Universität Karlsruhe (TH)
Institut für
Betriebs- und Dialogsysteme
Lehrstuhl Systemarchitektur

SMP for L4Ka::Pistachio/AMD64

Philipp Kupferschmied

Studienarbeit

Verantwortlicher Betreuer:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dipl.-Inf. Jan Stöss

09.08.2006

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Literaturhilfsmittel verwendet zu haben.

I hereby declare that this thesis is a work of my own, and that only cited sources have been used.

Karlsruhe, den 09.08.2006

Philipp Kupferschmied

Abstract

Multiprocessor and multicore systems become more and more popular. A modern operating system is expected to be able to deal with more than one CPU so that applications can benefit from the additional processor resources. Today's commonly used operating systems increase in size and complexity and are thus more and more error-prone. Microkernels are an approach to handle this complexity and to reduce the impact of errors by moving huge parts of the OS's functionality out of the kernel and to separated user-level applications. In this work we describe the design and implementation of SMP support in the L4Ka::Pistachio/AMD64 microkernel.

Contents

Chapter 1

Introduction

With the spreading of multicore central processing units (CPU), symmetric multiprocessing (SMP) systems have become popular for desktop systems and even notebooks. Multiple CPUs or CPU cores in one system are a way to increase system performance. In theory, an additional CPU can increase system performance by 100%. In practice, the performance benefit depends on the application workload and on how much the applications can be parallelized. To benefit from the additional CPUs in a system, the operating system must be able to deal with them. First and foremost, this requires the detection and initialization of the CPUs. Furthermore, the operating system must know the current workload for each CPU so that it can balance system load equally amongst all CPUs. It must be able to store data that is CPU-local, i.e. only of importance for a specific CPU. CPU-local queues are an example for CPU-local data.

1.1 Overview

The main goal of this thesis is to implement SMP support for L4Ka::Pistachio/AMD64. Our implementation is based on the AMD64 uniprocessor kernel and on the already existing SMP implementation for IA32. The AMD64 architecture is based on the IA32 architecture and fully compatible, but introduces a new operating mode called Long Mode. Long Mode consists of two submodes, 64 Bit Mode and Compatibility Mode. This work focuses on 64 Bit Mode, which requires both a 64 bit operating system and applications. In contrast, the second submode, Compatibility Mode, allows legacy 32 bit applications to be executed under a 64 bit operating system. 64 Bit Mode comes with several enhancements compared to the legacy IA32 operating modes. It allows virtual addresses of up to 64 bit, thus expanding the size of the virtual address space from 4 GByte to 2^{64} Byte or almost 17 Mio. TeraByte. It doubles the size of all 32 bit general purpose registers (GPR) to 64 bit and adds 8 new GPRs. Legacy segmentation functionality is largely disabled in 64 Bit mode, thus providing a flat address space. Paging is used instead for memory protection and data separation. 64 Bit mode uses a 4-level page table hierarchy for page translation.

Symmetric Multi Processing (SMP) systems consist of multiple CPUs connected to a single shared memory. The workload on the system is distributed amongst these CPUs, whereas each CPU can process every task in the system. One processor in the system is responsible for booting the system and executing initial operating system code. This processor is called bootstrap processor (BSP). The other processors in the system are called applications processors (AP). The operating system has to detect the APs and to start them up. That is, their operating mode must be changed from Real Mode to the desired operating mode, i.e. 64 Bit Mode in our case. The methods we used for CPU detection are the same as for the IA32 kernel, but the initialization works different.

The IA32 kernel manages CPU-local data such as ready queues by mapping a region of virtual memory to physical addresses different for each CPU. This is done by duplicating parts of the page table hierarchy per CPU. We wanted to adapt this concept to our implementation. In contrast to IA32, the 4-level page table hierarchy of AMD64 requires additional synchronization mechanisms to keep CPU-local page tables consistent where necessary. We implemented a method that performs synchronization at a low level, namely in the functions used to modify a single page table entry.

The performance of inter process communication (IPC) is an important criteria for the performance of a microkernel-based system. IPC is possible also between threads on different CPUs. The number of cycles (and thus the duration) needed for a cross-CPU IPC operation is a good measure to compare the performance of the IA32 kernel with the AMD64 kernel.

1.2 Organization of the Document

Chapter 2 gives some background information, starting with a brief overview about microkernels in general and especially the L4 microkernel. It then introduces the AMD64 architecture and points out differences to IA32. Finally, the concept of multiprocessing and especially SMP is explained.

Chapter 3 discusses our design and implementation. We explain the steps required to detect and start up the processors in a system. We then describe how a processor is initialized, that is, how its operating mode is switched from Real Mode to 64 Bit Mode. We introduce the idea of CPU-local memory and what we need it for. We outline the concepts of our realization of CPU-local memory. We show the steps necessary to implement CPU-local data. This covers allocation of page tables, initialization of the tables, mechanisms to keep page tables synchronized where necessary, and finally a concept to declare variables as “CPU-local” and thus to create a section of CPU-local data in the kernel binary.

In Chapter 4 we compare our implementation to the already existing IA32 SMP capable kernel. We benchmark the performance of cross-CPU inter process communication. Finally we analyze our work and make suggestions for future improvements in Chapter 5.

