel address space (fig. 3.4). But with this approach Fiasco/AMD64 comes close to the number of allocated page tables for kernel memory in Fiasco/IA32. The kernel address space is now 5Byte in size.

   For the TCB area there is no need of synchronization anymore, All address spaces see updates immediatly. The IPC-window can be handled as before and mapped in at page directory level. But for the IO-bitmap we have to reserve a new PML4 entry as it is an area that is not shared and not updated on context switch. This means setting-up a new page table hierarchy for this resource. But it might be a feasible approach, as such cases are rare. (Currently the IO-bitmap is only used for L4-Linux ([2]).

Deciding between approach A and approach C means to make the tradeoff between wasting memory but retain simplicity of the shared model and save kernel memory but add new complexity when the shared model breaks. The most effective solution depends on the application scenario.

Regarding memory as a limited resource which is very worth saving, approach C should be favored for the implementation.

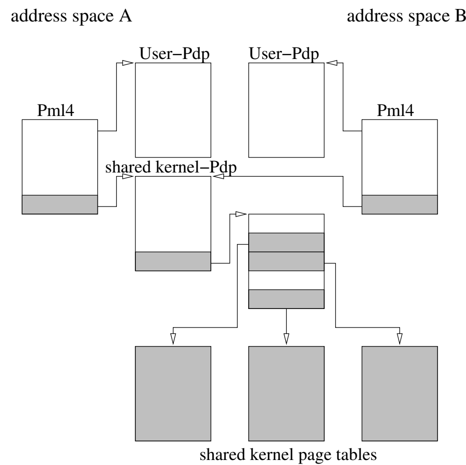

Figure 3.4: Shared page directory pointer

IPC-window optimzation

Knowing that the IPC-window contradicts the shared model, one can handle it seperatly from the rest of the kernel address space. There exists similar optimzation approaches as discussed for the page table optimization. Instead mapping the IPC window at page directory level one can map the IPC-window at PDP-level or even PML4-level. The later means that a whole PML4 slot is used, in which a part of the user space is mapped for long IPC operation.

The assumption with this approach is to avoid traversing the page tables and speed up the whole long IPC operation.

3.2.2 User Memory Management

The microkernel paradigm places the policy for page fault handling outside the kernel. When the kernel triggers a page fault it sends an IPC to the pager of the faulting thread. The pager replies with a mapping that resolves the fault. Therefore the kernel provides a secure primitive that allows the management of memory from user space.

Fiasco as an L4 implementation supports the model of recursive construction of address spaces [10]. The kernel has to handle a mapping database, which reflects hierarchical relationships between mapped pages.

The functions for handling user-level page tables are well encapsulated. They can be considered architecture dependent and must be reimplemented.

3.3 Kernel Entry/Exit

A kernel entry is a control transfer from user to kernel mode. Typical situations are system calls to request kernel services, exceptions or external events. A kernel entry is in Fiasco/IA32 implemented with the interrupt-gate mechanism, except for one special

system call entry. Therefore we discuss the interrupt handling first and examine the system call handling later.

3.3.1 Interrupts and Exceptions

Interrupts and exceptions are used in Fiasco for various purposes. Most interrupts and exception handlers create a trap frame before entering the kernel. This frame saves the register state plus the possible page fault address and is used to restore the user state on kernel exit. A fragmentary list of the Fiasco/IA32 IDT is given in table A.1.

When entering a kernel the question of saving the register state of the user level execution context arises. If the register state must be accessible because it is used to transfer data or should be dumped to screen than the handler must create a trap frame. But if the register state must only be restored upon execution resumption than saving it can be handled by the compiler in conjunction with a calling convention. The second case is faster and is applied to the page fault handler and the timer interrupt handler.

One porting problem relates to the double fault handler. A double fault means an interrupt entry failed, because for example the stack pointer is not valid. In Fiasco/IA32 this case should normally never happen but due to the development of the system it cannot totally be excluded. The purpose of the handler is to prevent the system from rebooting that the error can be investigated. In Fiasco/IA32 it uses a trap gate to set up a clean stack pointer. This mechanism is not possible in 64-bit mode, as trap gates are not available. Instead we have to use the IST mechanism to set up the stack pointer.

3.3.2 System calls

A system call enters the kernel via an interrupt gate. The entry-frame saves the whole register state, and contains parameters used for the system call. On kernel exit the thread state is restored and return values are passed back to the user-level thread. The entry-frame is saved on the thread's kernel stack.

Fiasco/IA32 saves all GPRs and in some cases the segment registers. In Fiasco/AMD64 there is no need to save segment registers, as segmentation is not supported in 64-bit mode. The register set is expanded for the registers R8 through R15.

Every system call has a convention on the semantics of register contents upon kernel entry and exit, called binding. It describes which registers are used for which parameters. The implementation of the bindings is therefore architecture dependent. In the appendix is list a list of the bindings for the seven Fiasco system calls.

3.4 Boot up and Initialization

The purpose of the boot code and startup code is to set up a well defined runtime environment for the kernel. This means initializing the CPU and the hardware data structures.

From our point of view, during bootup we have to decide where to switch to 64-bit mode. For this reason we first analyze the bootup path of Fiasco/IA32.

The following list describes the Fiasco/IA32 boot-up scenario:

1. Boot loader (GRUB)

   GRUB is a GPL licensed boot loader [1] with many advanced features, like reading configuration scripts, loading modules from networks and honoring the multiboot standard [4]. GRUB switches the processor to 32-bit protected mode and sets up a one-to-one mapping of the whole physical memory. It then loads a configured set of modules from storage or network and tranfers control to the first module.

2. Bootstrap module

   The modules are loaded in a binary format, called executable and linkage format (ELF), and need to be transformed into their runtime representation. The bootstrap module extracts subsequent modules loaded by GRUB, namely the kernel, sigma0 and the roottask binary. It then passes control to the kernel boot code entry-point.

3. Kernel boot code

   The first piece of kernel code is responsible to set up all hardware data structures as required by the kernel. It also initializes the CPU and address space mappings. At this point the kernel has a well-defined environment.

4. Kernel initialization code

   Although the kernel runs in a fully initialized environment it sets up hardware data structures again. There are various reasons for this decision. The kernel code recycles the kernel boot code after the initialization phase and the kernel has to repoint IDT vectors to new handler code. After this various kernel parts are staticly initialized, the scheduling starts and the first tasks sigma0 and roottask are created and started.

Considering this boot sequence, there are 4 possible approaches to switch from 32-bit mode to 64-bit mode. The following points will discuss each variant in detail:

1. Approach A: Switch in bootloader

   This seems the approach with least complexity. The bootstrap module and the kernel code are 64-bit code. We don't have to mix 32-bit and 64-bit code there. But unfortunately GRUB is not capable of initializing a 64-bit environment. Some Work-around solution for this problem might be to patch and enhance GRUB. But the drawback of this approach is to maintain this patched GRUB for future versions.

2. Approach B: Switch in bootstrap

   This second approach, has also the advantage of approach A: There is only 64-bit code in the kernel. The drawback is that, we have to deal at 3 points with the CPU initialization. Beside initializing the CPU in the kernel boot code and the kernel initialization code, also bootstrap has to be capable to deal with it.

3. Approach C: Switch in kernel boot code

   It seems the approach with the lowest overhead in terms of CPU initialization. Bootstrap loads Fiasco/AMD64 as a 32-bit binary. It passes control to 32-bit kernel boot code, which in turn extracts the 64-bit kernel image and sets up the 64-bit environment. But there are problems to integrate this solution into the current kernel. Because of the microkernel approach Fiasco doesn't handle binary images, like ELF. Integrating this functionality, might blow up the kernel image and is not well substantiated into the Fiasco build system.

