Grosser Beleg: Portierung von Fiasco auf IA-64

Alexander Warg
<alex.warg@gmx.net>
TU-Dresden
Fakultät für Informatik
Institut für Systemarchitektur

May 3, 2002

.

Contents

List of Figures

Chapter 1

Introduction

Today the use of computers increases in every field and thus the number of different computer platforms. There are computers integrated in washing machines, cars, but even PCs and workstations on the office desk, and big servers in the basement of companies. IA-64 is a computer architecture, developed by HP and Intel, targeting high end workstations and servers. It has some new concepts to reach a better performance and also better compatibility among different implementations.

Another fact is the so called realtime capability, which is needed to guarantee quality of service. Realtime systems are used in embedded devices like car engine controls, but more and more also on multimedia servers. At the Dresden University of Technology a project called Dresden Realtime Operating System (DROPS) is object of research. It is based on a realtime capable implementation of the L4 API, called Fiasco. The L4 API is a microkernel interface that is reduced to the minimum required functionality.

Fiasco is an implementation for the Intel 32-bit architecture (IA-32). It is used to run realtime applications, like an MPEG player with guaranteed frame rates, in combination with timesharing applications on top of a user-mode Linux port (L4Linux).

At the moment, there exist no microkernel-based operating systems for IA-64, hence there is no base for research on this area. The target of this work is to provide an L4 implementation for IA-64 (Fiasco64).

1.1 Terminology

The term Fiasco always refers to the IA-32 implementation. The term Fiasco64 stands for the IA-64 implementation, which is the result of this work.

1.2 About this Document

The following chapter (Fundamentals and Related Work) gives an overview of some basics that should help for a better understanding of this work. The Design chapter is the main part of this document. It contains the design of the Fiasco64 kernel itself, but also a description of the boot loader and the root pager. In Chapter 4, some, hopefully interesting, implementation efforts are outlined. A number of measurements are collected in Chapter 5, work to be done in the future is discussed in Chapter 6.

Chapter 2

Fundamentals and Related Work

In this chapter I introduce some basics about operating systems, computer platforms in general, and the IA-64 platform. This chapter may be skimmed and parts read later, if more detailed information is needed.

2.1 Operating Systems

Most common operating systems, like Linux or Windows, are based on a monolithic kernel. The term monolithic means that most of the device drivers are integrated into the operating system kernel and run with kernel privileges. This may be a problem if some driver does not work correctly, for example, a file-system driver that is integrated into the Linux kernel may crash the whole system by accidentally overwriting some sensitive kernel data structures in memory. Furthermore such complex programs are much more error prone than small ones.

One possible solution for this problem is the microkernel paradigm. This approach reduces operating-system kernel to the minimum required functionality. For instance, all device drivers are moved from the kernel to user-mode programs.

2.1.1 Microkernel Paradigm

The term *microkernel* is somewhat self explaining. *Micro* indeed refers to the small size of the kernel. The idea is to move as much functionality as possible from the operating-system kernel into non-privileged user programs. In the past there were some approaches to put the microkernel paradigm into practice, for example, the Mach project of the School of Computer Science, Carnegie Mellon University (see [Baron et al., 1990]). But most of them did not reach any success, because they mostly performed very poor and the positive properties could not compensate the performance impact (see [Härtig et al., 1997]).

Today, the microkernel approach is again subject of research. There are microkernels of the so-called second generation, they are very minimalistic and highly optimized. L4 is such a second-generation API mostly developed by Jochen Liedtke. It only provides very basic abstractions and tries to remove any policy from the kernel. The entities provided by L4 are *threads* and *address spaces*. Further the API offers an IPC mechanism to permit interaction of threads. All exceptions, like hardware interrupts or page faults are transformed into IPC and send to special user-space threads. The documentation of L4 can be found in [Liedtke, 1996].

FIASCO (see [Hohmuth, 1998]), which is the base of my further work, is a C++ for IA-32 implementation of the second-generation L4 Application Programming Interface (API). It was developed at the Dresden University of Technology in the context of the DROPS project.

2.1.2 Other L4 Implementations

At the moment there are a couple of implementations of the L4 interface for different platforms.

L4Ka/Hazelnut University of Karlsruhe (IA-32 and ARM)

L4/MIPS University of New South Wales (MIPS R4x00)

L4/Alpha Dresden University of Technology and University of New South Wales

A more recent and more complete listing of L4 projects can be found at `http://os.inf.tu-dresden.de/L4/impl.html`. The complete documentations of the Pentium, MIPS, and Alpha ports are [Liedtke, 1996, Elphinstone et al., 1999, Potts et al., 2001].

2.2 Computer Platforms

Today, there are many different computer platforms on the market. For example, there are on the PC sector the IBM compatible PC (aka IA-32), Apples Mac, and the PowerPC, and on the server side Sun's SPARC architecture, Intel's 64 bit (IA-64), and IBM's S390.

All these different platforms are in general characterized by the type of the central processing unit (CPU), the internal data bus, and some components close to the processor, like the interrupt controller.

For the development of microkernels, the main aspect is the CPU and may be the interrupt controller, because there are no drivers for peripheral hardware in the kernel. For the remaining document the term platform stands mainly for the type of the CPU.

The various kinds of processors differ almost only in the complexity and the structure of the instruction set, the management of virtual memory and the protection concepts.

2.3 IA-64, Intel 64-bit Architecture

Processors are mostly classified by the instruction-set architecture. The common IA-32 is a classic Complex Instruction Set Computing (CISC) architecture. The complement of CISC is Reduced Instruction Set Computing (RISC). MIPS R4000 is a typical member of the RISC family. Another more orthogonal class is the Very Long Instruction Word (VLIW) architecture that encodes several parallel computed instructions into one instruction word. VLIW architectures are not common in general purpose computing, but there are some implementations for embedded systems (especially DSPs).

The IA-64, developed by HP and Intel, cannot be put directly into one of these classes. IA-64 unifies RISC concepts, like the load-store architecture, with some VLIW concepts.

It is called an Explicit Parallel Instruction Set Computing (EPIC) architecture. The term EPIC points to the enhanced VLIW concepts that remove the VLIW's weaknesses on binary compatibility and make room for different implementations that process the same machine code.

The following part contains a short summary of IA-64 properties that are relevant for the porting of Fiasco to Fiasco64. The complete documentation of IA-64 and Itanium can be found in [Intel IA-64 V1, 2000, Intel IA-64 V2, 2000, Intel IA-64 V3, 2000, Intel IA-64 V4, 2000], these manuals are also available on the web at `http://developer.intel.com/design/itanium`

2.3.1 Instruction-Set Architecture

The characteristic of the EPIC architecture of IA-64 is, one instruction word, called bundle, encodes three instructions, which are executed on an execution unit according to their type. More than one bundle may be executed at a time. For example, Itanium, which is a special implementation of IA-64, executes at most two bundles in parallel, but other implementations may execute even more. To solve dependencies between consecutive instructions, explicit stop bits have to be set. Stop bits are also encoded in the instruction word and define points for sequential order of single instructions.

Figure 2.1 shows a brief example of the processing of the instruction stream in an Itanium processor. The letters 'M', 'I', 'F', and 'B' refer to the different execution units. 'M' stands for memory access, 'I' for integer, 'F' for floating point, and 'B' for branching.

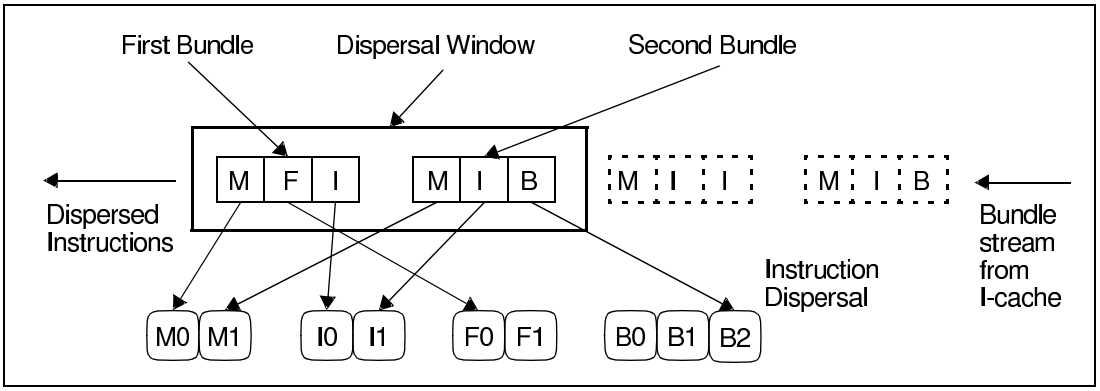


Figure 2.1: Processing of the instruction stream on Itanium, from [Intel Itanium, 2001]

2.3.2 Predication

The instruction set of Intel 64-bit architecture (IA-64) provides predication for almost every operation. Predication means that results of an instruction can be committed or thrown away according to a predicate. A predicate is a one-bit value stored in a special predicate register; the IA-64 provides 64 predicate registers.

