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

Compatibility Mode Support for L4Ka::Pistachio/AMD64

Sebastian Reichelt

Studienarbeit

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

December 7, 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 7. Dezember 2006

Sebastian Reichelt

Contents

Figure 1: AMD64 Operating Modes

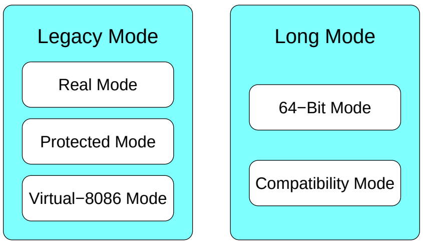
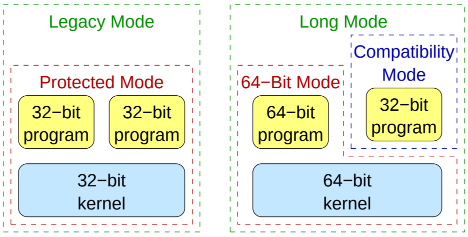


Figure 2: Usage of AMD64 Modes



1 Introduction

The AMD64 processor architecture [1], also known as Intel 64 [2], has started to replace the traditional IA-32 design in servers and personal computers. One reason for this success is the high degree of backward compatibility to IA-32 that AMD64 provides. In „Legacy Mode", AMD64 processors show the same behavior as IA-32 processors in most cases (see Figure 1), and therefore software written for IA-32 can be run unmodified (including operating systems, see Figure 2). Operating systems specifically written for AMD64 can switch to „Long Mode" and use 64-bit registers and addresses. To execute programs compiled for IA-32, they may activate „Compatibility Mode", a sub-mode of Long Mode. This causes a temporary transition to the 32-bit or 16-bit instruction set and enables operating systems to simulate a 32-bit or 16-bit environment.

L4Ka [10] is a microkernel developed at the University of Karlsruhe. In addition to the original IA-32 implementation, an AMD64 port exists. The objective of this study thesis was to implement Compatibility Mode support for the AMD64 port of L4Ka, to achieve binary compatibility with existing programs compiled for L4/IA-32.

Since L4Ka is designed for minimum kernel size and very fast IPC, the main goal of the design and implementation was to achieve good performance of IPC between all types of threads, while keeping the amount of code added to the kernel minimal. Analysis showed that aiming for good IPC performance or code minimality leads to different design alternatives. We implemented the solution that maximizes IPC performance, showed that the implementation performs well indeed, and that the complexity added to the kernel is acceptable.

2 Motivation

The simplest way to run IA-32 applications on an AMD64 system is to use a kernel compiled for IA-32 and let it operate in Legacy Mode. This approach does not require any additional programming effort, and usually even performs better (see Section 6.2). However, in many scenarios it is not sufficient:

- Porting applications from IA-32 to AMD64 can improve performance in some cases [3, Section 1.1]. If the AMD64 kernel supports IA-32 applications, performance-critical applications can be ported to AMD64, whereas others do not have to be modified.
- The number of pure AMD64 systems, consisting of an AMD64 kernel and applications compiled for AMD64 64-bit Mode, is likely to increase in the future. However, even in such environments, users will want to run existing IA-32 applications, for example if no AMD64 ports are available.
- In the L4Ka virtualization project, L4Ka acts as a host for other guest operating systems (e.g. Linux) [11]. If a 64-bit guest operating system attempts to run a 32-bit user application, L4Ka will need to run the guest application in Compatibility Mode.

3 Related Work

Support for running applications compiled for IA-32 is available in AMD64-targeted Linux [14] and BSD kernels (at least NetBSD [13]).

IA-64 is another architecture with similar support for running IA-32 binaries [7]. For the IA-64 target of the L4Ka microkernel, an experimental implementation exists as part of a diploma thesis [15].

In Section 4.1, we will analyze and compare how these implementations are designed, investigate on other design alternatives, and decide on a solution that meets the design goals.