4. Approach D: Switch in kernel initialization code

   Switch to 64-bit mode in kernel code is too late. As we see from the previous analysis the kernel needs an already a set-up environment to run. So this approach has to be rejected.

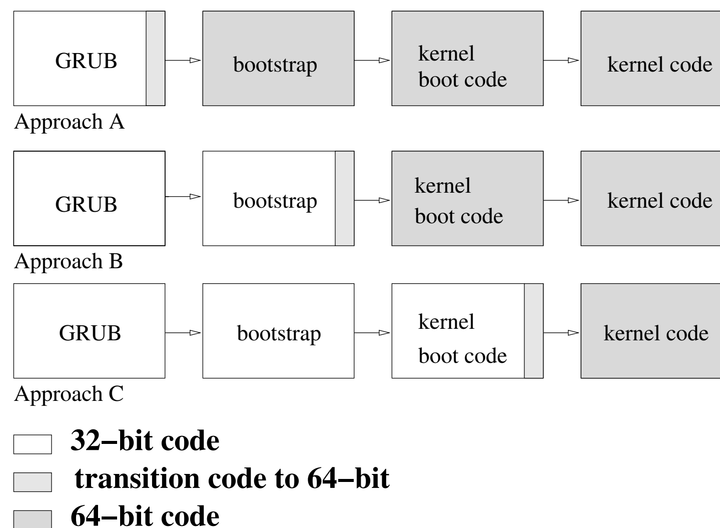

Figure 3.5: bootup sequences

The previous discussion leads to a decision between the approaches A, B and C. From the maintenance overhead approach A is not feasible and approach C has big impact into the Fiasco build system. So the most reasonable solution seems B, hoping that code duplication for CPU initialization will be limited.

3.5 Kernel Debugger

The built-in kernel debugger has become a powerful tool for developing the kernel and user-level programs. The debugger provides hardware-independent information about the state of threads, queues and hardware-dependent information about the underlying hardware. So it is capable of browsing page tables, dumping memory and disassembling

code. The tracing facility stores information about IPCs and other system calls in the tracing buffer, where it can be examined. The user can set breakpoints and single-step through programs. (For a full list of features refer to [8].)

The kernel-debugger is normally entered via a breakpoint exception. It can be used interactively for browsing or via batch-like mode to manipulate the configuration.

Porting the kernel debugger is essential for debugging and testing. In this work only the most important parts, such as showing page tables, are ported.

3.6 User-level Programs

To provide a basic functionality for testing the ported kernel a set of user-level programs and libraries have also to be ported. The most important ones are described in the following list.

1. Sigma0 - root pager

   The root pager is the first task started after completing the kernel boot-up, and has access rights to all user memory. The root pager uses a special protocol to handle this memory. There are various special zones of the address space like IO-ports and the BIOS-memory which are system specific. Due to the backward-compatibility of the AMD64 architecture the sigma0 protocol has not to be changed. Only some constants have to be adjusted to 64-bit size.

2. Roottask - root task

   This is the second task started after sigma0. It handles all resources, like the right to create new tasks, attaching to interrupts and also maps all memory from sigma0. The root task starts most other tasks, that are not configured to be handled by other loaders.

   The root task must be capable of handling 64-bit binaries instead of 32-bit binaries. This only requires changes in the ELF interpreter. For passing parameters to the started tasks a trampoline page is used. The mechanism for parameter passing changed and so the trampoline code had to change as well.

3. L4sys - bindings

   The bindings are the user-level counterpart to the system call implementation in the kernel. They are responsible for passing the right parameter to right register and catch up the return values. The bindings are per definition architecture-specific and have completely to be rewritten. In 32-bit systems there are problems with dealing 64-bit values like the thread-id, which has to be split up and distributed in two registers. Some parameters have to be passed onto the stack as the number of registers don't suffice.

   With the AMD64 architecture this situation slightly changes as the register size is now 64-bit and 15 GPRs available. There is more flexibility for passing parameters. The new binding implementation is in detail described in appendix A.

Knowing that the AMD64 architecture has 15 GPRs which all can be used for parameter passing in system call (excluding the RSP) this seems a disadvantage a design of this port. Especially for the IPC operation it is evident that increasing the numbr of transfered data in registeres has strong performance benefits.

4. Pingpong - micro-benchmark tool

   Pingpong is a performance measurement tool to test several IPC-cases, like short-IPC, long-IPC between threads and tasks. It can be considered a synthetic micro-benchmark exercising under optimal conditions. Therefore it uses its own set of hand-optimized bindings.

   Porting to AMD64 architecture is very time consuming for each test case the binding has to be reimplemented. But it provides a good basis for comparison the ported kernel with the Fiasco/IA32 kernel. So some common IPC-cases are ported and tested to provide information for later discussions.

3.7 Summary

From the discussions above we can conclude that most changes in the current Fiasco implementation are around the low-level hardware dependent code. That comprises mainly the hardware set-up code and the paging code. All functions which are hardware independent, like system call implementations and synchronzation don't need to be changed.

The design decisions are around the boot-up scenario and page-table sharing mechanism in the kernel address space.

4 Implementation

This chapter describes the implementation of Fiasco/AMD64. It discusses which modifications of the code base were made. Some special and interesting points are highlighted.

4.1 Context Management

4.1.1 Context State

As anticipated in the design section stack overruns occured in 64-bit mode. So the TCB size was doubled to 4 Kbyte. Therefore the TCB area in the memory layout of the kernel is expanded.

Table 4.1 shows the measured TCB size of Fiasco/AMD64 compared with Fiasco/IA32. The thread sigma0 is one example where the TCB size of 2 Kbytes is exeeded.

task	name	Fiasco/IA32	Fiasco/AMD64
5.00	hello	780	1456
4.01	roottask server	716	1024
4.00	roottask pager	888	1904
2.00	sigma0	1348	2304
0.00	idle thread	816	1808

Table 4.1: Measured TCB sizes of a 'hello world' application

4.1.2 Context Switch

There are two pieces of code that are important regarding the thread switch:

`Context::switch_cpu` does a regularly context switch from one thread to another

`Thread::user_invoke` clears out user-level registers when a thread is started the first time

Switching an address space is done in the following method:

`Space_context::switch_context` reloads the CR3 register

4.2 Memory Management

4.2.1 Kernel Memory Management

Kernel memory is managed in class `Kmem`. This class sets up the master page table of the kernel address space in the boot stage. A task associated with its address space is represented by the class `Space`. This class is responsible for allocating the root page table. For performance reasons there isn't maintained a pointer to the root page table instead the memory of the class itself represents it.

The following classes implement in Fiasco/IA32 the paging data structures:

`Pdir`	holds a page directory
`Pd_entry`	accesses a page directory entry
`Ptab`	holds a page table
`Ptab_entry`	accesses a page directory pointer entry

Fiasco/AMD64 adds a new set abstractions to reflect the four-level paging hierarchy.

`Pml4`	holds a PML4
`Pml4_entry`	accesses a Pml4 entry
`Pdp`	holds a page directory pointer table
`Pdp_entry`	accesses a Pdp entry

Fiasco/IA32 has no functions that encapsulate the parsing of kernel page tables. This turns out to be a problem, as traversing is used in many places in the kernel. To simplify parsing of page tables a set of new functions is added in the class `PML4`.

`Pml4::map_range`	maps a continuous range using special allocator function
`Pml4::map_superpage`	maps a superpage using special allocator function
`Pml4::map_page`	maps a page using special allocator function
`Pml4::map_slow_page`	maps a page using kernel memory allocator

The first three methods are used in kernel initialization phase when the kernel memory allocator is not available. The last function maps a page using the regularly kernel memory allocator. Currently this method is only used in one place.