2.3.3 Speculation

Speculation will not be described in detail, because it has less impact on the overall kernel design; no speculation is used in the kernel itself.

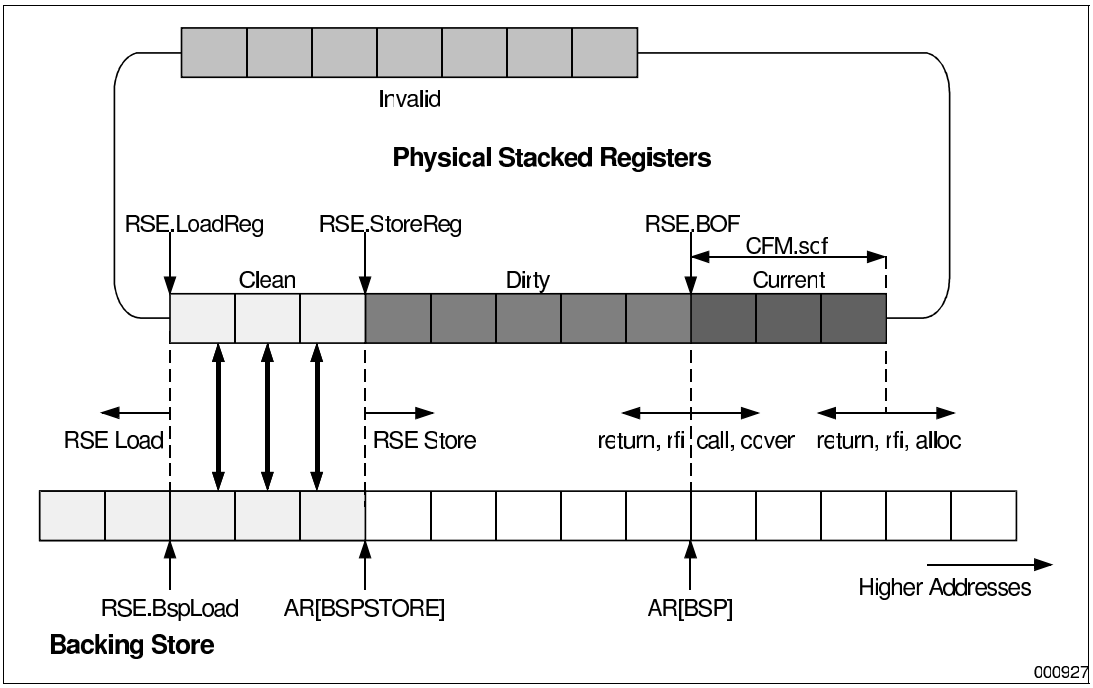


Figure 2.2: Functioning of the register stack, from [Intel IA-64 V2, 2000].

Speculation on IA-64 is introduced to hide the latency of memory transactions, means to remove them from the critical path of the program. It is intended to be used by a compiler to enhance instruction level parallelism.

From the FIASCO porting point of view it is limited to an extra bit in all general purpose registers, the so called Not a Thing (NaT) bit, or a special NaT value in the floating-point registers, and a NaT-consumption fault. The NaT-consumption fault is thrown when a register marked as NaT shall be stored to memory or is used as memory address in a store instruction.

The NaT bit in every general register indicates whether a speculative instruction, with the specific register as target, failed or is still in progress, thus the register contents are invalid. The NaT value in a floating-point register is the respective counterpart of a set NaT bit a general register.

To preserve register contents including the NaT bit a special store instruction, **spill**, has to be used. The **fill** instruction is used to restore a register that is saved with spill. These special instructions handle on the one hand the normal contents of a register and on the other hand store/restore the NaT bits to/from the *unat* register. Using the spill operation also avoids the NaT-consumption fault. The stacked registers, described in Section 2.3.4, are handled similar; their NaT bits are preserved in the *rnat* register, upon Register Stack Engine (RSE) spills.

2.3.4 Register Stack

To avoid unneeded spills and fills of registers at function calls the IA-64 provides the register stack. The register stack is based on register renaming and provides a virtually infinite number of registers. The processor unit that handles the register stack is also called Register Stack Engine (RSE).

At every function call the callee gets a new frame on the register stack. The current

register stack frame may be enlarged or shrunk with special instructions.

Because the number of physically implemented registers is limited to a finite number, the processor spills transparently registers of older stack frames to a backing store, if there are no more physical registers left to allocate a new frame. The backing store resides in normal memory.

To the contrary at return instructions the old stack frame is transparently restored from the backing store.

In Figure 2.2 on the preceding page the functioning of the register stack is shown. There the implementation of the physical registers as a ring buffer is outlined. The register AR[BSP] contains the backing store address, where the current stack frame will be spilled. AR[BSPSTORE] holds the memory address of the next RSE spill operation and RSE.BspLoad the address of the next RSE load operation.

2.3.5 Virtual Memory

Like most of the common general purpose processors IA-64 provides virtual memory. The translation from virtual to physical addresses is based on a software-filled Translation Lookaside Buffer (TLB), but there may also be hardware support for loading translations, the so called VHPT walker. The VHPT walker automatically loads translations from a virtual-mapped linear page table, hence reduces the number of TLB misses that must be handled by software and increases the system performance. The term virtual-mapped linear page table means either a linear indexed or hashed array of page-table entries that reside in virtual memory. The specific behavior is controlled via various bits in the processor status register.

Figure 2.3 on the following page shows the translation mechanism from virtual to physical addresses. Please consult the IA-64 System Architecture Manual for a full explanation.

2.3.6 Interruptions

Interruptions are events that transfer the flow of control to an interruption-handling routine. During this, some processor state is saved by the processor automatically. Upon completion of the handling a return from interruption (`rfi`) is executed, which restores the saved state.

Interruptions are classically divided into two groups. The first group are interruptions that are caused by special instructions, like page faults (caused by memory access operations). These interruptions are sometimes called to be synchronous. The other are external interrupts that occur with no association to the currently executed instruction, these interruptions are sometimes called to be asynchronous.

On IA-64, the terms synchronous and asynchronous are used in a different manner, because the classic definition assumes that no interruptions are delivered in the middle of an instruction. IA-64 uses the term **synchronous** for all interruptions that are synchronous with respect to the instruction stream, means all previous instructions appear to be completed before the delivery. Interruptions are called to be **asynchronous** if they may occur in the middle of an instruction and processor resources may be in an undefined state.

Interruptions on IA-64 are divided into four types: Aborts, Interrupts, Faults, and Traps.

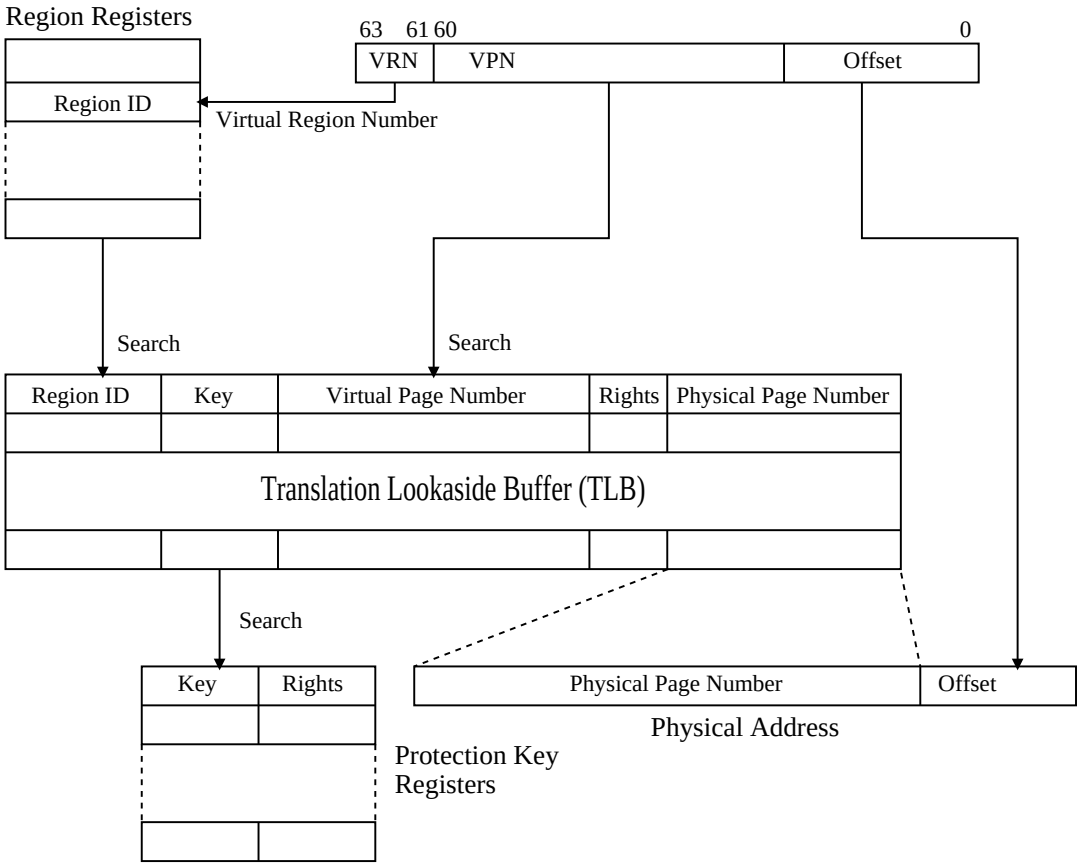


Figure 2.3: IA-64 Address Translation

Abort: A processor has detected an internal malfunction or a processor reset. Aborts are Processor Abstraction Layer (PAL)-based interruptions and therefore serviced in the processor firmware; they are not of interest for the operating system.