Chapter 2

Background

2.1 The L4 Microkernel

In contrast to a monolithic kernel, a microkernel provides only a minimal abstraction of the underlying hardware. Most system services like pagers or device drivers are implemented as user-space applications. This makes the kernel smaller and thus more flexible and maintainable.

L4 is a microkernel of the second generation, originally designed and implemented by Jochen Liedtke [5,6]. It provides two basic abstractions, namely threads and address spaces, and two basic mechanisms, inter process communication (IPC) and mapping. The former is used to establish synchronous communication between threads, the latter allows the recursive construction of address spaces. Furthermore, L4 offers special primitives for SMP systems. It allows user-level schedulers to migrate threads between CPUs. As L4 abstracts interrupts raised by the hardware with special interrupt threads, thread migration can also be used for IRQ routing, i.e. delivering an interrupt to a special CPU by migrating the according interrupt thread to this CPU.

L4Ka::Pistachio, which is the base of this work, is the latest implementation of L4 developed by the System Architecture Group at the University of Karlsruhe. It is mainly written in C and C++, with ports existing for currently 9 different architectures [7].

One of the most outstanding features of L4 is its extremely good IPC performance. IPC is possible between threads of the same address space, between threads in different address spaces, and between threads on different CPUs.

2.2 The AMD64 Architecture

In this section, we discuss some basic aspects of the AMD64 architecture, with focus on the differences to IA32. A much more detailed description of the architecture can be found in [2] and [3].

2.2.1 Operating Modes

The AMD64 architecture is designed to provide full binary compatibility with previous implementations of the x86 architecture. Thus, all operating modes supported by IA32 are also supported by AMD64, and two new operating modes are added.

The supported operating modes can be classified in 3 major classes:

Legacy Mode

Legacy mode provides binary compatibility with 16 and 32 bit x86 application- and system software. It consists of three submodes:

Real Mode In real mode, the processor supports a physical memory space of up to one megabyte. Paging is not supported. All software runs at privilege level 0. Following a power-up or reset, a processor always enters real mode.

Protected Mode In protected mode, virtual and physical memory spaces of up to 4 Gbyte are supported. Segmentation mechanisms and hardware multitasking functions are completely available, paging can be enabled. Protected mode provides four privilege levels (from 0 to 3), where 0 is the highest privilege. Operating system code normally runs at privilege level 0, while application software runs at level 3.

Virtual 8086 Mode This mode allows software written for 8086, 8088, 80186 or 80188 to be executed as a privilege level 3 task in a protected mode environment. It is not of importance for the work discussed here.

Long Mode

The AMD64 architecture introduces Long Mode, which consists of two submodes:

64 Bit Mode 64 Bit Mode provides support for 64 bit operating systems and applications. There are several enhancements compared to IA32:

- virtual addresses of up to 64 bit, depending on the processor implementation. This results in a virtual address space of up to 2^{64} bytes.
- eight new GPRs
- width of GPRs expanded to 64 bit
- eight 128 bit SSE registers
- 64 bit instruction pointer (RIP)
- a new RIP-relative addressing mode
- flat-segment address space with single code-, data- and stack space

64 Bit Mode is enabled or disabled by the operating system on an individual code segment basis. However, the legacy segmentation mechanism is largely disabled in 64 Bit Mode: Segment basis and limits are ignored, so every segment covers the entire virtual address space. The only exceptions from this rule are the FS and GS segments, whose base addresses are used. A 64 Bit toolchain and operating system are required to build and run programs in 64 Bit mode.

Compatibility Mode Compatibility Mode allows legacy 16 or 32 bit software to be executed under a 64 bit operating system without recompilation. In this mode, applications can access only the first 4 Gbyte of virtual address space. Segmentation functions are the same as in legacy x86 protected mode. Like 64 Bit Mode, Compatibility Mode is enabled on an individual code segment basis. To an application, Compatibility Mode looks like a legacy Protected Mode environment, although from the viewpoint of the operating system, Long Mode mechanisms are used for page translation, exception and interrupt handling and system data structures.

Both 64 Bit Mode and Compatibility Mode require a 64 bit operating system.

System Management Mode

System Management Mode is designed for system control activities like power management features. It is mainly important for BIOS code and specialized low-level device drivers and thus mentioned here only for the sake of completeness.

2.2.2 Memory Management

Segmentation

Segmentation was designed to provide isolation between tasks and the data used by these tasks in memory. Each application can use a number of segments on its own, and the hardware ensures that all memory accesses are within the current segment limits.

The AMD64 architecture supports all forms of legacy segmentation. But as segmentation is rarely used nowadays, it is nearly completely disabled in 64 bit mode, which provides a flat memory model instead and uses paging mechanisms for task protection.

Paging

The x86 paging mechanism allows operating systems to create different address spaces for each application, which are commonly referred to as virtual address spaces. Virtual address spaces do not only avoid unwanted interference between tasks, but also provide each task its own range of addresses, not depending on the actual layout of code and data in physical memory. The translation from virtual to physical addresses is done by means of page tables. In 64 bit mode, AMD64 uses a 4-level page table hierarchy to translate virtual addresses of up to 64 bit into physical addresses of up to 52 bit. In contrast, IA32 architecture uses a 2-level page table hierarchy.