One problem that appeared recently during implementation was the question of the existence of intermediate page tables during traversing the page table hierarchy. The following methods used for traversion replace their counterparts without allocation. So traversion can be done using this methods while hiding the mechanism of allocation of intermediate page tables on demand.

`Pml4_entry::alloc_pdp`	returns the Pdp if it exists otherwise it allocates one
`Pdp_entry::alloc_pdir`	returns the page directory if it exists otherwise it allocates one
`Pdir_entry::alloc_ptab`	returns the page table if it exists otherwise it allocates one

The synchronization of kernel page tables is done in the following methods:

`Kmem::dir_init`	initializing the kernel address space an task creation
`Space::kmem_update`	synchronizing the address space with the master page table after kernel page fault
`Space::remote_update`	synchronizing a part of the address space with some other address space, used for example to setup th IPC window

The kernel memory layout is a related aspect. In general it has not to be changed in comparison the Fiasco/IA32 memory layout. The first exception is the expanded TCB area, that uses a separate page directory. Another exception are two regions which in fact correspond to variables in the class `Space_index`. As this class now represents a PML4 table, the space-index and chief-index occupy one PML4 slot each. A detailed figure of the current memory layout of Fiasco/AMD64 is in appendix D.

4.2.2 User Memory Management

As mentioned in the design section, the kernel provides primitives that facilitate user-level management. The required methods are the following

`Space::v_insert`	inserts a new page in the user address space or changes the access rights of the corresponding page
`Space::v_lookup`	searches a page in the user address space
`Space::v_delete`	deletes the correspnding page from the user address space
`Space::~Space`	destructs the whole address space

Problems occurred when a task was destructed. The current implementation uses the mechanism of Fiasco/IA32 scanning the whole user-level address space to remove mappings from the mapping tree. Cleaning up the 64-bit address space this way takes far to much time (about one second). Therefore a second implementation is considered where the cleaning of the address space is simultaneously done with the deallocation of the page tables. This solution should resolve the performance bottleneck, but is not implemented in the current version.

4.3 Kernel Entry/Exit

Most entry and exit code is written in assembler. Kernel entries in Fiasco/AMD64 have to support the expanded register set but don't need to save segment registers.

4.3.1 Exception and Interrupts

A new class `x86_64desc` is introduced to support the 128-bit wide interrupt descriptors. The class `trap_frame` is expanded to 64-bit machine word size. The complete initialization mechanism of the IDT is reused with only minor changes.

The double-fault handler using the new IST mechanism, timer interrupt handler for the RTC and the page fault handler working properly.

There are some handlers which are not available, because they are not tested or not implemented in Fiasco/AMD64:

- FPU not available handler
- timer interrupt handler for PIT
- timer interrupt handler for APIC

4.3.2 Syscalls

Fiasco/IA32 represents the content of the registers in a system call specific class. These classes make the syscall mostly architecture independent. The following list shows which class handles which syscall frame.

`Sys_ipc_frame`	IPC syscall
`Sys_id_nearest_frame`	id_nearest syscall
`Sys_ex_regs_frame`	lthread_ex_regs syscall
`Sys_thread_switch_frame`	thread_switch syscall
`Sys_unmap_frame`	fpage_unmap syscall
`Sys_task_new_frame`	task_new syscall
`Sys_thread_schedule_frame`	thread_schedule syscall

The SYSENTER/SYSEXIT entry is replaced with the new SYSCALL/SYSRET mechanism. The whole implementation can be hidden in architecture specific code of class `Cpu` and the entry code. In the result the SYSCALL entry must construct the same `Entry_frame` as an entry via a software interrupt.

So far all seven syscalls are ported and tested. Of course there are some special cases (IO-Flexpages) and extensions to the v2 API that may not work yet.

4.4 Bootup and Initialization

In the current implementation the bootstrap module sets up 64-bit mode as described in approach B. The module is consists of two parts:

1. bootstrap: 32-bit code The first part, loaded by GRUB, is responsible for extracting the self-containing 64-bit section. In a second step it sets up the page tables and GDT. Most CPU initialization code was taken from the kernel. At the end a far jump to 64-bit code is made that activates a 64-bit code segment and thus switches to 64-bit mode.
2. boostrap64: 64-bit code This code extracts further 64-bit modules, namely the kernel, root pager and root task and finally transfers control to them.

The kernel boot code is fully 64-bit and reestablishes all mappings and data structures. The only new thing it does, is to install a 64-bit IDT.

The linking and build process for bootstrap seems rather complex now, there is also a special small library which is build for the 64-bit part.

For some time there exists also a test implementation of approach C, where the kernel boot code is responsible for switching to 64-bit mode. The complexity of this code was much lower compared to the above implementation. But as the kernel build system changed, handling of binary images was excluded from the kernel. Therefore also this implementation was rejected.

The last point to mention is the multiboot structure provided by GRUB. This data structure contains mainly information about the loaded modules. As GRUB runs in 32-bit protected mode

all addresses are 32-bit wide. The design of the structure is not changed in any way to maintain compatibility to the multiboot standard.

4.5 Summary

There exists a main version Fiasco/AMD64, called Fiasco/AMD64(v1). This version implements approach A of the page table optimization, meaning that only the kernel page directory is shared in the kernel address space.

In a second step I implemented approach C of the page table optimization, called Fiasco/AMD64(v2), hoping to simplify kernel synchronization.

There exist also a third implementation, called Fiasco/AMD64(v3), that uses the IPC-window optimization.

All the implementations will be compared in the next section.

5 Measurements and Evaluation

In this chapter we show and discuss the results of the performance measurements. The benchmark tool pingpong which has several test scenarios for IPC between threads and tasks is used as a micro benchmark. Macro benchmarks could be ported to Fiasco/AMD64 in the limited time frame of this work. The used hardware is an AMD Athlon 64 (Newcastle) with 3,2 GHz and the following parameters:

Instruction TLB:	512 Entries with 4 Kbyte pages (8 Entries for 2 Mbyte pages)
Data TLB:	512 Entries with 4 Kbyte pages (8 Entries for 2 Mbyte pages)
L1 Data Cache:	64 Kbyte - 2 way associative, 64 bytes per line
L1 Instruction Cache:	64 Kbyte - 2 way associative, 64 bytes per line
L2 Cache:	512 Kbyte - 8 way associative, 64 bytes per line

All scenarios discussed here, distinguish between a warm case, where the cache is filled with proper lines and a cold case, where the cache is filled with waste, and therefore cache-misses are occur. The first one can be considered near the best case, whereas the second one can be considered a raw worst case scenario.

5.1 Performance for short IPC

The first table (5.1) shows the results of the short IPC, which transfers only two registers. The IPC path taken doesn't use a special handling for short IPCs. In the last column we see the relative overhead of Fiasco/AMD64 related to Fiasco/IA32. The small performance loss for Fiasco/AMD64 is hard to determine. A detailed analysis of the cache misses and the length and structure of the code is necessary.

Entry	Cache	Fiasco/IA32	Fiasco/AMD64	overhead/%
int30	warm	1302	1319	+1.3
syscall/sysenter	warm	1102	1129	+2.4
int30	cold	4599	5231	+13.7
syscall/sysenter	cold	3643	4833	+32.6

Table 5.1: short IPC within one address space with no shortcut

The second table (5.2) uses the same scenario as above but now an optimized path is taken, called C-shortcut. Again the performance of Fiasco/AMD64 is only a few percent slower then Fiasco/IA32 with warm cache. With cold cache the difference is considerable.