Interrupt: An external or independent entity, like an I/O-device or another processor, requires attention. Interrupts are synchronous with respect to the instruction stream. All previous instructions appear to be completed. Interrupts are divided into initialization, platform management, and external interrupts. Platform management and initialization interrupts are serviced by the processor firmware. External interrupts are serviced by the operating system.

Fault: The current instruction requests an action that cannot or should not be carried out. Faults are synchronous with respect to the instruction stream and must be serviced by the operating system. For example, TLB misses or access right violations are faults.

Trap: The instruction just executed requires system intervention. Traps are synchronous with respect to the instruction stream. All previous and the trapping instruction are completed before the interruption. The operating system is responsible for servicing traps. An example is the *Taken Branch Trap* that is triggered after a taken branch, if a special bit in the processor status register is set.

Unless otherwise indicated, the term “interruption” in the rest of this document refers to operating-system serviced interruptions.

Upon an interruption, the hardware saves the minimum state required to enable software to service the event and continue. The processor provides a set of interruption resources, to save the state. This state together with the interruption vector are enough information either to resolve the cause or surface the event to higher levels of the operating system.

In addition the processor switches the banked general registers (r16–r31) to a second register bank. This immediate set of general registers can be used to service the interrupt efficiently, or to save the context and enter the high-level operating-system code.

Interruptions are delivered via the Interruption Vector Table (IVT). The address of the IVT has to be stored in the IVA control register (cr2). The IVT directly contains the code of the interruption handlers.

You can find a description of the saved state and how to control interruption handling in [Intel IA-64 V2, 2000].

Chapter 3

Design

The main goal of the work was to provide an L4 implementation for IA-64. The result should be a running microkernel and some basic test applications. To run test applications it is also necessary to have a Root Pager and a Root Task. The root pager is the initial address space for an L4 system; it is the first level of the mapping hierarchy of L4. The root task is the task that owns the task-creation rights on system start and so has to create further boot-time address spaces.

The work is based on Fiasco, an IA-32 implementation of L4. And a major concern was to reuse as much code as possible. Also the kernel interface is taken from Fiasco that provides the L4 API Version 2. Because of the preceding things, some limitations have to be accepted at the moment.

3.1 General Data types

Most of the general data types, which are an important part of the kernel interface, are similar to the types used on MIPS (see [Elphinstone et al., 1999]). In compare to the IA-32 data types, only the sizes of some bit fields are adjusted with respect to the 64-bit data words on IA-64.

3.1.1 Unique IDs (UIDs)

The layout and meaning of unique IDs is taken from Fiasco, because on IA-32 UIDs are already 64-bit wide values. The layout of UIDs is illustrated in Figure 3.1(a, b, c, and d) a description in more detail can be found in [Liedtke, 1996].

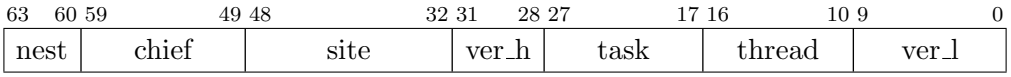
3.1.2 Flexpages (Fpages)

Flexpages are the L4 representation of virtual memory areas (see [Liedtke, 1996]).

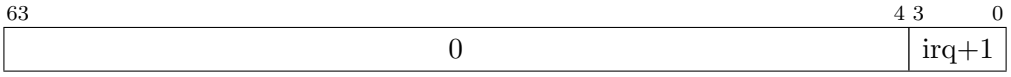
Flexpages (Fpages) have in principle the same layout as on IA-32 simply extended to 64 bits. So as shown in Figure 3.1(e) only the *page* part is grown to 52 bits and the *size* part is extended to 7 bits.

In contrast to IA-32, no fpages for I/O-ports are needed, because on IA-64 all I/O-ports are memory mapped and can be handled with the normal fpage mechanism.

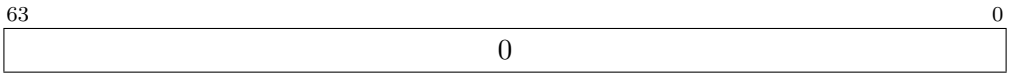
(a) Thread ID



(b) Interrupt ID

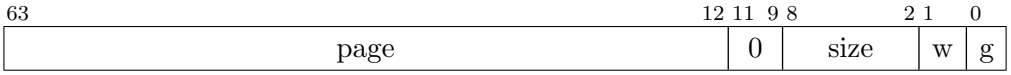


(c) Nil ID

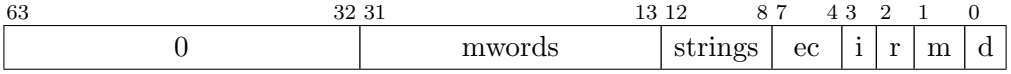


(d) Invalid ID

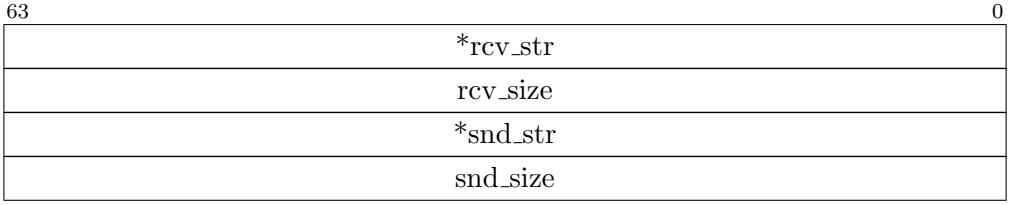
(e) Flexpage



(f) Message dope



(g) String dope



(h) Timeout

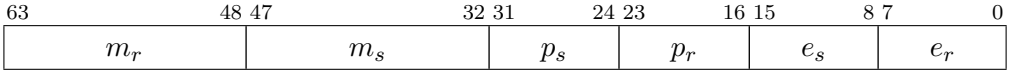


Figure 3.1: Layout of General Data Types

3.1.3 Messages

Messages are the basic entities that can be transfered with IPC operations.

Also messages have in general the same layout as on IA-32, the only difference is the size of a data word, which is 64 bits on IA-64 in contrast to 32 bits on IA-32. In account of the bigger size of message words the typed words in a message also have a slightly different layout.

Message dopes are zero extended to 64 bits, so the result is the layout shown in Figure 3.1(f). A String dope consists of four 64-bit values, pictured in Figure 3.1(g).

3.1.4 Timeouts

Timeouts are used to control IPC operations. The send, receive, and page fault timeouts have the same meaning and encoding as on IA-32. The difference is the width of some

fields, so the mantissas of send and receive timeout are now 16 bit wide, the page fault timeouts and the exponents are 8 bit wide. The layout of the IA-64-timeout type is pictured in Figure 3.1(h). For a description of the encoding refer to [Liedtke, 1996]. The timeout data type is an exception from the MIPS-like layout; on MIPS it is only a 32-bit word.

3.2 Memory

3.2.1 Address Space Layout

First of all, the common layout of all virtual address spaces must be defined. The design is on the one hand influenced by the underlying platform and its virtual memory mechanisms; and on the other hand by the algorithms used in the kernel itself.

The IA-64 has a virtual memory management as described in Section 2.3.5. As shown there the 64-bit address space is divided into eight regions, specified by the three most significant bits of the address. The region number on their part is mapped to a region ID (RID) via the region registers. In order to reduce the frequency of TLB flushes the Virtual Memory Region Ids (RIDs) may be used as address space identifiers. So it is very common to split the address space into task private regions and common (global) regions. Common regions contain for example globally shared data or the kernel that is mapped into each address space.

The layout of all address spaces in Fiasco64 is shown in Figure 3.2 and is based on the region model of IA-64.