The table at the top of the hierarchy is called page map level 4 (PML4). Each of its entries can point to a table at the next lower level, called page directory pointer (PDP), from where each entry might point to a page directory (PDIR). A bit of each entry indicates if 2 Mbyte or 4 kbyte pages are used. In case of 2 Mbyte page translation, the corresponding PDIR entry points directly to a physical page, whereas in case of 4 kbyte translation the entry points to another translation table, called page table (PTAB). The PTABs are the lowest level of the translation hierarchy, entries point to 4 kbyte pages of physical memory.

Current implementations of the AMD64 architecture use virtual addresses of 48 bit, thus bit 48 through 63 are a sign extend of bit 47. Bits 47 through 39 index into the PML4, bits 38 through 30 into the PDP, bits 29 through 21 into the PDIR, and, if 4 kbyte translation is used, bits 20 through 12 into the PTAB. Bits 11 through 0 provide the byte offset into the physical 4 kbyte page. In case of 2 Mbyte translation, bits 20 through 0 provide the byte offset into the physical (4 Mbyte) page.

As 9 bits are used for each index, each page table consists of $2^9 = 512$ entries. An entry is 8 byte long, such a page table fits exactly into a 4 kbyte page.

2.3 Symmetric Multiprocessing

Multiprocessing in general refers to computer systems consisting of more than one processor. Symmetric multiprocessing (SMP) refers to multiprocessor systems where all processors can execute every task in the system. In contrast, asymmetric multiprocessing refers to systems where each processor is assigned a specific task. On SMP systems, the operating system can relatively easy balance the workload of the processors by moving tasks to another processor when necessary. Because the CPUs are connected to a single shared memory, communication between threads on different CPUs can be established relatively easy. However, there is also the need for a basic notification mechanism between CPUs to let one CPU wakeup or interrupt another. On IA32 and AMD64 systems, hardware provides local advanced programmable interrupt controllers (APIC), one per CPU. Each local APIC has a unique APIC id, which also identifies its correspondent processor. A local APIC has a number of memory mapped registers which can be read and written by system software. Thus, the local APICs can be programmed to trigger inter processor interrupts (IPI), which can be addressed to a specific CPU in the system or broadcast to multiple CPUs. There are two special forms of IPIs, the INIT IPI and the STARTUP IPI (SIPI). A sequence of INIT IPIs and STARTUP IPIs is used to start up a CPU. As APICs are not AMD64 specific, we could adapt the already existing code with only very few modifications.

2.3.1 Multiprocessor Initialization

On system startup, one processor is determined to be the bootstrap processor (BSP). The BSP is responsible for running BIOS initialization code and later booting the operating system. The other processors in the system are referred to as application processors (AP). On AMD Opteron systems, the processor connected directly to the HyperTransport I/O-Hub becomes BSP. After having executed BIOS initialization code, the APs are halted again. According to [4], the BSP can start up an AP by sending an INIT IPI and two SIPIs with a short delay between the IPIs. The AP starts code execution in real mode. The startup code can change the operating mode e.g. to Protected Mode or to 64 Bit Mode.

Chapter 3

Design and Implementation

3.1 Processor Detection and Startup

We assume the BSP to be set to Protected Mode by the bootloader when it begins with execution of L4 code. The BSP then executes kernel code that enables paging and changes the BSP's operating mode to 64 Bit Mode. The kernel parses the ACPI-tables to find local APICs (and thus CPUs) in the system. It then remaps the APIC-registers to a predefined location. Now the BSP can start the first AP. It waits until the AP has performed its own initialization before it starts the next (if any).

We use two spinlocks for processor startup: The first ensures that all APs are started sequentially. The BSP waits at the lock after it started one of the APs. This AP is responsible for releasing the lock again so that the BSP can continue to start the next AP. The second lock is used as a barrier to synchronize time stamp counters. Every AP waits at this lock after it has been initialized. When all APs are started, the BSP releases the lock so that all processors continue execution. Directly after the barrier is passed, every processor sets its time stamp counter to zero.

3.2 Processor Initialization

BSP startup code calls a function named *init_paging* that enables and activates paging and 64 Bit Mode. This function jumps to the actual L4 startup code (*startup_system*) which initializes kernel memory, data structures, and the APs. Each AP also uses *init_paging* to activate 64 Bit Mode, but an AP must not perform the kernel initialization again. So we must be able to detect if *startup_system* was called by the BSP or by an AP. We do this by passing a parameter from low-level startup code (which is different for BSP and APs) to *init_paging* and from there to *startup_system* using the according calling conventions. During this sequence of function calls, the processor operates in Protected Mode as well as in 64 Bit Mode. Thus, calling conventions both for IA32 and AMD64 must be used. On IA32, all arguments are passed to a function on the stack, while on AMD64, registers are used for the first arguments, as described in [1]. We pass only one argument, for which we use the *%edi* register. Depending on the argument passed, *startup_system* either continues executing the initialization code for kernel data structures (if running on the BSP) or calls a function to complete the initialization of the APs (if running on an AP).

3.3 CPU-Local Data

A page table in the page table hierarchy which is used only by a single CPU is referred to as CPU-local pagetable. A page table that is used by all CPUs we call a global pagetable.