4 Design

Apart from the straightforward task of activating Compatibility Mode in the processor, the kernel needs to be extended in several ways to support running existing 32-bit user applications. The main reason is that the IA-32 and AMD64 kernel interface of the L4 microkernel are different. Specifically, data structures exported by the kernel differ in word size and other aspects, and different registers are used for arguments of system calls. The AMD64 kernel, in turn, currently expects applications to contain 64-bit executable code. For example, the kernel debugger decodes certain instructions to perform special actions such as entering debugging mode or waiting for a key press.

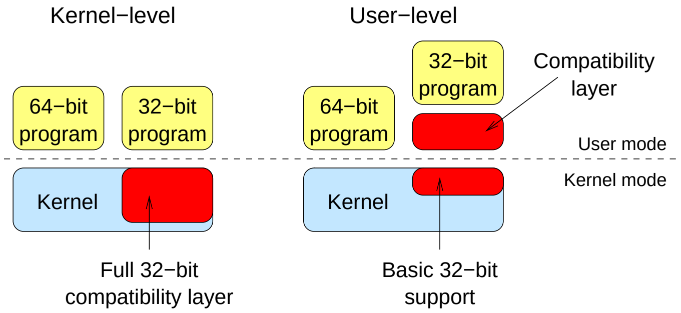
4.1 Kernel vs. User Level

There are basically two design alternatives: a user-level and kernel-level approach (see Figure 3). They have different implications on complexity and performance, the most notable impact is the implementation of system calls (see Figure 4).

1. The kernel can be designed to treat IA-32 and AMD64 programs more or less equally. The kernel presents itself to AMD64 programs with an AMD64 interface, and to IA-32 programs with an interface compatible to the one provided by the IA-32 kernel. Whenever the kernel communicates with a user-level program, it needs to determine the type of program and use the correct version of the interface. From the user's point of view, there is practically no difference between the two types of programs. The Linux kernel takes this approach: It implements the IA-32 system call interface inside the kernel, as a wrapper that converts the arguments and calls the actual system call functions designed for AMD64 (see Linux kernel source code [14], version 2.6.17, file `arch/x86_64/ia32/sys_ia32.c`).

2. Alternatively, the kernel may provide an AMD64 interface only, since the interface required by IA-32 binaries can be implemented by a compatibility layer that does not necessarily need to be executed in kernel mode. This solution is similar to user-level software running programs written for a different operating system (e.g. WINE). Translation of data structures is performed at user level, which is possible in L4 because system calls are not invoked directly but by calling a kernel-supplied user-mode stub. However, the kernel needs to provide minimal support for switching between AMD64 and IA-32 mode. The experimental implementation for L4Ka/IA-64 was designed to take this approach; however, due to restrictions of the IA-64 architecture, the data conversion actually happens in kernel mode, as illustrated by Figure 4 (b) [15, Figure 4.5].