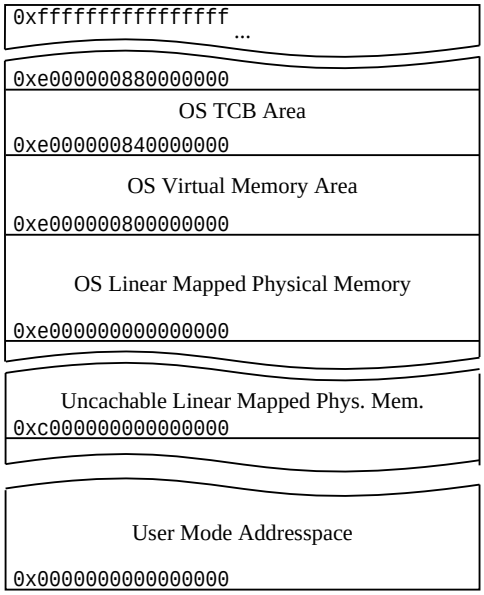


Figure 3.2: Layout of Fiasco64 Address spaces

At the moment there are two shared regions, the kernel region, and the uncached memory region. The kernel region (Region 7) contains all mappings needed by the kernel itself. The uncached region (Region 6) is provided to have uncached access to any physical memory or memory-mapped devices. For example, the VGA console memory or the memory mapped I/O-ports of the serial port are accessed via this region. The uncached region may be removed if other appropriate mechanisms to select mapping

attributes are provided.

The remaining six regions (0-5) are actually available for user address spaces, but the current implementation of the mapping database, described in Section 3.2.3, restricts mappings to the first 4 GByte of virtual memory and consequently to Region 0.

3.2.2 Page tables

Page tables are the data structures used for the mappings of virtual to physical addresses.

My first goal was to encapsulate page tables completely into a C++ class. The base for the encapsulation is the class `space_context_t`, already defined in Fiasco. The major difference from Fiasco is that page tables should almost never be accessed directly that means around the interface provided by the `space_context_t` class. The only exception from this rule is the highly optimized TLB-miss handler, which is a Lightweight Interruption Handler (see Section 3.4.1) and therefore implemented in assembler.

The target of this encapsulation is to reduce redundancy in the code and to pave the way for other page-table implementations. Furthermore the new design is not restricted to IA-64 version, but may be also ported back to IA-32 Fiasco.

At the moment, page tables are implemented as a three level tree structure, as shown in Figure 3.3 on the following page. The current page size is 4 KByte, thus more Fiasco code could simply be reused.

A problem is that such primitive forward tree structures are not sufficient for 64-bit address spaces. They must either consist of a large number of levels or a vast number of entries per level. For instance to cover a complete 64-bit address space with a three-level table, there have to be $\sqrt[3]{2^{(64-12)}} \approx 165140$ entries per level at a page size of 4 KByte.

To circumvent such large page tables, only a part of the whole address space is covered by the page table and thus is available. This limitation is not really a problem, because IA-64 implementations do not have to implement all 64 address bits in hardware, hence the virtual address space is limited by the hardware anyway. At least 51 virtual address bits and the three region number bits have to be implemented by all IA-64 implementations.

The page table of Fiasco64 has currently levels of different size. The first level contains 511 entries of 8 bytes each. This strange number is suitable, because the data structure for an address space, which consists of the first level page table itself and two 4-Byte IDs, should fit into a 4-KByte page. The one of the two IDs is the space identifier and the other the chief's space identifier (see [Liedtke, 1996] ... clans and chiefs) of the specific address space.

The second and the third level have 1024 entries each, so these two levels cover a 4-GByte region of the address space. For every second and third level table two physical continuous 4-KByte pages are needed to hold the 1024 entries (8 bytes per entry).

Index calculation for the first level is a bit tricky, in account of the region concept of IA-64 and the need to cover at least a part of each region. The calculation is done as follows:

$$i = va_{(63:61)} \ll 6 \mid va_{(37:32)} \qquad (\ll \text{ is shift left})$$

The first part ($va_{(63:61)} \ll 6$) extracts the virtual region number from the three most significant bits of the virtual address and shifts it into position. The second part

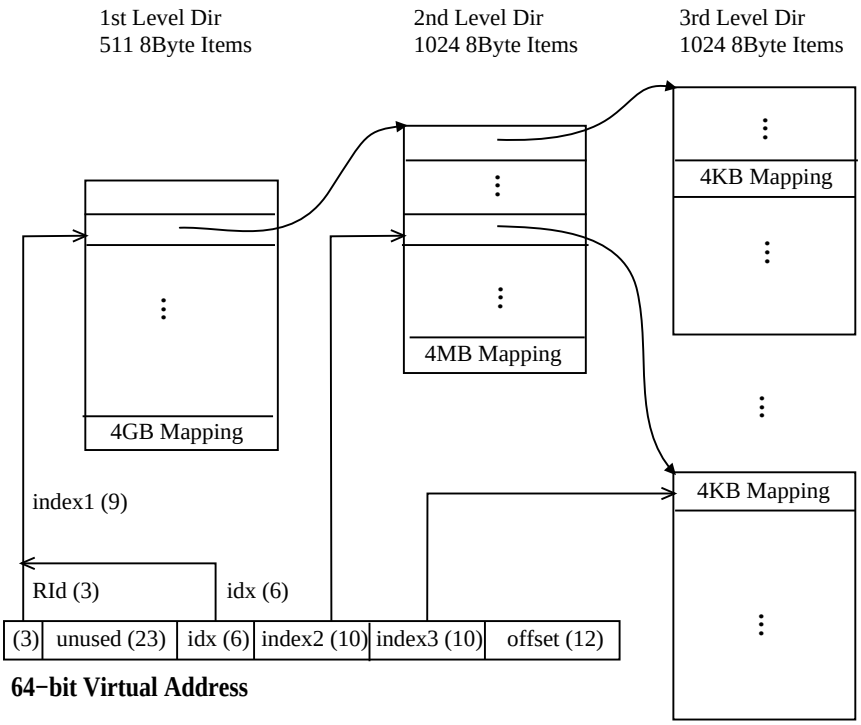


Figure 3.3: Structure of a Page table

extracts the most significant bits of the implemented virtual address.

As result every region but the last gets 64 first level entries, the last region gets only 63 entries, because in all there are only 511 entries. Every first level entry potentially covers 4 GByte of virtual address space, so the first 256 GByte of each memory region are available.

3.2.3 Mapping Database

The mapping database is used to find out which tasks have to flush[1] a specific mapping on a flush operation. The rule is that all tasks that got a mapping from a Task A must also flush the mapping if Task A removes it.

The implementation of the mapping database is taken from Fiasco and implies a limitation to 4 GByte of virtual memory for flush operations. This restriction is acceptable for the moment, because 4 GByte are sufficient for all the tested applications.

To weed out the 4-GByte limit, another implementation of the mapping database is necessary. A possible solution is the combination of the mapping database with a guarded page table like proposed in [Szmajda, 2001].

3.2.4 Kernel Memory Management

This topic is about the microkernel's memory management infrastructure. The main part here is page granular allocation of physical and virtual memory. Almost all other memory allocators in the Fiasco kernel are based on these page allocators.

[1] Flush is the operation that removes a mapping from an address space

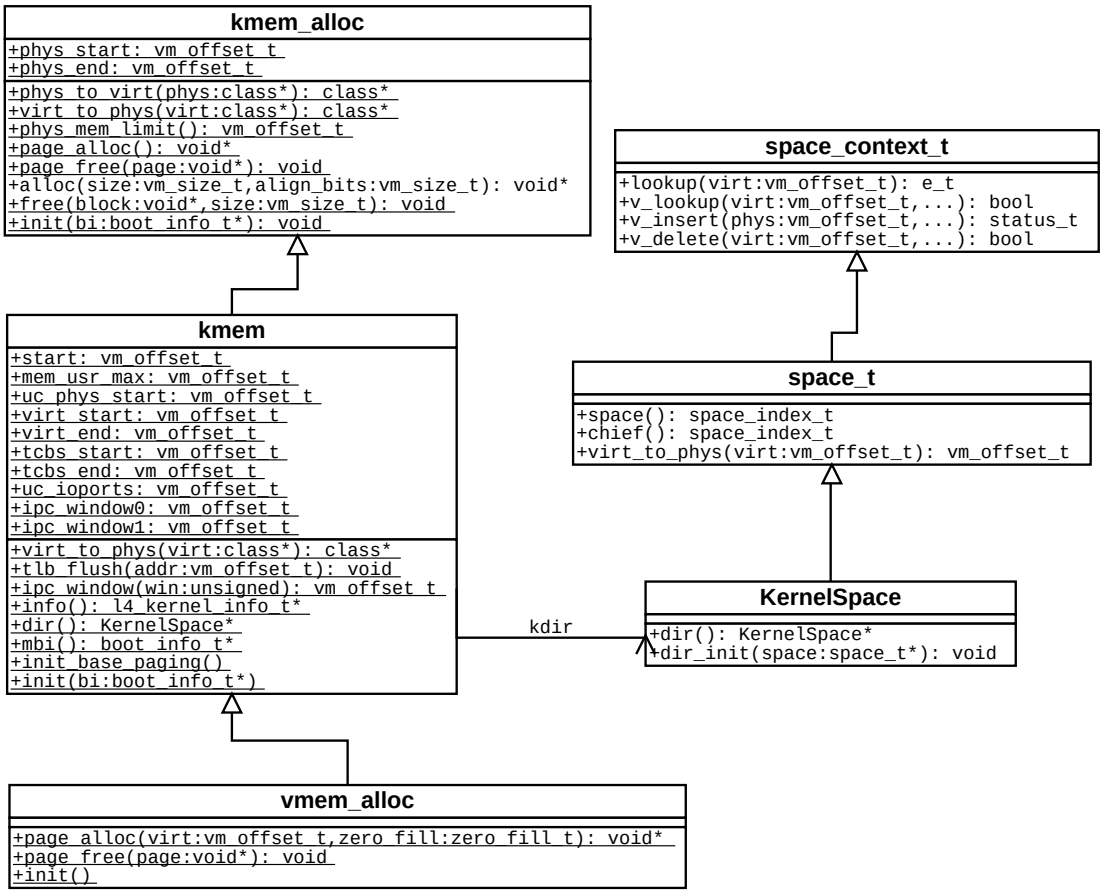


Figure 3.4: OOD of The Kernel Memory Management

Now the structure of the kernel memory management in Fiasco is explained, to make clear why a new design is needed.