Entry	Cache	Fiasco/IA32	Fiasco/AMD64	overhead/%
int30	warm	1380	1431	+3.6
syscall/sysenter[1]	warm	308	312	+1.2
int30	cold	5008	6402	+27
syscall/sysenter	cold	1741	2934	+68

Table 5.2: short IPC within one address space with C-shortcut

The next two tables (5.3 and 5.4) present the result of short IPC between two tasks with with and without the C-shortcut. Now the effects of reloading the page tables and the TLB flush are added. Refilling the TLB seems more expansive in Fiasco/AMD64 as more page tables must be traversed. To interpret these results properly more detailed measurements have to be made, as for example examining the cache misses. Unfortunately no working infrastructure was available at this point.

Entry	Cache	Fiasco/IA32	Fiasco/AMD64	overhead/%
int30	warm	1639	1896	+15.5
syscall/sysenter	warm	549	837	+7.3
int30	cold	5634	6618	+17.5
syscall/sysenter	cold	2492	3142	+26.0

Table 5.3: short IPC between two address spaces with C-shortcut

Entry	Cache	Fiasco/IA32	Fiasco/AMD64	overhead/%
int30	warm	1641	1773	+8.0
syscall/sysenter	warm	1406	1543	+16.7
int30	cold	5202	6467	+24.3
syscall/sysenter	cold	4761	6129	+28.7

Table 5.4: short IPC between two address spaces with no shortcut

5.2 Performance for Long IPC

The following discussions deal with the long IPC performance. In the measurements we distinguish between direct long IPC and long IPC with indirect strings (indirect IPC for short). For both IPC variants there are three different cases. At first we make the direct comparison between Fiasco/IA32 and Fiasco/AMD64 in a best-case scenario with warm caches. In a next step the same is done with cold caches. And the third case examines the performance differences between the three Fiasco/AMD64 variants to validate that we achieved some performance improvements as expected. Further on the indirect IPC performance discussion can be divided into the number of transfered strings. For the

sake of completeness there is next to the diagram with 1 string transfer shown a diagram with 4 string transfers. A deeper discussion about differences is omitted here.

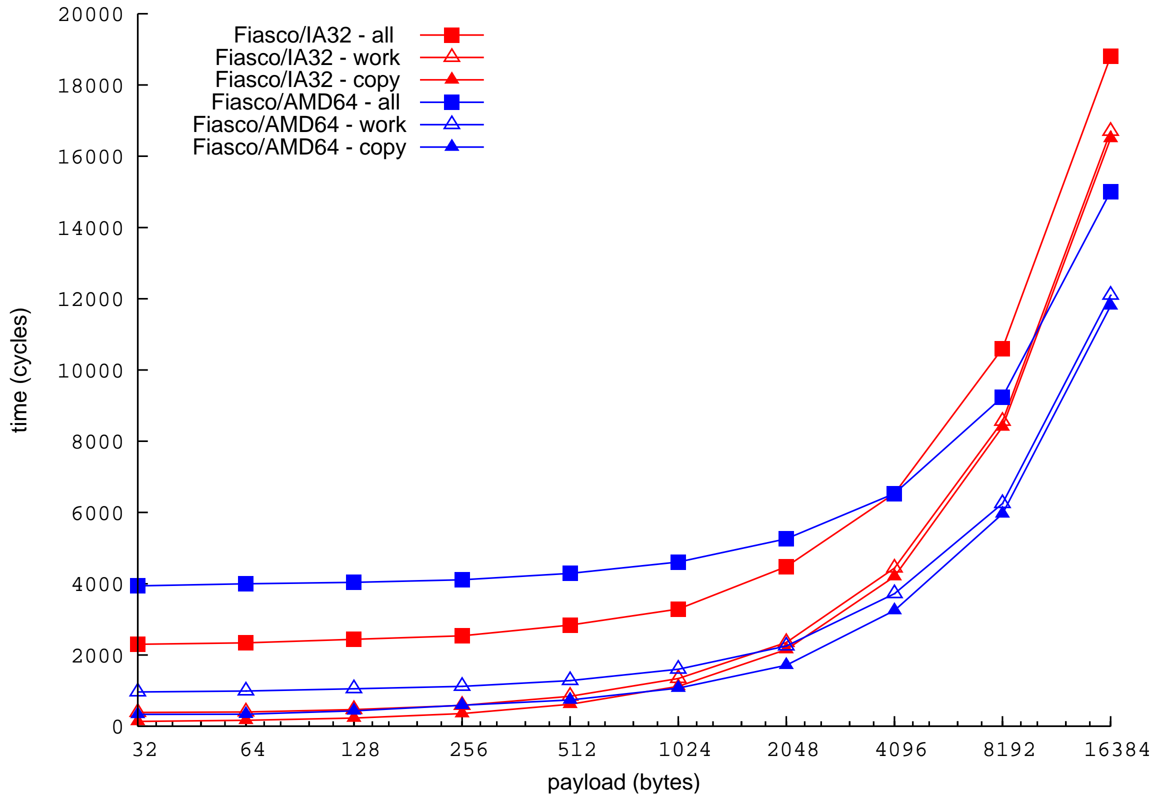

Figure 5.1: Comparison of direct long IPC between Fiasco/IA32 and Fiasco/AMD64 with warm caches

In figure 5.1 the performance difference of long IPC between Fiasco/IA32 and Fiasco/AMD64 related to the number of transfered bytes is shown. The diagram contains beside graphs for the total IPC time (labelad all), also the graphs for copying the data (labeled copy) and the sum of the copy time and setting up the IPC window (labeled work).

The performance loss compared to Fiasco/IA32 is particularly small for large payloads very high. As the number of bytes increases, Fiasco/AMD64 catches up and performs even superior for payloads larger than 4096 bytes.

The reason for this result is that the pure copy time scales better on the 64-bit environment. So the fixed overhead that is introduced by the longer code path becomes less important as the number of transfered bytes increases.

The time used for setting up page tables can be neglected both in Fiasco/IA32 and Fiasco/AMD64.