Each CPU needs to store some data that is important only for itself, but not for the other CPUs. For example, each CPU has its own runqueue and its own idle thread. A possible approach to store this data would be to assign a different region of virtual memory for each CPU. Whenever accessing CPU-local data at runtime, an explicit address calculation would be required to find out the address of the data for the specific CPU. To avoid this overhead, we chose another solution. Instead of having different locations of virtual memory for CPU-local data, we decided to have only one CPU-local memory region which is mapped to different physical addresses depending on the processor currently performing the access. This way we can avoid explicit address calculations or table lookups at runtime - the entire translation is performed by page translation mechanisms that would by used anyway. We have to pay for this comfort with additional page tables and synchronization overhead.

To achieve this goal, we keep a PML4 for each CPU. Some entries of every PML4 can now be used to point to CPU-local PDPs, from where some entries can point to CPU-local PDIRs. Synchronization is required for entries that do not point to CPU-local subtables, but to global tables. That is, whenever an entry inside a CPU-local table pointing to a global tables changes, the new value must be copied to the corresponding CPU-local tables of all other CPUs. This avoids duplication of global page tables, thus reducing memory consumption.

The operating system must know for each page table entry if it points to a global or to a local subtable and if this entry must be synchronized. Entries that point to CPU-local subtables must not be synchronized, but entries pointing to global tables require synchronization only if they are part of a CPU-local table. The format of a page table entry is defined by the hardware, so additional information can not be placed arbitrarily inside an entry. But on AMD64, each entry also has three bits freely available to system software, i.e. not interpreted by the MMU hardware in any way. We decided to use two of these bits for our purposes, one to indicate if an entry points to a CPU-local subpage (the CPU-local bit) and one to indicate if an entry must be synchronized (the sync-bit).

The CPU-local memory region we use is determined at link time, so all data inside the region is static and thus part of the kernel binary. This simplifies the handling of the CPU-local memory region, because no mechanisms for dynamic allocation are required, and the size of the region does not change at runtime. The memory region lies inside the kernel area of the virtual address space and is not accessible by the user.

We use the last PML4-entry to point to a CPU-local PDP. The last entry of each CPU-local PDP in turn points to a CPU-local PDIR, all other entries are synchronized. This is a result from the CPU-local region being part of the kernel area, which is covered by these entries. We use 2 Mbyte pages for CPU-local data to avoid having CPU-local PTABs. An entry inside a CPU-local PDIR points either to a CPU-local 2 Mbyte physical page or to global data. The latter case can occur because only a few entries inside a CPU-local PDIR are used for CPU-local data. All other entries will thus point to global data, i.e. a global PTAB or a global 2 Mbyte page of physical memory.

To simplify the initialization of the page table hierarchy, we decided to limit the maximum size of the CPU-local region to 1 Gbyte (this is the amount of memory covered by a complete PDIR or one PDP entry). This is far enough for current purposes and requires only one entry of each CPU-local PML4 and of each CPU-local PDP to point to CPU-local subtables. As a result, we have n CPU-local PDPs and n CPU-local PDIRs, where n is the number of supported CPUs.

3.3.1 Page Table Allocation

On system startup, the BSP operates on a preliminary PML4 and later allocates memory for the real PML4s, one per CPU. When the last entry inside the PML4 of CPU0 is set, it is marked as CPU-local by setting its CPU-local bit (bit 10). After that, memory for all CPU-local PDPs is allocated at once so that these PDPs will be located sequentially in memory. All entries of the CPU-local PDPs are initialized by setting the sync bit (bit 9) except the last of each PDP, which is initialized by setting the CPU-local bit. Then the last entry of the PML4 of CPU0 is set to point to the first of the newly allocated PDPs.
The allocation of the CPU-local PTABs works similarly: When the last entry inside the PDP of CPU0 is set (which has its CPU-local bit set), all CPU-local PDIRs are allocated. The sync bit is set for every entry of the CPU-local PDIRs.

3.3.2 Page Table Initialization

Each AP is responsible for setting up its own CPU-local part of the page table hierarchy. As the CPU-local tables are already allocated by the BSP and synchronization is performed, there is not much work left. First, the AP has to set the last entry of its own PML4 to point to its already allocated CPU-local PDP. Then it sets the last entry of its CPU-local PDP to point to its CPU-local PDIR. For both entries, the sync bit is set to zero and the CPU-local bit is set to one, because entries pointing to CPU-local subtables must not be synchronized.

Finally, pages of physical memory must be allocated and the corresponding entries inside the CPU-local PDIR must be set to point to those newly allocated pages. For each entry, the CPU-local bit is set and the sync bit is cleared.

The already initialized CPU-local data structures of CPU0 are copied afterwards so that the AP does not operate on uninitialized data. From then on, all modifications a CPU makes to CPU-local data are visible only to itself.

Figure 3.1 shows an example for 2 CPUs with 2 physical pages for each CPU. CPU-local page tables are completely initialized, with the last element of each CPU-local PML4 and PDP pointing to a CPU-local PDP or PDIR, respectively. The dashed arrows indicate pointers to global page tables or global pages of physical memory.

3.3.3 Page Table Synchronization

As can be seen above, not all entries inside the CPU-local pagetables cover addresses of the CPU-local memory region. That is, there are entries inside CPU-local tables which point to global tables of the next-lower level. As mappings might change at runtime inside the CPU-local PDPs and PDIRs, synchronization between those tables is required. The sync bit of each entry is used to determine if an entry must be synchronized whenever a mapping is added or modified. Synchronization is performed eagerly, that is an entry for a new or modified mapping is copied to all other CPU-local page tables immediately.