The Fiasco kernel memory is abstracted by the class `kmem`, which contains some memory-related constants, like addresses of IPC windows[2], a reference to the kernel page table, some methods to initialize itself and a method called `stupid_alloc`. The method `stupid_alloc` is a memory allocator for early kernel initialization.

Further there is the kernel memory allocator, encapsulated in `kmem_alloc`, that on the one hand allocates pages in the linear mapped physical area and on the other hand has methods to allocate pages to specified virtual addresses. To do the latter it has to manipulate the kernel page table, but the page tables itself use the allocator for new second or third-level tables.

In Fiasco the page-table manipulation is made directly, means around the interface defined in `space_context_t`. But because of the full encapsulation of the page tables, this manipulations must be made through the interface methods, rather than directly. This use of `space_context_t` introduced a circular dependency between the kernel memory allocator, which inserts mappings into the page table if memory is allocated to a specific virtual address, and the page-table implementation that uses the allocator for physical memory allocation.

Another weakness is the `stupid_alloc` function, a simple implementation of a memory allocator, that could be replaced by the general kernel allocator `kmem_alloc`.

[2] are used to copy long messages from one address space to another

To remove these weaknesses I decided to redesign the kernel memory management. The first step was to separate the allocator for physical memory from the one for virtual memory. The separation breaks the circular dependency and paves the way to replace `stupid_alloc` by `kmem_alloc`, which now only manages physical memory.

Figure 3.4 on the page before shows the new design, it consists of the physical memory allocator called `kmem_alloc`, the virtual memory allocator named `vmem_alloc`, the global kernel memory definitions in `kmem`, and the page table `space_context_t`. The class `space_t` is the encapsulation of an L4 address space and `KernelSpace` a special derivation for the kernel page table. `KernelSpace` provides a method to initialize new address spaces.

3.3 Context Management

One of the main things, the microkernel has to do, is to provide user level execution contexts. The execution contexts are also called *threads* and 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.

In Fiasco, threads are a kernel entity and consist of a kernel-level part and a user-level part. In this section almost only the kernel part is of interest. The transition between user level and kernel level is described Section 3.4.

On IA-64 a context consists of the contents of the static general register, the floating point state, the current instruction pointer, the processor status, the stack pointer, and the register stack backing store pointer (see Section 2.3.4). All this state has to be preserved on a context switch, except the very huge floating point state that can be saved lazily. Please consult [Intel IA-64 V2, 2000] for a full description of all registers.

3.3.1 Thread Control Blocks (TCBs)

The kernel thread state and also the user thread state are preserved in the TCBs. A TCB, which represents one thread in the kernel, consists of the following parts:

- kernel stack
- kernel register stack backing store (see Section 2.3.4)
- thread state information:
  - current thread state (running, in IPC, etc.)
  - current kernel stack pointer, if the thread is not running
  - a reference to a floating point state buffer.
  - further information for scheduling, IPC and so on.

The layout of a TCB is pictured in Figure 3.5 on the facing page. As shown there the memory stack and the register stack use the same area. This is suitable because the memory stack grows downward and the register stack upward, with respect to memory addresses. The only problem is that stack overflows cannot be detected simply . But in a well known kernel, the maximum stack size can be calculated and so stack overflows may never occur.

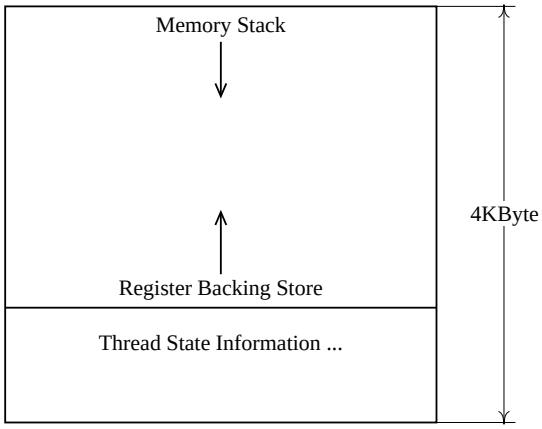


Figure 3.5: Layout of a Thread Control Block

At the moment, a TCB has a size of 4 KByte. TCB are virtually present in the *OS TCB area* (see Figure 3.2 on page 16), but the physical memory for them is allocated on demand. The allocation is triggered by a kernel page fault in the TCB area.

3.3.2 Context Switch

If the kernel provides an abstraction for contexts, it also has to switch among them, for example to provide parallel 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 target context must be restored, from their respective TCB.

Compiler-generated code spills/fills general registers to/from the kernel stack. The floating point state is saved lazily, so only the access to the floating point registers needs to be denied. The remaining state — the backing store pointer as well as the state of the register stack (see Section 2.3.4), the stack pointer, the NaT registers (see Section 2.3.3), the predicates, and the function state — have to be preserved manually. A small piece of inline assembly in `context_t::switch_to` stores this state, in a structure called `switch_stack_t`, on top of the kernel stack.

3.3.3 Non-Blocking Synchronization

I fully reused the non-blocking synchronization scheme of Fiasco. The synchronization is based on abstract *compare and swap* and *test and set* operations. These abstractions are simply adapted to IA-64 instructions.

3.4 Kernel Entry/Exit

To provide protection to programs and to the kernel itself, it is necessary to have some code that runs most privileged (the kernel) and code that runs in non privileged (user programs). In order to that the privileged code is protected from faulty or bad user programs.

To have different privilege levels is one thing, but the other is to switch among them. Such a privilege switch is also somehow a context switch, a switch from user context

to the kernel context. Therefore user thread state has to be preserved on kernel entry and restored on kernel exit.

3.4.1 Lightweight Interruption Handling

On an interruption the processor switches to the highest privilege level, saves a minimum state, switches general registers r16 to r31 to a second register bank, and transfers control to the specific interruption handler. This handler may handle the interruption with the limited resources (r16–r31) and immediately return to the interrupted code. This technique is called lightweight or efficient interruption handling. It inhibits the overhead of storing and restoring the whole processor state.

Lightweight interruption handling is something less than a full kernel entry. It is used for TLB-miss handling, for instance.

3.4.2 Full Kernel Entry

All full kernel entries that result from interruptions (even system calls via a `break`) have to be done in a lightweight interruption handler.

The entry code has to save the whole user context, to switch back to the second register bank, enable interruption collection, and then transfer control to the kernel-level handler.

The context is saved on the kernel stack of the current thread, in a structure called `thread_ret_regs_t`. If the interruption occurred in user space this structure is the first thing on the kernel stack and the register stack (see Section 2.3.4) is switched to the kernel register backing store. In the other case, the interruption occurred in kernel mode, the `thread_ret_regs_t` structure is put on top of the kernel stack and the register stack remains untouched.

The `thread_ret_regs_t` structure contains all registers that are defined as *scratch* by the software conventions. Compiler-generated code of subsequent kernel functions spills the, per software convention *preserved*, registers.

In addition to the scratch registers, `thread_ret_regs_t` contains the following:

- interrupted processor status register, *psr*
- interrupted context's instruction pointer, *iip*
- interrupted context's function state, *ifs*
- user NaT register, *unat* (see Section 2.3.3)
- previous function state register, *pfs*
- register stack control register, *rsc* (see Section 2.3.4)
- register stack NaT register, *rnat*
- register stack backing store pointer, *bsp*
- predicate registers, *pr*
- the size of the register stack's dirty partition, *loadrs*
- global pointer, *r1*

3.4.3 System Calls

User programs need to use the functionality of the operating system, for example to create a new thread or communicate to another thread, may be in another address space. System calls are the well defined interface for such user–kernel interactions.

There are two ways to do a privilege change on IA-64, and thus to implement system calls. The first one is the `break` instruction that results in a *Break Fault* (see Section 2.3.6). The other way is an `epc` (Enter Privileged Code) instruction in conjunction with special page rights. The `epc` method is potential faster, because it is executed in the normal instruction stream and prevents extra instruction serialization effort and pipeline flushes.