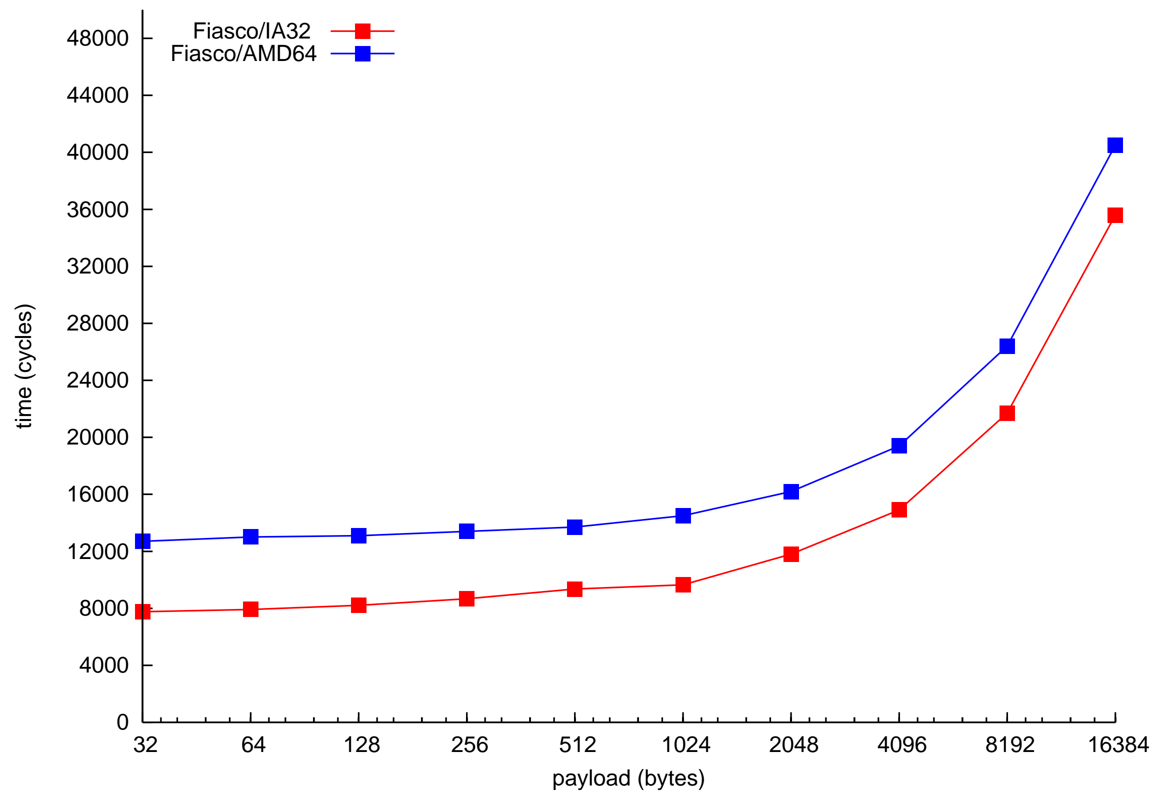

Figure 5.2: Comparison of direct long IPC between Fiasco/IA32 and Fiasco/AMD64 with cold caches

The case with cold caches in figure 5.2 shows that Fiasco/AMD64 slows down compared to Fiasco/IA32. The reason might be that there are more instruction cache misses because of the longer code path.

Further analysis is needed in order to access this hypothesis.

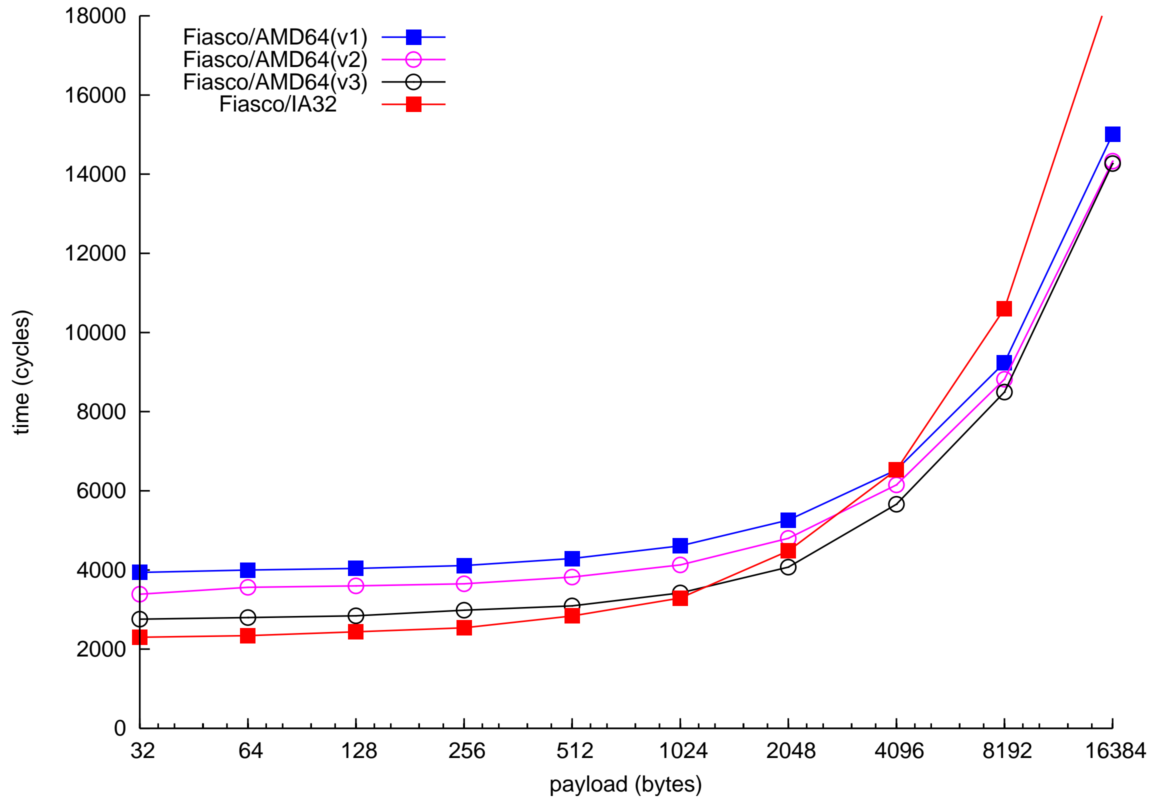

Figure 5.3: Comparison of direct long IPC between Fiasco/AMD64 variants

Figure 5.3 analysis the speedup of the two experimental optimizations of Fiasco/AMD64. Fiasco/AMD64(v2) uses a shared PDP and Fiasco/AMD64(v3) sets up the IPC window in a seperate PML4 slot. Both variants optimize the page table traversion and syncronization. For the v3 variant exists another variation without flushing the IPC window.

The first thing to notice is that both variants, v2 and v3, perform better than v1. This is an expected result, as the overhead of page table handling is reduced. But as this performance gain is constant, it becomes marginal as the payload grows.

Another point is that the TLB flush operation has a small but not imarginal impact.

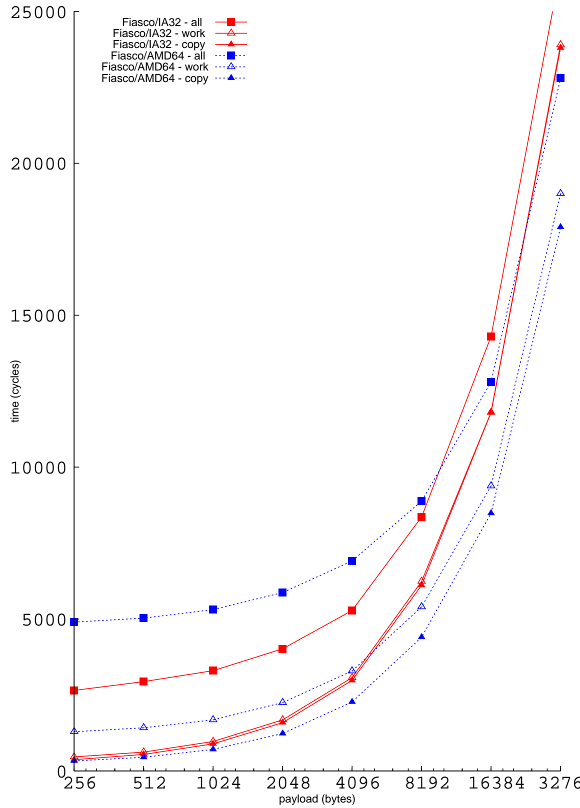

Figure 5.4: Comparison of indirect IPC with 1 string between Fiasco/IA32 and Fiasco/AMD64