Synchronization mechanisms are all implemented in the class that represents a single page table entry. A programmer writing code that modifies page tables using the functions provided by that class has not to deal with synchronization. This advantage

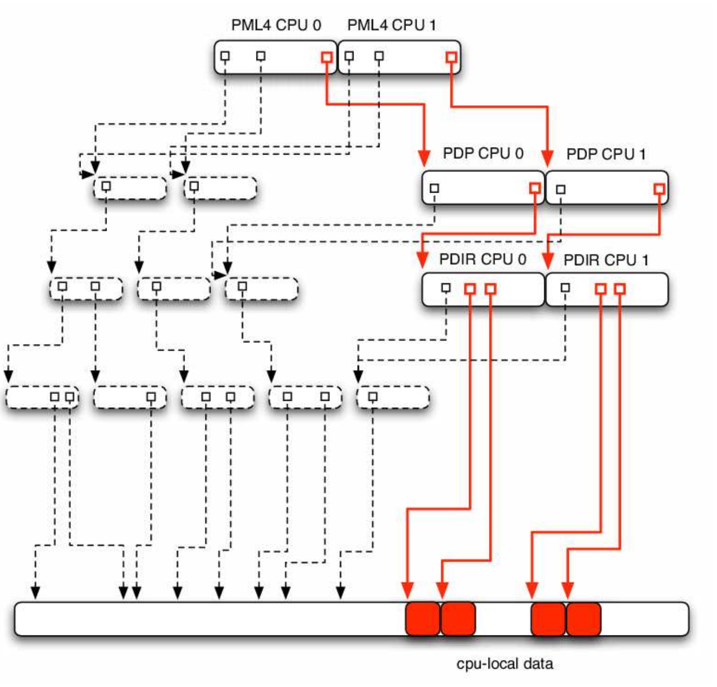

Figure 3.1: Page table hierarchy for CPU-local data.

is payed with a higher overhead, because every modification of a page table includes a check for the necessity of synchronization. Furthermore, when a new value is written to a page table entry, the sync bit of the previous value must be preserved because the necessity for synchronization does not change at runtime (as the CPU-local memory region itself does not grow or shrink). This requires an additional read operation before the new value can be written.

Synchronization itself is performed by checking if the sync bit is set and copying the entry into the other CPU-local pagetables, if necessary.

3.3.4 The CPU-Local Memory Region

Our approach does only allow static data to be CPU-local. The declaration of a CPU-local variable looks like this:

```
type varname UNIT("amd64.cpulocal");
```

This instructs the linker to put the variable *varname* into the CPU-local section of the kernel binary. The number of variables declared as CPU-local defines the size of this section. The start address of this section is also determined by the linker, depending on the size and location of the preceding sections. We ensure that it lies in the uppermost Gbyte of virtual memory, as described above. Furthermore, we instruct the linker to align the lower and upper boundary of the section at 2 Mbyte. This section thus determines size and position of the CPU-local region in virtual memory.

Chapter 4

Evaluation

IPC is the mechanism for communication between threads, which can reside on different CPUs. To IPC between threads on different CPUs, we refer to as X-IPC. We compared the performance of our implementation with the IA32 kernel by measuring the duration of X-IPC. We used a program called `pingpong`, which creates two threads, one on CPU0, the other on CPU1. The thread on CPU0 (the ping thread) sends an IPC message to the thread on CPU1 (the pong thread), which in turn replies with another message. This is repeated several times to calculate the average cycle count needed for the X-IPC operation. `pingpong` benchmarks the performance for IPC messages of different sizes, starting with 0 messages registers (MR) and increasing the length by 4 MRs up to a message size of 60 MRs.

We used two different systems to benchmark the performance of both the AMD64 and the IA32 kernel. The first is an AMD Opteron system with two CPUs, 1.8 GHz each. The second is a Pentium D/830 system with a 3.0 GHz Dual Core CPU with Intel's EM64T architecture. EM64T is mostly compatible to the AMD64 architecture.

Figure 4.1 shows the results on the Opteron system, Figure 4.2 shows the results on the Pentium system. As can be seen, the performance of our implementation is worse than the performance of the IA32 kernel.

To find out the reason for this performance difference, we analyzed the phases of an X-IPC operation. The CPU executing the (current) sender thread we refer to as SRC CPU, the CPU that the receiver thread runs on we refer to as DST CPU. The X-IPC operation works as follows:

- A sequence of IPIs is used for synchronization between SRC and DST CPU before the actual message transfer takes place:
  - The SRC CPU triggers an IPI to the DST CPU and goes idle
  - The DST CPU receives the IPI and replies with an IPI back to the SRC CPU
  - The SRC CPU receives the IPI, the sender thread wakes up again
- The sender thread starts the IPC transfer
- The SRC CPU notifies the DST CPU with another IPI when the transfer is complete and idles again
- The DST CPU receives the IPI and the receiver thread of the IPC message wakes up to process the message.