Figure 3: Kernel-level and user-level solutions



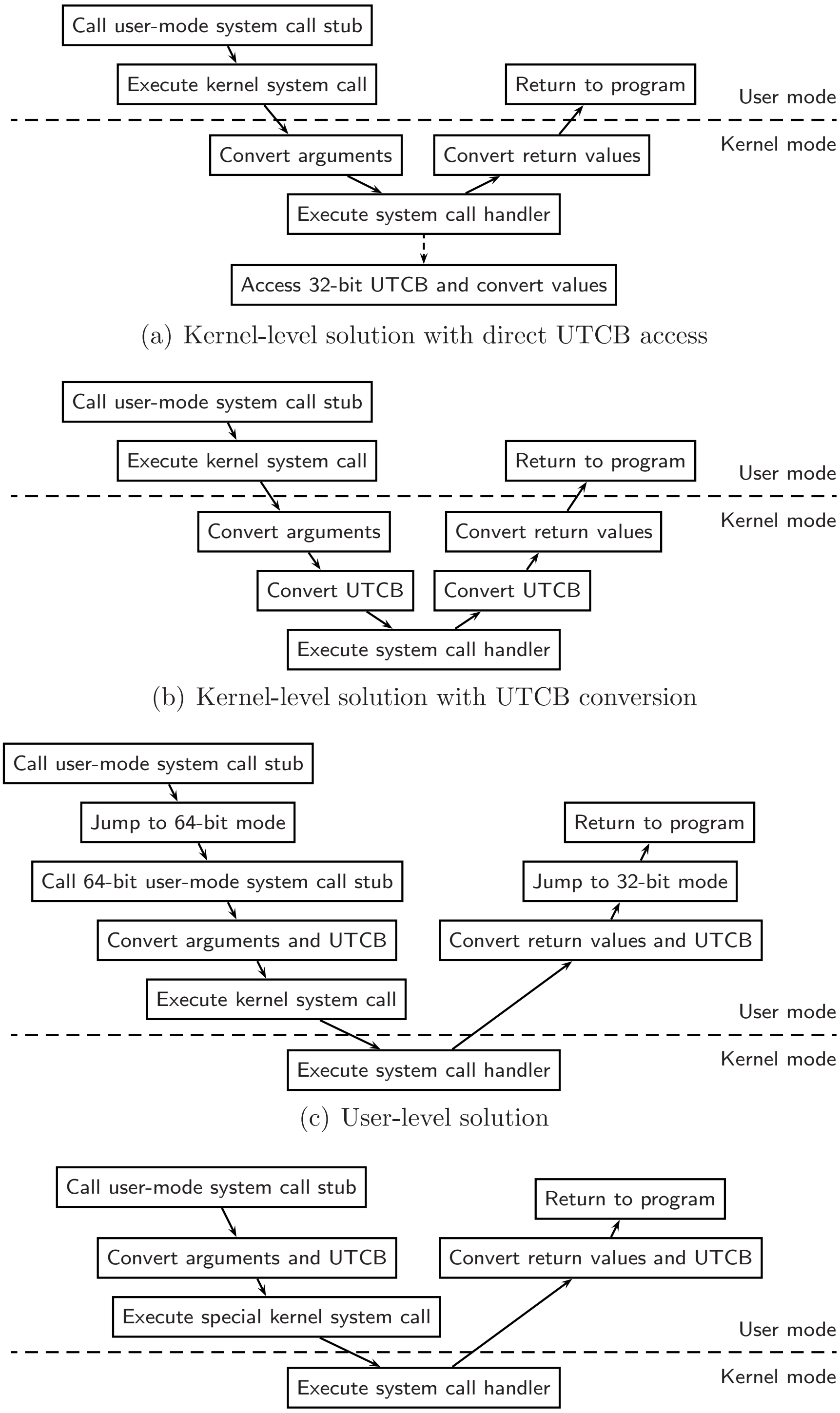
At first sight, the user-level approach appears to be best suited for a microkernel:

- It keeps the kernel as small as possible, which is one of the main principles of microkernel design [16].
- It is more robust and secure, since errors in the conversion process do not have any impact on the kernel.
- The kernel does not need to differentiate between 32-bit and 64-bit programs in most cases. In the kernel-level solution, the kernel frequently needs to determine the type of program, which introduces some overhead even for AMD64 programs.

However, there are several important drawbacks:

- The speed of IPC operations is critical in a microkernel. Multi-threaded IA-32 applications will generally invoke a lot of IPC operations from 32-bit to 32-bit threads. A user-level compatibility layer usually needs to convert IPC data to 64-bit values first, then convert it back on the receiving side. Data conversion obviously causes a fairly large overhead.
  - Alternatively, a special IPC short path can be implemented, as outlined in the diploma thesis about 32-bit support for IA-64 [15, Section 5.2]. However, this solution requires additional kernel support, contrary to the goal of keeping kernel code minimal.
- Translation of system call arguments and other data structures always causes a delay. Since system calls are executed very often, such a delay may actually be noticeable. If the kernel natively supports IA-32