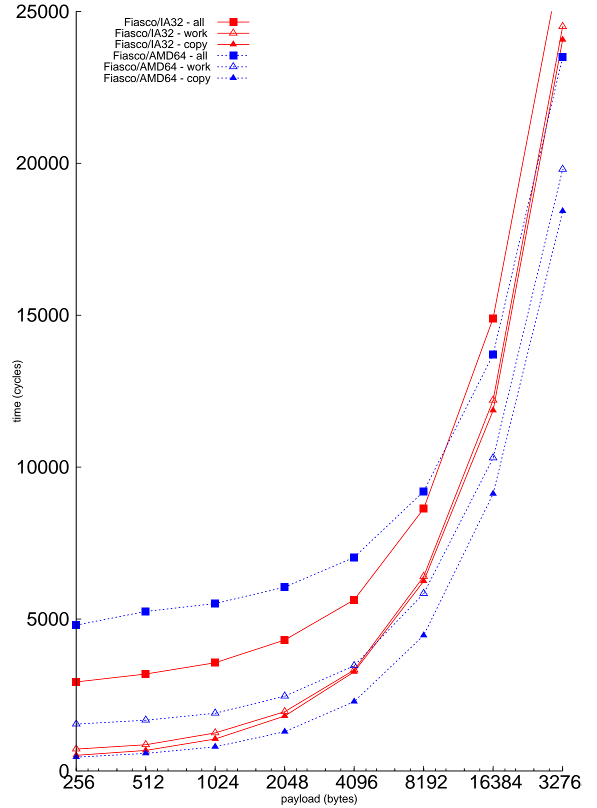

Figure 5.5: Comparsion of indirect IPC with 4 strings between Fiasco/IA32 and Fiasco/AMD64

Figure 5.4 and 5.5 show the results of the comparison between Fiasco/IA32 and Fiasco/AMD64 for indirect IPC with warm cache. Beneath the graph for the total used IPC time (labeled all) there are two more graphs. There is one graph that shows the pure copy time (labeled copy) and another graph that aggregates the copy and time for the setup of the IPC window (labeled work).

The result is similar to the direct long IPC discussion above. For small payloads Fiasco/AMD64 is slower than Fiasco/IA32. However, this slowdown decreases for growing payloads and is even reversed in a speedup for payloads of 8 Kbytes or larger.

This is due to the faster copy operation of Fiasco/AMD64 which has a significant impact as the payload grows.

The time for code execution is nearly constant and independent from the transfered payload. The difference between all-graph and the work-graph, which reflects the time of the execution path, is constant. So we can conclude that the total time only depends on the used copy function.

From both figures we can also deduce that for Fiasco/IA32 the time for setting up page

tables can be neglected. That is because the work-graph and the copy-graph are nearly identical. But the for Fiasco/AMD64 this time is a significant part of the performance loss.

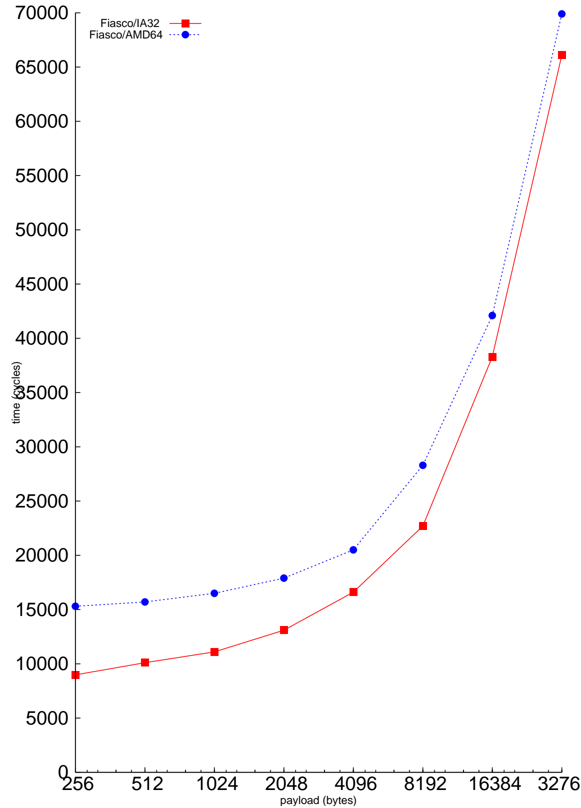

Figure 5.6: Comparison of indirect IPC between Fiasco/IA32 and Fiasco/AMD64 with 1 string and cold cache

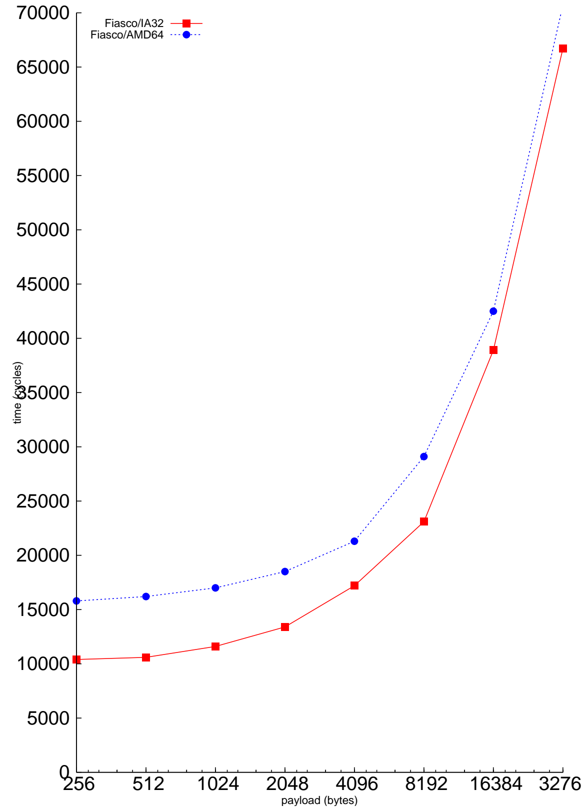

Figure 5.7: Comparison of indirect IPC between Fiasco/IA32 and Fiasco/AMD64 with 4 strings and cold cache

In figure 5.6 and figure 5.7 the comparison for cold caches is shown. Again we see that Fiasco/AMD64 is slower than Fiasco/IA32. The performance penalty of Fiasco/AMD64 does not depend on the number of transfered bytes. This may be because the code path is longer and therfore more cache misses occur.

The almost constant performance difference is a positive indication for this assumption. Other effects like TLB flushes and page table traversing can be considered of low impact.

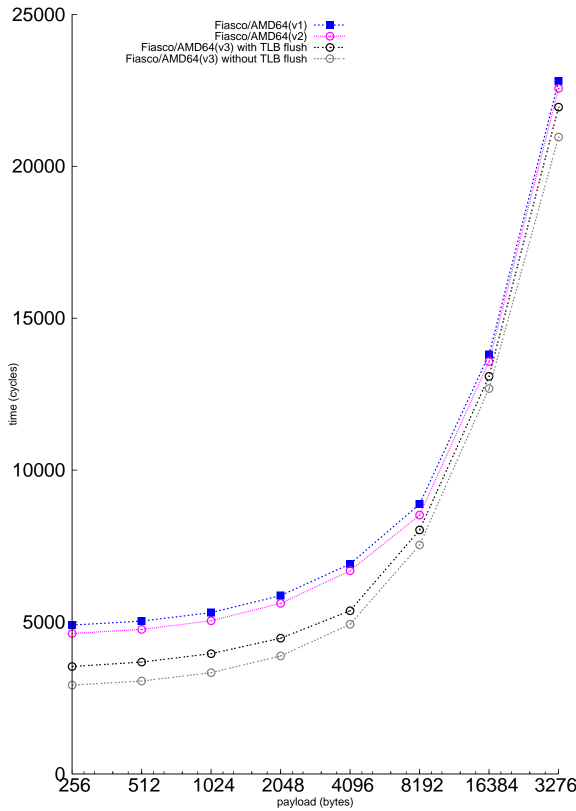

Figure 5.8: Comparison of indirect IPC between the Fiasco/AMD64 variants with 1 string and warm cache