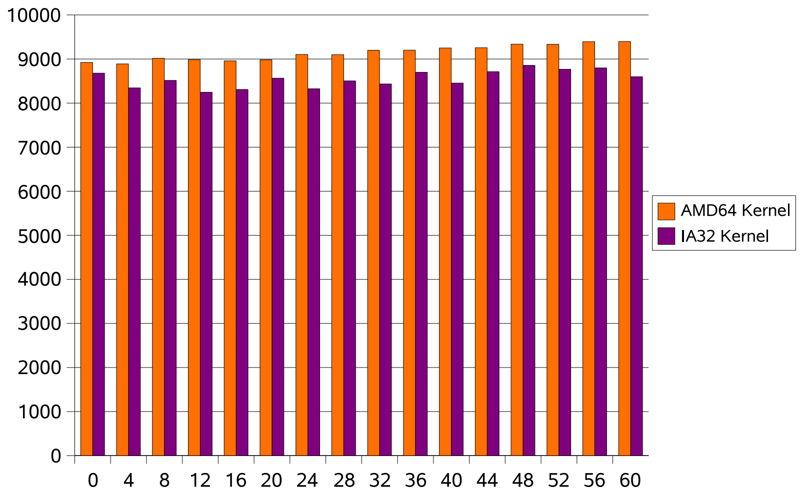

Figure 4.1: X-IPC Performance (Opteron)

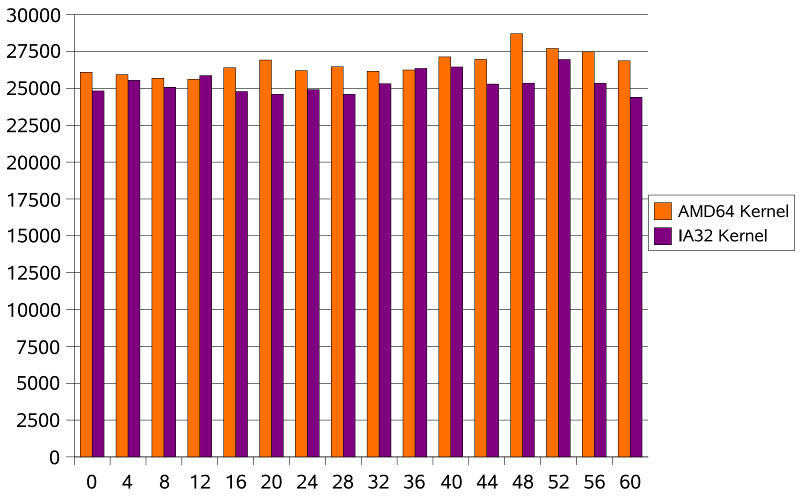

Figure 4.2: X-IPC Performance (Pentium D)

The tracebuffer is a region in kernel memory which is used for logging system events. For each event, the name of the event is stored together with the timestamp counter (TSC) value when the event occurred. The time differences between events can thus be derived from the traces. We analyzed tracebuffer output to find out the duration of the phases of an IPC operation. An X-IPC operation has 14 different tracebuffer-events, namely:

1. sys_ipc (SRC CPU) - set directly after calling the IPC syscall
2. XCPU_IPC_SEND (SRC CPU) - set if a cross-CPU operation is necessary (i.e. when the receiver is on a different CPU)
3. SMP_IPI_SEND (SRC CPU) - set before triggering an IPI from the sender's CPU to the receiver's CPU
4. switch sender ⇒ IDLETHREAD (SRC CPU) - set after triggering the IPI
5. SMP_IPI_RECEIVE (DST CPU) - set in the interrupt handler for the IPI
6. SMP_IPI_SEND (DST CPU) - set before triggering an IPI from the receiver's CPU to the sender's CPU
7. SMP_IPI_RECEIVE (SRC CPU)
8. switch IDLETHREAD ⇒ sender (SRC CPU) - set when sender thread wakes up again
9. STARTING IPC TRANSFER from sender to receiver (SRC CPU) - set when the actual transfer of the message starts
10. XCPU_IPC_SEND_DONE (SRC CPU) - set when the transfer is completed
11. SMP_IPI_SEND (SRC CPU) - set before triggering an IPI from the sender's CPU to the receiver's CPU to notify it that the transfer has been completed
12. switch sender ⇒ IDLETHREAD (SRC CPU)
13. SMP_IPI_RECEIVE (DST CPU)
14. switch IDLETHREAD ⇒ receiver (DST CPU)

The events and the corresponding TSC values can be used for comparing the performance of the AMD64 and the IA32 kernel, as they are the same for both kernels. We analyzed the performance of an IPC from CPU0 to CPU1 and the performance of an IPC from CPU1 to CPU0 separately, both for the AMD64 and the IA32 kernel. Figure 4.3 shows the result of the AMD64 kernel on an AMD Opteron system, Figure 4.4 shows the results for the IA32 kernel on the same system. Normally, one would expect that the costs are symmetric, but as can be seen, the TSC values differ depending on whether the IPC was sent from CPU0 to CPU1 or vice versa. This phenomenon also occurs on the Pentium D system, as can be seen in Figure 4.5 and 4.6. We also modified `pingpong` and migrated the ping thread to CPU1 instead of the pong-thread. This made almost no difference, especially the costs for an IPI from CPU1 to CPU0 remained higher then the costs for an IPI from CPU0 to CPU1. Currently, we do not have an explanation for this asymmetric behavior.