Figure 4: IPC and other system calls


Call user-mode system call stub
Execute kernel system call
Return to program
User mode
Kernel mode
Convert arguments
Convert return values
Execute system call handler
Access 32-bit UTCB and convert values
Call user-mode system call stub
Execute kernel system call
Return to program
User mode
Kernel mode
Convert arguments
Convert return values
Convert UTCB
Convert UTCB
Execute system call handler
Call user-mode system call stub
Jump to 64-bit mode
Return to program
Call 64-bit user-mode system call stub
Jump to 32-bit mode
Convert arguments and UTCB
Convert return values and UTCB
Execute kernel system call
User mode
Kernel mode
Execute system call handler
Call user-mode system call stub
Return to program
Convert arguments and UTCB
Convert return values and UTCB
Execute special kernel system call
User mode
Kernel mode
Execute system call handler

(a) Kernel-level solution with direct UTCB access

(b) Kernel-level solution with UTCB conversion

(c) User-level solution

(d) Intermediate solution

programs, it does not always need to convert arguments (e.g., if it needs to decode arguments anyway before continuing, or if some arguments are ignored in certain situations).

- The code that converts system call arguments needs to be executed in 64-bit mode (see Figure 4 (c)), since it needs to conform to the AMD64 system call interface if it is not built into the kernel. (The AMD64 interface of L4 uses 64-bit registers not available to 32-bit programs.) Therefore a switch from 32-bit to 64-bit mode is necessary before the actual system call, which causes an additional delay.

  – An intermediate solution is possible as well (see Figure 4 (d)): 32-bit user-level code converts the arguments, saves them at locations known to the kernel, and invokes a special 32-bit system call. The system call handler reads the values and directly passes them to the 64-bit system call handler. However, this approach does not require much less in-kernel code than implementing 32-bit system calls natively.

- In the L4 microkernel, a UTCB (User-Level Thread Control Block) is used as part of the interface between a program and the kernel [9]. Both kernel and user programs write to the UTCB and expect the change to be visible on the other side. Since the UTCB data structures of the IA-32 and AMD64 interface are different, two UTCBs are necessary, and the compatibility layer constantly has to transfer data between them. In the L4Ka/IA-64 implementation, UTCB synchronization has a significant impact on IPC performance [15, Section 5.2].

- The KIP (Kernel Interface Page) of L4 contains certain volatile system data, such as the processor frequency. Since the compatibility layer needs to provide a KIP compatible with the IA-32 L4 API, it needs to read and translate all volatile data from the AMD64 KIP periodically. It cannot provide the information on demand because 32-bit code may read the values without notifying the compatibility layer.

To avoid these drawbacks, especially the high cost of IPC operations, we chose to implement the kernel-level alternative (as in Figure 4 (a)), providing a native IA-32 interface to 32-bit programs.

4.2 Address Spaces

Another less important design decision is whether 32-bit and 64-bit threads are allowed to run in the same address space. In theory, the AMD64 architecture supports this, with the limitation that 32-bit threads can only access the first 4 GB of their address space [4, Section 1.3.3]. However, this feature is not required in any case – the same effect may be achieved using two identical address spaces – and possible applications are very rare. In L4, this would require the address space to contain two KIP structures, one for 64-bit programs, the other for 32-bit programs. Since the design of the L4Ka implementation strongly depends on a single KIP area, which is referenced at various places throughout the entire source code, our implementation will not permit 32-bit and 64-bit threads to run in the same address space.

4.3 Transparency

Communication with 64-bit threads must be transparent to 32-bit threads, since the goal of Compatibility Mode support is to run unmodified 32-bit programs. Transparency is not strictly required on the 64-bit side, but still desirable, as it reduces programming effort for 64-bit programs.

4.4 API

Since loading and decoding of executable files is done at user level in L4, the AMD64 kernel interface must be extended so user-level code can specify that a thread is to be executed in Compatibility Mode. Moreover, 64-bit programs sometimes need to know whether an IPC message was sent by a 64-bit or a 32-bit thread. There a a few possible solutions, all of which have pros and cons:

- Encode the thread type information in the thread ID, by setting a previously unused bit for 32-bit threads. In other words, divide the thread ID space into two parts, one for 32-bit threads, the other for 64-bit threads. Ideally, the kernel analyzes this bit only when it needs to decide whether a thread is running in Compatibility Mode, and 64-bit user-level code can use it to detect IPC messages from 32-bit threads. An experimental implementation used such a design, but the solution has some serious flaws:

  - The API specification states: “A global thread ID consists of a word, where 18 bits (32-bit processor) or 32 bits (64-bit processor) determine the thread number and 14 bits (32-bit processor) or 32 bits (64-bit processor) are available for a version number.” [9, Section 2.1] Although it does not require all 32 bits of the thread number to be usable, assigning another role to any bit of the thread ID violates the present specification.
  - Before the kernel delivers an IPC message, it checks whether the destination thread carries the correct thread ID. However, when a 32-bit thread sends a message, it cannot differentiate between 32-bit and 64-bit destination threads, since it only knows 32 out of the 64 bits representing the thread ID (which must be the lower 18 bits of the thread number and the lower 14 bits of the version number). Therefore the check will fail for either 32-bit or 64-bit destination threads, unless the bit is either excluded from thread ID comparison in the kernel or set correctly when system call arguments are converted from 32 to 64 bits.

- Add a flag to the `ThreadControl` system call responsible for thread manipulation in L4. Since no control argument exists, it must be encoded in another argument for existing 64-bit applications to work. Using `ThreadControl` especially makes sense because logically the type of executable code is an attribute of the thread.

- Specify it via `SpaceControl`, which is responsible for address space manipulation, instead of `ThreadControl`. Such an approach is possible if 32-bit and 64-bit threads are not allowed to run in the same address space. One advantage is that the `SpaceControl` system call contains a control argument featuring architecture-specific flags. However, since `SpaceControl` is always executed after `ThreadControl`, the kernel does have any information about the type of a thread when it is created. This can cause problems, for example when setting the initial values of the segment registers.
  The final implementation uses this design, and specifically addresses the issue of `SpaceControl` and `ThreadControl` call order.
  When a 32-bit thread sends an IPC message to a 64-bit thread, the kernel notifies the 64-bit thread by modifying its UTCB.

5 Implementation

In this section, we will present the details of the implementation of Compatibility Mode support in L4Ka::Pistachio/AMD64. We will describe how the required hardware features are accessed, how each aspect of the overall design is implemented, and which special problems are met.

5.1 Hardware Support

The processor switches to Compatibility Mode if the `L` bit in the segment descriptor of the current code segment is set [4, Section 4.8.1]. This bit must either be set and reset on every thread switch, or two different segment descriptors must be set up, and the segment selector must point to the correct one.

The latter approach is always preferable because the code segment selector (`CS`) must be reloaded anyway if the segment descriptor is modified [4, Section 4.4], thus the cost of modifying the segment descriptor is definitely higher. Moreover, `sysret`, the processor instruction used to return to user mode from system calls, sets the segment selector to different predefined values depending on whether it is called with a certain prefix [5, Section 4]. Thus, instead of setting the segment selector explicitly, the implementation sets it implicitly by using the prefix in the 64-bit system call handler only.

The stack segment selector (`SS`) is always set to the same value by `sysret`, regardless of the target mode. In L4, the value is the same as the user data segment selector (`DS`). In the segment descriptor, the `D` bit must be set for correct operation in Compatibility Mode. It is ignored in 64-Bit mode.

On the AMD64 test machine, in some cases a stack exception occurred on the first stack access after the system call return. I have not found the cause of this problem, actually it seems to be a hardware bug. It can be worked around by reloading the `SS` register after the system call.

5.2 KIP and UTCB

One of the main design goals was to keep the amount of code added to the kernel minimal. To achieve this goal, the header files containing the KIP (Kernel Interface Page) and UTCB (User-Level Thread Control Block) data structures were modified to be able to include them multiple times in the same source file, creating data types of different word size.

Figure 5: Redefinition of Data Types

```
namespace ia32 {
    typedef u32_t word_t;
    #include <file.h>
}
```

To create two data types, a header file is included normally at first, then again within a namespace declaration. Using a namespace works around the obvious name conflict, and more importantly the namespace will be searched first for every data type *used* by the header file. Therefore even general data types may be redefined for the 32-bit data structures, by defining types of the same name in the namespace (see Figure 5).