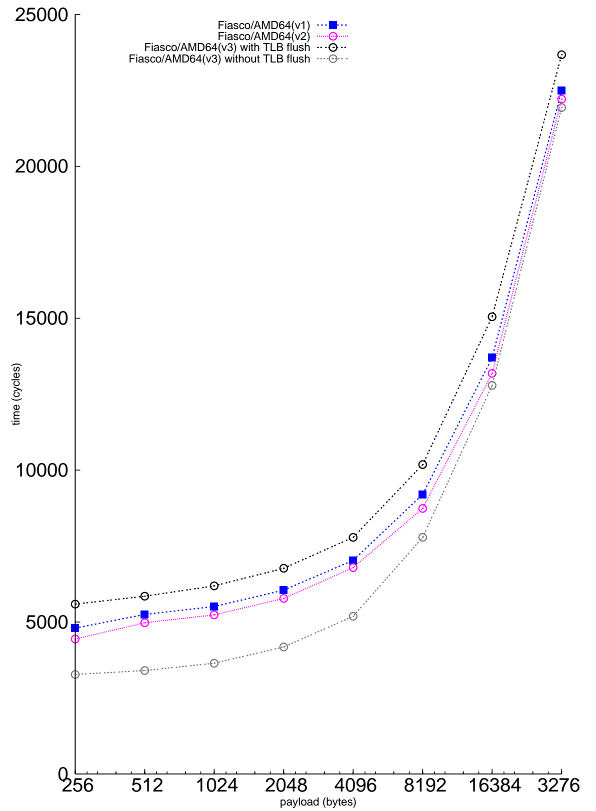

Figure 5.9: Comparison of indirect IPC between the Fiasco/AMD64 variants with 4 strings and warm cache

In figure 5.8 and figure 5.9 we see the performance of the the indirect IPC operation of the different Fiasco variants. Variant v2 and v3 of Fiasco/AMD64 behave slightly better than its original when one string is transfered.

In the case where 4 strings are copied the TLB flush operation shows significant for Fiasco/AMD64(v3). If we disable the TLB flush on the IPC window setup, the performance speeds up considerable. It follows that flushing the TLB on PML4-level is a very expensive operation which should be avoided.

5.3 Summary

The Short IPC under Fiasco/AMD64 is only marginal slower than under Fiasco/IA32. For long IPC both direct and indirect, the performance is considerable slower on a small number of transfered bytes. But due to the faster copy routine the performance penalty decreases and even becomes reversed if the number of transfered bytes grows. It also seems promising to explore more optimization technics to improve performance as shown

in two examples.

6 Conclusions, Open Topics and Future Work

The current work shows that it is possible to port Fiasco/IA32 to the AMD64 architecture with limited effort. The portion of modified and rewritten code is located mostly in well-defined and architecture-depended parts in the kernel sources.

Currently there is a minimal set of user-level packages ported to validate and measure the kernel. The results show that there are some performance impacts for 64-bit architecture.

The performance for short-IPC is comparable to Fiasco/IA32. For long-IPC the overhead of the longer code path and the four-level page table hierarchy is non-negligible.

From my point of view there are three pieces for future work:

1. The kernel has to be enhanced with features already implemented in Fiasco/IA32. This comprises the following set:

   - PIC/APIC-Support
   - Assembler IPC-shortcut
   - IO-Protection
   - UTCB-Support
   - Exception-IPC
   - SMAS

   Almost all, except the last point, can be done with little effort.

2. The kernel can be enhanced with new features unique to Fiasco/AMD64. This might be the subset of:

   - Implementing the NX-Bit
   - Growing the number of transferred registers in short-IPC
   - Supporting legacy 32-Bit applications

3. kernel debugger support should be completed, with the following features:

   - Backtracing
   - Loading Symbols and Lines Debug Information
   - Single-step tracing

4. Porting more user-level programs seems very urgent to enable full application support as known for Fiasco/IA32 and get more experience in 64-bit architectures.

The direction of development for Fiasco/AMD64 heavily dependes on future of hardware developments. As Intel switches from IA64 architecture to EMT64 architecture as 64-bit architecture, which is compatible with the AMD64 architecture, there might be a strong need.

Nevertheless the experience of one more successful 64-bit port of Fiasco helps this kernel to leverage for future research projects.

Appendix A

Interrupt handling

number	name	gate	frame	handling
0x0	#DE	Interrupt gate	trap-frame	Dumping trap state
0x1	#DB	Interrupt gate	trap-frame	Enter kernel debugger
0x2	NMI	Interrupt gate	trap-frame	Dumping trap state
0x3	#BP	Interrupt gate	trap-frame	Enter kernel debugger
0x4	#OF	Interrupt gate	trap-frame	Dumping trap state
0x5	#BR	Interrupt gate	trap-frame	Dumping trap state
0x6	#UD	Interrupt gate	trap-frame	Dumping trap state
0x7	#NM	Interrupt gate	trap-frame	Switching FPU
0x8	#DF	Task gate	trap-frame	Dumping trap state
0x9		Interrupt gate	trap-frame	Dumping trap state
0xa	#TS	Interrupt gate	trap-frame	Dumping trap state
0xb	#NP	Interrupt gate	trap-frame	Dumping trap state
0xc	#SS	Interrupt gate	trap-frame	Dumping trap state
0xd	#GP	Interrupt gate	trap-frame	Dumping trap state or Handling a special case
0xe	#PF	Interrupt gate	no trap-frame	Handling user-level and kernel page faults
0x20		Interrupt gate	no trap-frame	Handling PIT timer-interrupt
0x28		Interrupt gate	no trap-frame	Handling RTC timer-interrupt
0x30		Interrupt gate	syscall-frame	IPC syscall
0x31		Interrupt gate	syscall-frame	id_nearest syscall
0x32		Interrupt gate	syscall-frame	fpage_unmap syscall
0x33		Interrupt gate	syscall-frame	thread_switch syscall
0x34		Interrupt gate	syscall-frame	thread_schedule syscall
0x35		Interrupt gate	syscall-frame	lthread_ex_regs syscall
0x36		Interrupt gate	syscall-frame	task_new syscall
0x3d		Interrupt gate	no trap-frame	Handling APIC timer-interrupt

Table A.1: Handling of interrupts in Fiasco

Appendix B

System call conventions

ipc

Input	Register		Register	Output
snd descriptor	RAX		RAX	*msg.dope*
timeouts	RCX		RCX	∼
msg.w0	RDX		RDX	*msg.w0*
msg.w1	RBX	— INT 0x30⟶	RBX	*msg.w1*
rcv descriptor	RBP		RBP	∼
dest id	RSI		RSI	*source id*
∼	RDI		RDI	∼
∼	R8		R8	∼
	...		...	
∼	R15		R15	∼

id_nearest

Input	Register		Register	Output
∼	RAX		RAX	*type*
∼	RCX		RCX	∼
∼	RDX		RDX	∼
∼	RBX	— INT 0x31⟶	RBX	∼
∼	RBP		RBP	∼
dest id	RSI		RSI	*nearest id*
∼	RDI		RDI	∼
∼	R8		R8	∼
	...		...	
∼	R15		R15	∼

fpage_unmap

Input	Register		Register	Output
fpage	RAX		RAX	∼
map mask	RCX		RCX	∼
∼	RDX		RDX	∼
∼	RBX	— INT 0x32⟶	RBX	∼
∼	RBP		RBP	∼
∼	RSI		RSI	∼
∼	RDI		RDI	∼
∼	R8		R8	∼
	...		...	
∼	R15		R15	∼

thread_switch

Input	Register		Register	Output
∼	RAX		RAX	∼
∼	RCX		RCX	∼
∼	RDX		RDX	∼
∼	RBX	— INT 0x33⟶	RBX	∼
∼	RBP		RBP	∼
dest id.low	RSI		RSI	∼
∼	RDI		RDI	∼
∼	R8		R8	∼
	...		...	
∼	R15		R15	∼