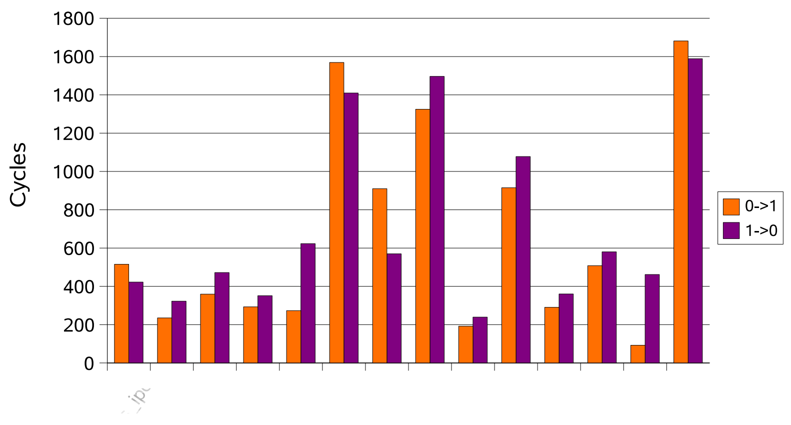

Figure 4.3: X-IPC Phases, AMD64 Kernel, Opteron

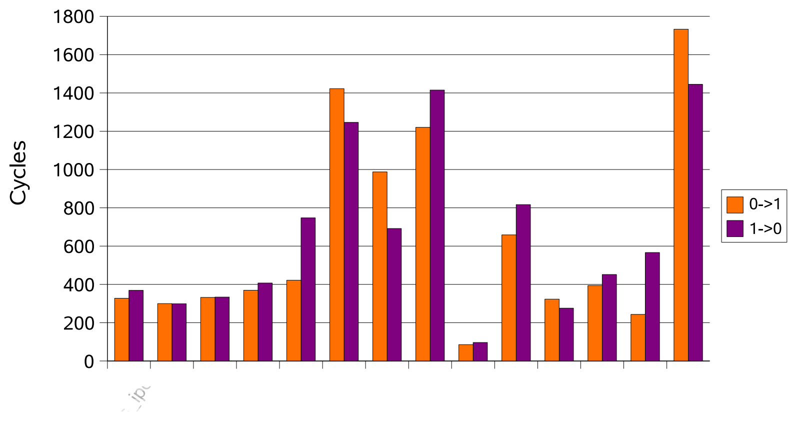

Figure 4.4: X-IPC Phases, IA32 Kernel, Opteron

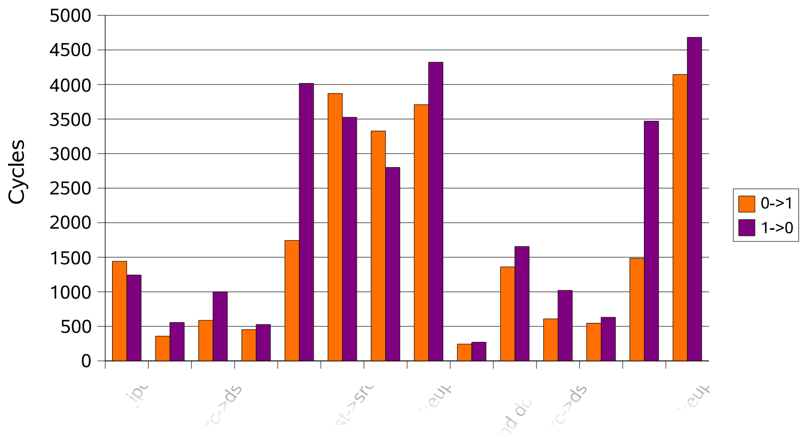

Figure 4.5: X-IPC Phases, AMD64 Kernel, Pentium D

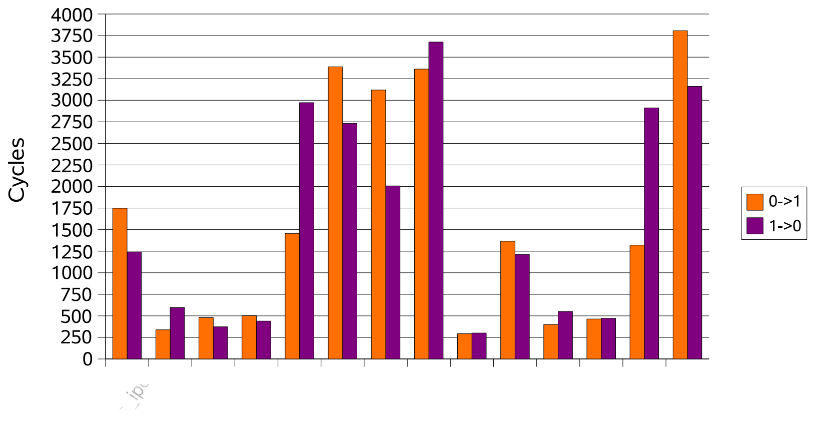

Figure 4.6: X-IPC Phases, IA32 Kernel, Pentium D

To complete our investigations, we averaged the two values for each event. The results are diagrammed in Figure 4.7 and Figure 4.8. As expected, the costs are higher on the AMD64 kernel for almost all events. The exceptions are not the same on the Opteron and on the Pentium D system. For example, on the Pentium D, `sys_ipc` is faster on the AMD64 kernel as on the IA32 kernel, while on the Opteron system, the AMD64 kernel needs more cycles. This makes it difficult to find an explanation without further analysis.