With respect to the targets of these work, Fiasco64 uses the suboptimal `break` instruction for now. Therefore the system calls may share the code for kernel entry and exit with involuntary interruptions, like page faults and hardware interrupts

Because system calls use the same code to enter the kernel, as interruptions, the `thread_ret_regs_t` structure is also saved onto the kernel stack. As a result the system-call handlers in the kernel may use the contents of the stored registers directly as arguments.

3.5 Boot up and Initialization

The boot sequence of Fiasco64 has a new structure, in contrast to the Fiasco startup, where the microkernel and the root pager σ_0 are loaded as raw files, and the Resource Manager (RMGR) is started as the kernel, from the boot loader's point of view. The RMGR's first stage then unpacks the ELF-encoded microkernel and σ_0, and transfers control to the microkernel.

On IA-64, the boot loader is designed to unpack not only the kernel, but also other ELF boot modules. At least σ_0, RMGR and the microkernel are loaded as executables, and control is directly transfered from the boot loader to the Fiasco64 kernel.

The kernel itself has to provide a piece of code that is either position independent or linked to the physical load address of the kernel binary. This early startup code has to map the kernel to its virtual address and switch to virtual addressing mode.

At this point of initialization no faults or traps including TLB misses must occur, because interrupt service routines are not set up.

The first action, running in virtual memory, is to set up the RSE, with its initial backing store memory, so that function calls and returns work properly and a call to `bootstrap` can be made.

In `bootstrap` the early console output is initialized and the kernel-function `startup` is called. Now `startup` starts the kernel subsystems:

1. The Uninitialised data segment (BSS) is cleaned out.
2. `kmem::init_base_paging` sets up the native kernel page-table entries.
3. `kmem_alloc::init` starts the kernel physical memory allocator.
4. `kmem::init` initializes the kernel info page.
5. `vmem_alloc::init` starts the kernel virtual memory allocator.

6. Constructors of static objects are executed.

7. The Console System (see Section 3.6) is started.

After this initialization the `main` function of the kernel is started, `main` creates the kernel thread resp. its TCB and starts `kernel_thread::bootstrap` in its context. In `kernel_thread::bootstrap` the kernel thread initialization is finished and the address spaces and initial threads of σ_0 and the root task (RMGR) are created and started. After that the kernel thread enters the idle-loop.

3.5.1 Bootinfo Structure

The Bootinfo structure is actually not only part of the kernel. It is more the interface between the boot loader and the kernel. The Bootinfo structure has to be provided either by the boot loader itself or by an intermediate layer of boot code.

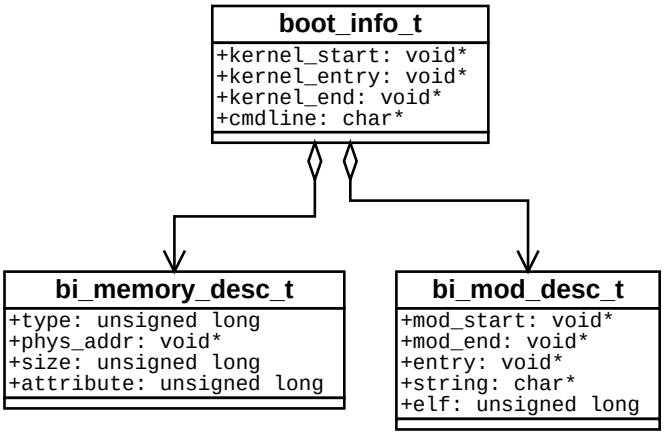


Figure 3.6: OOD of the Bootinfo Structure

As base for the design the, *Multiboot* specification (see [Ford and Boleyn, 1996]) for IA-32 is taken. It is transformed into a more general, platform independent format. Figure 3.6 shows only the general layout of the Bootinfo, the exact specification of the structures is given in the Appendix A.

3.6 Kernel Debugger

The in-kernel debugger of Fiasco is not yet ported to IA-64, only simple output functions are implemented, based on the console system.

The Console System

The console driver, which is part of the debugging system of the microkernel, had to be replaced, because it was IA-32 specific and part of the OSKit (see Section 3.8) that is dedicated mostly to IA-32 and therefore must be replaced.

The new console system is implemented in C++ and provides a simple VGA and a serial port console. At the moment only output functions are implemented, so that `printf` works properly.

3.7 Non-Kernel Programs

The L4 microkernels, like FIASCO, are not usable without a minimum of specific and trusted user tasks. The first task needed is the root pager, called σ_0, it is the first address space and by default owns all physical memory. Further there has to be a root task that own the task-creation right. All further tasks must be started by the root task, RMGR. The third non-kernel program is the boot loader that is necessary to bootstrap the operating system and after that is never needed (can be removed from the memory).

3.7.1 Boot loader

The boot loader is responsible for loading the operating system from a specific boot media and may be to hand over some platform and configuration information. In the case of FIASCO resp. FIASCO64, where no device drivers are in the kernel, the boot loader has to load all programs necessary to run the system or access a mass-storage media. On IA-32 *Grub* is used, because *Grub* is conform to the *Multiboot* standard and is able to load some modules beside the kernel.

Also on IA-64 a boot loader that is able to load multiple modules is needed. In addition there should be the option to load the data via TFTP and to use DHCP for the host configuration.

The main problem is that *Grub* is not available on IA-64; hence there was the question which boot loader to use. The candidates were *Grub*, which had to be ported, or *ELILO*, which is the standard Linux loader on IA-64, but does not support multiple modules.

After some analysis of the source code, I made the decision to the favor of *ELILO*. The advantages of *ELILO* are that it is already available on IA-64, it may also be used on IA-32[3], and the size of its code is much less than *Grub*'s. On the other hand *Grub* has a lot more features than *ELILO*, but most of them are, on IA-64, already integrated in the boot up firmware interface (EFI). The EFI already provides for example, a boot menu or the possibility to integrate new device drivers dynamically.

In order to the previous discussion, *ELILO* is taken as the base for the new dedicated FIASCO64 boot loader. It is enhanced to load multiple modules and to provide an appropriate Bootinfo structure (see Appendix 3.5.1) to the kernel. The FIASCO64 boot loader supports to load raw or ELF-encoded files from a local storage device or via TFTP. Different boot configurations can be specified in a configuration file, which has a format quite similar to *Grub's*.

3.7.2 Root Pager σ_0

The root pager σ_0 is essential for the proper work of an L4 system. It supports at least the σ_0 protocol, which provides a mechanism to request memory from it. The protocol on IA-64 is nearly the same as on IA-32, only some platform specific features, like special handling of the VGA memory area, are removed. The IA-32 protocol is described in [Liedtke, 1996].

The main work on porting σ_0 was to change data types from 32 to 64 bits and to integrate the slightly changed system-call bindings.

[3] if an Extensible Firmware Interface (EFI) implementation is provided

The major change is the initialization process of the root pager; now the Bootinfo structure is used to figure out the memory layout and usage. In addition, the new σ_0 assigns the memory of all ELF modules to their later tasks. This methodology has the favor that the root task (RMGR) has no longer to support the σ_0 protocol and the boot-time tasks can use σ_0 as their pager directly.

3.7.3 Root Task RMGR

On L4 systems the root task is per default the only task with the right to create new address spaces, and therefore has to start all further boot-time tasks. This is done with the help of the module information in the Bootinfo structure.

The term RMGR is a bit overdone on IA-64, since it no longer manages any resources beside the task creation right. The memory and hardware interrupt (IRQ) management functions, which were supported by the IA-32 version, are simply removed, because more intelligent components of the L4-Environment may do this jobs in the future.

3.8 The OSKit

The OSKit is a collection of libraries, which shall simplify the construction of operating systems. It is developed at the University of Utah, please consult [Ford and Flux Project Members, 1996] for more information.

The OSKit is mentioned here because Fiasco makes use of it for various functions. The problem is that the hardware abstractions, provided by the OSKit, are only for IA-32. So in order to get Fiasco64 running on IA-64 the platform dependent pieces of the OSKit must be replaced by appropriate IA-64 code. The following list shows only an overview of the OSKit usage in the Fiasco microkernel and the state in the new Fiasco64.

- Console and serial port driver — replaced.
- Abstractions for I/O-port access — replaced.
- Some libc[4] functions — replaced by a mini C library.
- Some IA-32 processor initialization stuff — no longer needed.
- A list based memory manager — adapted to 64 bit.
- A Address Map Manager (virtual kernel memory) — adapted to 64 bit.

[4]Library for standard C functions like `memmov`

Chapter 4

Implementation

This chapter is not a complete documentation of the implementation process, only some goodies of the implementation shall be mentioned. If you are interested in more detailed information, the source code is the right place to get it.

4.1 The Boot loader

My first experience with IA-64 was, to fiddle around with the Linux boot loader, *ELILO*. The first try to compile it, put it on a boot floppy, and just start it took me a whole day. Better the cognition that the workstation is not able to read a floppy that is formated with Linux mtools took most of the time. After the machine accepted the disk, now formated with *mkdosfs*, the first start of the self-compiled *ELILO* caused the workstation to hang. The problem was that even after a reboot the boot up firmware (EFI) hung and the machine was unusable.