thread_schedule

Input	Register		Register	Output
param word	RAX		RAX	*old param word*
∼	RCX		RCX	*time*
∼	RDX		RDX	∼
ext preempter	RBX	— INT 0x34⟶	RBX	*old preempter*
∼	RBP		RBP	∼
dest id	RSI		RSI	*partner*
∼	RDI		RDI	∼
∼	R8		R8	∼
	...		...	
∼	R15		R15	∼

lthread_ex_regs

Input	Register		Register	Output
lthread no	RAX		RAX	*old RFLAGS*
RSP	RCX		RCX	*old RSP*
RIP	RDX		RDX	*old RIP*
int preempter	RBX	— INT 0x35⟶	RBX	*old preempter*
∼	RBP		RBP	∼
pager	RSI		RSI	*old pager*
∼	RDI		RDI	∼
∼	R8		R8	∼
	...		...	
∼	R15		R15	∼

task_new

Input	Register		Register	Output
mcp / new chief	RAX		RAX	∼
initial RSP	RCX		RCX	∼
initial RIP	RDX		RDX	∼
pager	RBX	— INT 0x36⟶	RBX	∼
∼	RBP		RBP	∼
dest task	RSI		RSI	*new task*
∼	RDI		RDI	∼
∼	R8		R8	∼
	...		...	
∼	R15		R15	∼

Appendix C

Compiler calling convention

The used compiler gcc 3.4 has the following convention when passing parameters to the a function:

The first six parameters are saved the registers: RDI, RSI, RDX, RCX, R8 and R9 in this order. All other parameters are passed on the stack.

Caller saved registers are RAX, RCX, RDX, RSI, RDI, R8, R9, R10 and R11. Callee saved registers are RBP, RBX, R12, R13 and R14.

The compiler calling convention defines a red-zone on the stack. A red-zone is a 128-byte area beyond the location of the stack pointer by signal or interrupt handlers. And therefore can be used for temporary data without adjusting the stack pointer. The compiler flag -m no-red-zone disables this feature.

Appendix D

Kernel address space

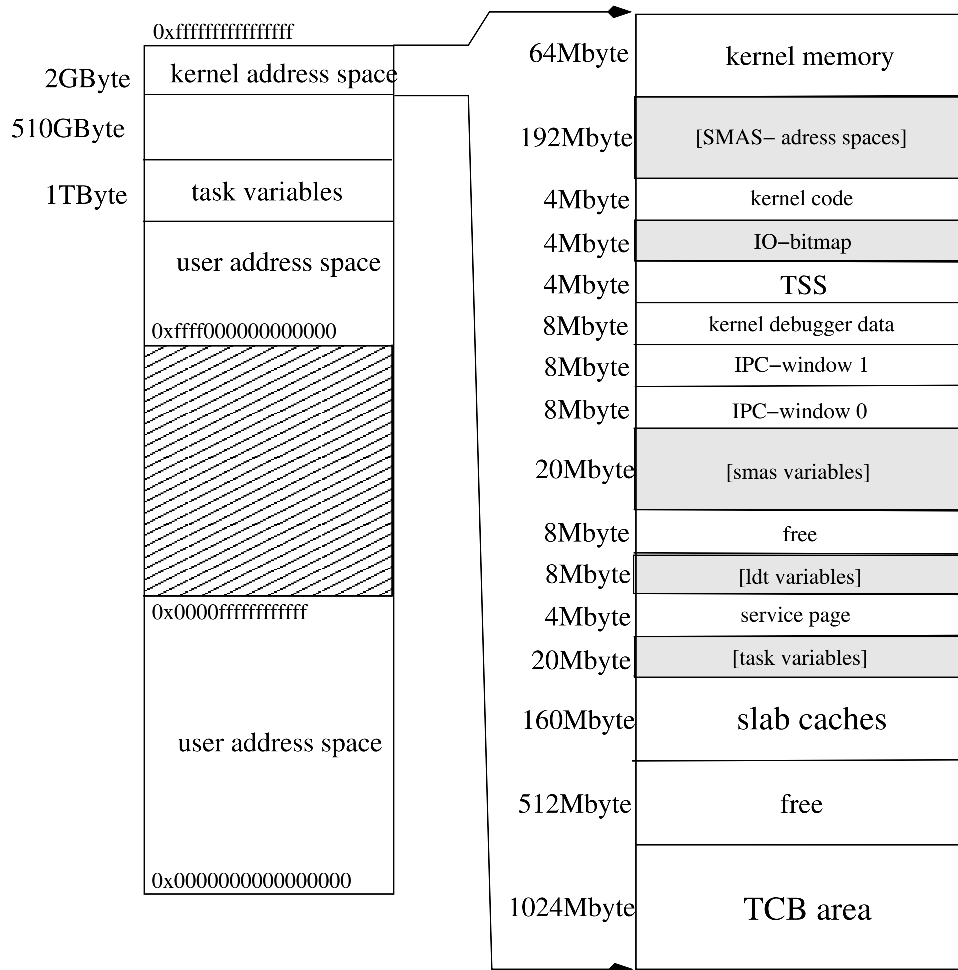

Figure D.1: Shared page table

Appendix E

Glossary

ABI Application Binary Interface

APIC Advanced Programmable Interrupt Controller

CPU Central Processing Unit

ELF Executable and Linkage Format

GDT Global Descriptor Table

GPRGeneral Purpose Register

GRUB Grand Unified Boatloader

IDT Interrupt Descriptor Table

IPC Interprocess communcication

IRQ Interrupt Request

IST Interrupt Stack Table

PDP Page Directory Pointer

PIC Programmable Interrupt Controller

PIT Programmable Interrupt Timer

PML4 Page Map Level 4

TCB Thread Control Block

TSS Task State Segment

Bibliography

[1] Grub manual, August 2005.

[2] Martin Borriss, Michael Hohmuth, Jean Wolter, and Hermann Härtig. Portierung von Linux auf den μ-Kern L4. In *Int. wiss. Kolloquium*, Ilmenau, September 1997.

[3] U. Dannowski, J. LeVasseur, E. Skoglund, and V. Uhlig. L4 experimental kernel reference manual, version x.2. Technical report, 2004. Latest version available from: `http://l4hq.org/docs/manuals/`.

[4] Bryan Ford, Erich S. Boleyn, Kunihiro Ishiguro, and OKUJI Yoshinore. *The Multiboot Specification*. Free Software Foundation.

[5] H. Härtig, R. Baumgartl, M. Borriss, Cl.-J. Hamann, M. Hohmuth, F. Mehnert, L. Reuther, S. Schönberg, and J. Wolter. DROPS: OS support for distributed multimedia applications. In *Proceedings of the Eighth ACM SIGOPS European Workshop*, Sintra, Portugal, September 1998.

[6] S. Hoffmann. Kleine addressräume für fiasco, July 2002.

[7] Intel Corp. *Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference*, 1999.

[8] J. Liedke J. Glauber, F. Mehnert. Fiasco kernel debugger manual, November 2005.

[9] B. Kauer. L4.sec implementation, kernel memory management, May 2005.

[10] J. Liedtke. L4 reference manual (486, Pentium, PPro). Arbeitspapiere der GMD No. 1021, GMD — German National Research Center for Information Technology, Sankt Augustin, September 1996. Also Research Report RC 20549, IBM T. J. Watson Research Center, Yorktown Heights, NY, September 1996.

[11] AMD Technology. Amd64 architecture programmer's manual volume 2: System programming, September 2003.

[12] A. Warg. Portierung von Fiasco auf IA-64, 2002.