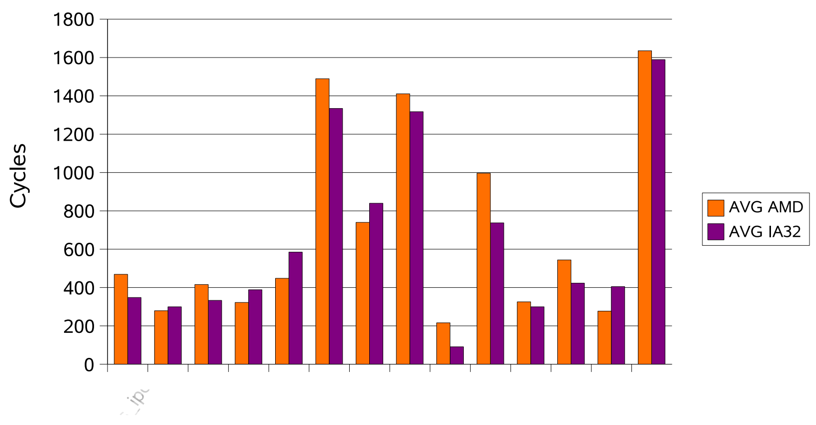

Figure 4.7: X-IPC Phases, AMD64 vs. IA32 Kernel, Opteron

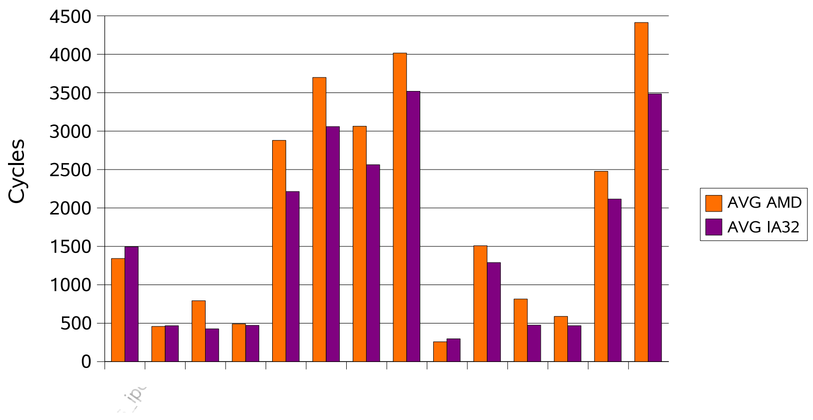

Figure 4.8: X-IPC Phases, AMD64 vs. IA32 Kernel, Pentium D

Chapter 5

Conclusion

The primary goal of this thesis was to add SMP support for L4Ka::Pistachio/AMD64. We used the already existing IA32 implementation as a template, porting the code and adapting it where necessary. The resulting implementation was tested on a Opteron system with two CPUs, a dual-core Pentium D system, and in the Simics simulator.

5.1 Suggestions for Future Work

Benchmark results of X-IPC are worse than the results on IA32. During this thesis, we were not able to find out the reasons for this behavior. More detailed performance analyses are required to detect potential bottlenecks.

Next, the implementation of CPU-local memory, especially the synchronization between CPU-local page tables, might be done better. First of all, two of three free bits in a page table entry are needed by our implementation, which could be a problem if more information would have to be stored for each page table entry. A possible solution would be to use all three bits together, thus being able to represent $2^3 = 8$ different states. This of course only makes sense if the states exclude each other.

To reduce the synchronization overhead we described above, lazy synchronization might be possible at least when establishing new mappings. Thus, a new mapping would be inserted only in the page table hierarchy of CPU0. If any other CPU would try to access the corresponding memory location, it would raise a page fault and then copy the entry or entries from the hierarchy of CPU0. Page fault handling would become more complicated this way. In contrast to establishing new mappings, modifications of already existing page table entries can not be handled lazily, as a CPU accessing a mapping that has been modified on the page table hierarchy of CPU0 would not generate a pagefault but instead access the wrong memory location without noticing.

Another feature that could be added is the dynamic allocation of CPU-local memory. This would especially be useful if user level applications were allowed to use CPU-local memory for themselves.

AMD Opteron systems are NUMA capable. NUMA stands for Non-Uniform Memory Access and refers to systems where each processor has its own local memory which can be accessed faster than non-local memory [8]. Implementing NUMA support for L4 might be a further step to improve performance and scalability on systems with multiple CPUs.

Bibliography

[1] *System V Application Binary Interface AMD64 Architecture Process or Supplement Draft Version 0.96*, 2005.

[2] AMD. *AMD64 Architecture Programmer's Manual Volume 2: System Programming*, February 2005.

[3] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon 64 and AMD Opteron Processors*, October 2005.

[4] Intel. *MultiProcessor Specification Ver 1.4*, May 1997.

[5] Jochen Liedtke. On μ-kernel construction. In *Proceedings of the 15th Symposium on Operating System Principles*, pages 237–250, Copper Mountain, CO, December 3–6 1995.

[6] Jochen Liedtke. Toward real μ-kernels. 39(9):70–77, September 1996.

[7] University of Karlsruhe System Architecture Group. *http://l4ka.org*.

[8] Wikipedia. *http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access*.