The HP technician, some days later, replaced the I/O-board and the Itanium workstation worked again. The cause for the bad crash was the use of GCC version 2.9 that has problems to generate position-independent code for IA-64; GCC 3.0 solved this trouble.

But in the end it doesn't even matter — it works well!

4.2 Fiasco64 Microkernel

At the beginning it took me some time to understand the design and the functions of the Fiasco microkernel and to learn how to use the funny *Preprocessor*, from Michael Hohmuth, that truly went into a quite useful tool after some introduction.

In the whole porting the two most interesting things were the fight with the OSKit and the right handling of the Register Stack Engine (RSE). The `#include <flux/oskit...` line in a source file started to be a nightmare, because commenting them out triggered a lot of errors that must be solved in some way. Most of the OSKit functionality is now replaced by new C++ implementations, for example the new console drivers. The remaining parts of the OSKit that could be easily ported are fully encapsulated by appropriate C++ classes, for instance the the kernel memory manager uses the "list based memory manager" from the OSKit.

```
static void * kmem_alloc::low_level_alloc(vm_size_t size, int align_bits)
```

```
{
  void *ret;
  {
    helping_lock_guard_t guard(&lmm_lock);
    // The OSKit function (from the LMM-Library)
    ret = lmm_alloc_aligned(&lmm, size, 0, align_bits, 0);
  }

  if (ret) return ret;

  // out of memory -- try to find more
  morecore();
  {
    helping_lock_guard_t guard(&lmm_lock);
    ret = lmm_alloc_aligned(&lmm, size, 0, align_bits, 0);
  }

  return ret;
}
```

The Register Stack of IA-64 is a quite complex mechanism, especially in connection with context switches and kernel entry and exit. Context switches are the simpler part, because the dirty stacked registers must be flushed explicitly to the backing store (to hold the data local to their respective threads) and the backing-store pointer must be set to the target's TCB. But at a kernel entry the overhead of spilling the whole dirty partition of the register stack (see Section 2.3.4) should be removed and so only the backing store pointer is switched to the kernel backing store. This has the effect that some user-land register values are spilled to the kernel register backing store. So far so good, but now on kernel exit the right number of stacked registers has to be filled from the kernel backing store explicitly, or the user task runs into trouble — they did.

Furthermore there were sporadic page faults on the first exit to the user space, at this point some tasks tried to fill some stacked registers from below the user-space register backing store. After thousands of new `printf`s in the kernel, the encountered problem was a not properly initialized function state, which specifies the number of used stacked registers.

And last but not least there were some floating-point exceptions from within the kernel — but who uses floating point calculations in a microkernel? The exceptions could be tracked down to the `printf` function that sometimes has to put numbers onto the screen and therefore uses division and modulo to calculate the single digits. But IA-64 has neither an integer division instruction nor an integer multiplication instruction that runs on general-purpose registers. After all, a simple algorithm is used to calculate the remainder and the quotient simultaneously, without the use of floating-point registers. Multiplication is realized directly with shifts and additions, so no more floating-point registers are used in the kernel for now. To keep the compiler form using floating-point registers implicitly, special compiler switches must be used.

4.2.1 Context Switch

To switch between certain threads is one of the fundamental jobs of L4 microkernels. The following sequence is executed at every context switch that is issued by the kernel. All spills and fills of general registers are done by compiler-generated code, thus there is only the need to handle special registers and the RSE. The dirty partition of the

register stack (see Section 2.3.4) must be flushed explicitly to the backing store, to ensure data integrity.

switch_to(context target) {

1. handle time slice donation (switch to donatee if target is locked)
2. check if target is already running, if it is return.
3. check if target's kernel stack pointer is valid, if not return.
4. check the FPU usage (via the modified flags in the psr) and prohibit the access if necessary.
5. do lazy ready list enqueuing.
6. save the current thread's state to its kernel stack
   (a) general registers are spilled by compiler-generated code
   (b) flush the Register Stack to the backing store, save its state.
   (c) spill function state, predicates, and stack pointer.
7. switch to the target threads kernel stack.
8. restore the state of the target thread (contrary to the save above)
9. if necessary switch address space (see switchin_context)

}

After switching to the new TCB, address space changes have to be carried out. This is done by the following sequence. Since IA-64 provides the possibility to flush specific TLB entries, it is possible to flush the IPC windows and/or the private user regions (see Section 3.2.1) selectively.

switchin_context() {

1. check if IPC-windows require a TLB flush.
2. if switch to another address space
   (a) user regions need a TLB flush.
   (b) set page-table base register to new page table.

}

4.2.2 Kernel Entry / Exit

In the present version of Fiasco64 all full kernel entries and exits use the same code sequence to store and restore the user-level context. The flow is shown next:

saveContext() {

1. put a `thread_ret_regs_t` structure onto the kernel stack of the current thread.
2. if entry from user space:
   (a) save RSE control register and backing store pointer
   (b) switch to kernel register backing store

3. do `cover` to create a new RSE stack frame
4. save processor status and the not *preserved*[1] registers in the range from r0 to r15 (see [Intel IA-64 V1, 2000])
5. switch to register bank 1 (worked on bank 0 since this code is used in the context of an lightweight interruption handler)
6. save the remaining unpreserved registers (the range from r16 to r31)

}

To store all *non-preserved* registers even on voluntary kernel entries is quite expensive. But for now the reuse of the entry and exit code is less error prone and the values of the saved registers can be used as system-call parameters directly.

4.2.3 System Call Interface

Here the system-call handlers in the kernel are of interest, neither kernel entry nor exit are taken into account. System calls take its parameters from resp. put the results to the saved registers on the kernel stack. This has the effect that the results are available in the appropriate registers, after the return to user space. This procedure heavily influenced the register conventions for system calls, which are shown in Appendix B.

4.3 Root Pager σ_0

The porting of σ_0 was straight forward. The first step was to replace the OSKit that was used only for standard C library functions. The mini C library, also used in the microkernel itself, replaces the OSKit. The next step was to change some data types from 32 to 64 bits.

The biggest changes are applied to the initialization code that has to set up the memory map with all reserved and available regions and the right owners. So the memory of the boot-time ELF modules is assigned to their later tasks. The initialization is now done with the information provided by the Bootinfo structure rather than with the limited information in the kernel info page.

At the end I replaced the C bindings of the system calls that are slightly different to the IA-32 bindings.

4.4 Root Task (RMGR)

The Root Task should better be no longer called RMGR, because it now only the starts the ELF modules that were loaded by the boot loader. The root task for Fiasco64 is a complete new implementation and contains only the functionality to run the small test applications.

[1] by the IA-64 software conventions (see [Intel IA-64, 2001])

Chapter 5

Measurements

For now I measured only basic IPC round trip times. The measurements are made on a 733-MHz Itanium single processor workstation. The resulting values can be used to compare further development to them. The measurement results are shown in Figure 5.1. A complete analysis of the theoretical and practical IPC durations is beyond the scope of this work.

	average	minimum	maximum
same address space	2083	2038	4095
across address spaces	3088	3031	5266

Figure 5.1: IPC Round Trip Times on Itanium (in total cycles)

Chapter 6

Conclusions, Open Topics, and Future Work

The porting of Fiasco to Fiasco64 shows that it is possible to use most of the algorithms and strategies of the IA-32 microkernel on Itanium resp. IA-64. The now existent Fiasco64 kernel can be used to do further experiments on IA-64.

One open topic is to integrate the new designed components into Fiasco and merge the source trees, so that further development improves Fiasco and Fiasco64. Another thing is to optimize the critical pieces of code and to use better mechanisms for system calls, to increase the performance and reach a better usability.

Future work based on Fiasco64 can be the port of L4Linux to IA-64, to have a complete user environment for test and development. Fiasco64 is also a good platform to test new implementations of page tables or mapping databases, especially for large 64-bit address spaces.

To switch to a more recent API like L4 Version 4 (aka X2), which is more general and may help to gain the performance on IA-64, and may be even on IA-32.

Fiasco64 is further a possible groundwork for running secure applications, like electronic signing, beneath untrusted applications, for example, a web browser or a web server.

Merge of Fiasco and Fiasco64

Merging the sources of Fiasco and Fiasco64 is an important goal to pursue the development of Fiasco64. This merging process can be divided into several jobs: the back porting of newly designed components to Fiasco, the extraction and encapsulation of hardware specific abstractions, the replacement of all platform dependent types through abstract types that must be defined on a per-machine manner.

The first step, the back porting, should be an acceptable effort, because all major design decisions are made with regards to portability and flexibility.

The encapsulation of platform specific functions, which includes the elimination of the OSKit from any hardware independent sources, is the bigger task, because relevant parts are scattered over a lot of source files (e.g almost all modules use some OSKit headers).

All in all I think that about 80% of the Fiasco source code is platform independent, but the remaining 20% are spread over the whole source.

Chapter 7

Summary

The result of this work a quite acceptable base for further development on the IA-64 platform. The realtime capabilities of Fiasco64 make it useful for realtime services, like video streaming, that may run in parallel to time sharing applications.

The experiences made during the development of Fiasco64 are very helpful for future tasks, like the porting of L4Linux to IA-64 and further improvement of the performance and predictability of Fiasco64 and also of Fiasco.

Some new design decisions can be used directly in Fiasco and gain its maintainability and portability.

Acknowledgment

I like to thank all those who gave me assistance to solve the various problems during this work.

Acronyms

ABI Application Binary Interface

API Application Programming Interface

BSS Uninitialised data segment

CISC Complex Instruction Set Computing, means that there are many instructions and also very complex and time consuming instructions

CPU central processing unit

DHCP Dynamic Host Configuration Protocol, used for automatic configuration of network addresses.

DROPS Dresden Realtime Operating System

DSP Digital Signal Processor

EFI Extensible Firmware Interface, is a well defined software interface for machine bootup (see [Intel EFI, 2000, Intel PXE, 1999])

ELF Executable and Linking Format, see [ELF Spec, 1995, Dehnert, 1998]

EPIC Explicit Parallel Instruction Set Computing

Fpage Flexpage, is a region of virtual memory

IA-64 Intel 64-bit architecture

IA-32 Intel 32-bit architecture, aka Intel x86

ID identifier

IPC Inter Process Communication, also Inter Thread Communication

IRQ hardware interrupt

ISA Instruction Set Architecture

Itanium Implementation of the IA-64

IVA Interruption Vector Address

IVT Interruption Vector Table

MEPG Motion Picture Experts Group

NaT Not a Thing, is a special bit or value to mark a register as invalid.

OOD Object Oriented Design

OSKit Flux Operating System Toolkit (see [Ford and Flux Project Members, 1996])

PAL Processor Abstraction Layer, is a layer of firmware that provides an interface to processor specific functions.

RID Virtual Memory Region Id

RISC Reduced Instruction Set Computing

RMGR Resource Manager, the root task that handles further task creation etc.

RSE Register Stack Engine, is the piece of hardware that manages the register stack.

TCB Thread Control Block, the encapsulation of the kernel thread state.

TFTP Trivial File Transfer Protocol

TLB Translation Lookaside Buffer, is the buffer where the translations from virtual to physical addresses reside in.

UID Unique ID

VGA Video Graphics Adapter

VHPT Virtual Hashed Page table, either a per memory region linear mapped page table or a linear mapped and hashed page table for the whole address space.

VLIW Very Long Instruction Word

Bibliography

[Baron et al., 1990] Baron, R. V., Black, D., Bolosky, W., Chew, J., Draves, R. P., Golub, D. B., Rashid, R. F., Avadis Tevanian, J., and Young, M. W. (1990). *Mach Kernel Interface Manual*. School of Computer Science, Carnegie Mellon University, ftp://ftp.cs.cmu.edu/project/mach/doc/unpublished/manual.ps.

[Dehnert, 1998] Dehnert, J. (1998). *64-bit ELF Object File Specification, Version 2.4*. MIPS Technologies / Silicon Graphics Computer Systems.

[ELF Spec, 1995] ELF Spec (1995). *Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification, Version 1.2*. TIS Committee.

[Elphinstone et al., 1999] Elphinstone, K., Heiser, G., and Liedtke, J. (1999). *L4 Reference Manual MIPS R4x00*. School of Computer Science and Engineering The University of New South Wales.

[Ford and Boleyn, 1996] Ford, B. and Boleyn, E. S. (1996). *Multiboot Standard*. http://www.nilo.org/multiboot.html.

[Ford and Flux Project Members, 1996] Ford, B. and Flux Project Members (1996). *The Flux Operating System Toolkit*. University of Utah, Salt Lake City, http://www.cs.utah.edu/projects/flux.

[Hohmuth, 1998] Hohmuth, M. (1998). The fiasco kernel: Requirements definition. Technical report, Dresden University of Technology.

[Härtig et al., 1997] Härtig, H., Hohmuth, M., Liedtke, J., Schönberg, S., and Wolter, J. (1997). *The Performane of μ-Kernel-based Systems*.

[Intel EFI, 2000] Intel EFI (2000). *Extensible Firmware Interface Specification*. Intel Corporation.

[Intel IA-64, 2001] Intel IA-64 (2001). *Itanium Software Conventions and Runtime Architecture Guide*. Intel Corporation.

[Intel IA-64 V1, 2000] Intel IA-64 V1 (2000). *Intel IA-64 Architecture Software Developer's Manual, Volume 1: Application Architecture*. Intel Corporation.

[Intel IA-64 V2, 2000] Intel IA-64 V2 (2000). *Intel IA-64 Architecture Software Developer's Manual, Volume 2: System Architecture*. Intel Corporation.

[Intel IA-64 V3, 2000] Intel IA-64 V3 (2000). *Intel IA-64 Architecture Software Developer's Manual, Volume 3: Instruction Set Reference*. Intel Corporation.

[Intel IA-64 V4, 2000] Intel IA-64 V4 (2000). *Intel IA-64 Architecture Software Developer's Manual, Volume 4: Itanium Processor Programmers Guide*. Intel Corporation.

[Intel Itanium, 2001] Intel Itanium (2001). *Intel Itanium Processor Reference Manual for Software Optimization*. Intel Corporation.

[Intel PXE, 1999] Intel PXE (1999). *Preboot Execution Environment (PXE) Specification*. Intel Corporation.

[Liedtke, 1996] Liedtke, J. (1996). *L4 Reference Manual 486 Pentium Pentium Pro*. GMD – German National Research Center for Information Technology.

[Mosberger and Eranian, 2002] Mosberger, D. and Eranian, S. (2002). *ia-64 linux kernel, design and implementation*. Hewlett-Packard Books.

[Potts et al., 2001] Potts, D., Winwood, S., and Heiser, G. (2001). *L4 Reference Manual Alpha 21x64*. UNSW CS&E.

[Szmajda, 2001] Szmajda, C. (2001). *Calypso: A Portable Translation Layer*. UNSW CS&E.

Appendix A

Bootinfo Specifications

$flags_{(64)}$	+0
$*kernel\_start_{(64)}$	+8
$*kernel\_entry_{(64)}$	+16
$*kernel\_end_{(64)}$	+24
$*mem\_map_{(64)}$	+32
$mem\_map\_size_{(64)}$	+40
$*cmdline_{(64)}$	+48
$mods\_count_{(64)}$	+56
$*mods\_addr_{(64)}$	+64

Figure A.1: Layout of the Bootinfo Structure (`struct boot_info`)

$type_{(64)}$	+0
$*phys\_addr_{(64)}$	+8
$size_{(64)}$	+16
$attribute_{(64)}$	+24

Figure A.2: Layout of a Memory Descriptor (`struct bi_memory_desc`)

$*mod\_start_{(64)}$	+0
$*mod\_end_{(64)}$	+8
$*entry_{(64)}$	16
$*string_{(64)}$	+24
$*flags_{(64)}$	+ 32

Figure A.3: Layout of a Module Descriptor (`struct bi_mod_desc`)

Appendix B

System Call Register Conventions

IPC	*wait for id/0*	r2		r2	∼
	dest id	r3		r3	*real dest id*
	∼	r8		r8	*msg dope + cc / cc*
	∼	r9		r9	*source id*
	timeouts	r10		r10	∼
	snd descriptor	r14	→`break 0x10`→	r14	∼
	rcv descriptor	r15		r15	∼
	msg.w0	r16		r16	*msg.w0 / ∼*
	msg.w1	r17		r17	*msg.w1 / ∼*
	msg.w2	r18		r18	*msg.w2 / ∼*
	...				...
	msg.w15	r31		r31	*msg.w15 / ∼*
ID_NEAREST	∼	r2		r2	*nearest id*
	dest id	r3	→`break 0x11`→	r3	∼
	∼	r8		r8	*type*
FPAGE_UNMAP	*fpage*	r16		r16	∼
	map mask	r17	→`break 0x12`→	r17	∼
THREAD_SWITCH	*dest id*	r3	→`break 0x13`→	r3	∼
THREAD_SCHEDULE	*ext preempter*	r2		r2	old ext preempter
	dest id	r3	→`break 0x14`→	r3	*partner*
	param word	r8		r8	*old param word*
	∼	r9		r9	*time*
LTHREAD_EX_REGS	*lthread no.*	r2		r2	∼
	preempter id	r3		r3	*old preempter id*
	pager id	r9		r9	*old pager id*
	ip	r16	→`break 0x15`→	r16	*old ip*
	sp	r17		r17	*old sp*
	bsp	r18		r18	*old bsp*

TASK_NEW	*pager id*	r2			r2	∼
	dest task id	r3			r3	∼
	ip	r16	→`break 0x16`→		r16	*old ip*
	sp	r17			r17	*old sp*
	bsp	r18			r18	*old bsp*
	mcp / new chief	r19			r19	∼