This works even if the header file contains several data types that use each other, since all of them will be redefined in the namespace. They can be accessed outside of the namespace by prefixing them with the namespace identifier.

Since the KIP is usually not modified after the system has been initialized, copying the data from the 64-bit KIP to the 32-bit KIP at boot time is enough. In the future, the 64-bit and 32-bit KIPs will have to be updated simultaneously, for example when processor frequency scaling support is introduced. (The user program can read the current processor frequency from the KIP.)

When an address space is initialized, the KIP is mapped into it at a user-defined virtual address. Therefore at that time, the required word size needs to be known, and the KIP must be chosen accordingly. Kernel access to the 32-bit KIP is never needed at any other time.

Providing a 32-bit UTCB is more complex, as the kernel needs to read and modify its fields. Fortunately, all accesses are performed via functions in the TCB data structure, which we rewrote to check the type of thread and access the UTCB accordingly. A cleaner and possibly even faster approach would be to use two UTCB wrapper classes inside the kernel, which inherit virtual UTCB access functions from a single abstract class. However, even though the kernel is written in C++, the infrastructure necessary for virtual functions is not present. Moreover, special magic would be needed to avoid virtual functions if only one type of UTCB is used. And in general, virtual functions impose performance problems rather than solving them.

The operation to copy message registers from one UTCB to another is part of the TCB data structure as well. The Compatibility Mode version uses optimized copy loops for 64-bit to 64-bit and 32-bit to 32-bit transfers. This way good performance may be achieved very easily.

At user level, the UTCB is accessed by dereferencing a pointer stored at `GS:0` in the IA-32 and AMD64 interfaces. As a result, the kernel must set the `GS` segment base to point into a special page containing such pointers. In both IA-32 and AMD64 kernels, the page is mapped outside of the user area, to avoid conflicts with user mappings. However, in Compatibility Mode, the upper 32 bits of the segment base are ignored, so another mapping at the same address truncated to 32 bits must be set up. For security, the user must be prevented from mapping this page to other programs, and especially from unmapping or overmapping it.

5.3 System Calls

In L4, user code invokes system calls by calling kernel-supplied stub functions which execute the actual system call instructions. The kernel supplies the addresses of these functions via the KIP. In the AMD64 interface, the addresses are absolute, but on IA-32 they are relative to the base address of the KIP. Since the kernel only allocates a single KIP for each word size, these relative addresses are fixed. However, the KIP is mapped into address spaces at different locations, so the virtual addresses of the 32-bit stubs depend on the virtual address of the KIP. Therefore, the stubs need to be combined with the KIP and mapped along with it.

On AMD64, the preferred instruction to enter the kernel for system calls is `syscall` [4, Section 6.1]. The kernel may specify different entry points for system calls executed by 64-bit and 32-bit programs. The 32-bit system call handler must conform to the IA-32 kernel ABI, so it needs to read the arguments from the registers used in the IA-32 interface, convert them from 32-bit to 64-bit data structures, and then call one of the system call functions which are used by the 64-bit handler as well. When this function returns, the system call handler must convert the return value and copy it into the register specified by the IA-32 ABI.

The `syscall` instruction clobbers the `ECX` register, which is used by some system calls in the IA-32 ABI of L4. The user-mode system call stub supplied by the kernel therefore needs to copy the contents of this register to another register or to memory before calling the `syscall` instruction. The system call handler must be adapted accordingly.

The Intel equivalent of AMD64, called EM64T, differs from AMD64 only in a few aspects, but one of these is that the `syscall` instruction may not be called from Compatibility Mode. Instead, Intel offers `sysenter` [6, Section 4.8.7.1], which has slightly different semantics, requiring modified user-mode stubs and a different system call handler. `sysenter`, in turn, is not available on AMD64 in Compatibility Mode. To keep the kernel simple, the type of system must be chosen at compile time. Actually, the use of `syscall` on EM64T or `sysenter` on AMD64 raises an illegal opcode exception, so these instructions can be emulated easily by calling the system call handler directly from the exception handler. However, since the overhead of exceptions is much higher than the overhead of regular system calls, this method is not a realistic option.

5.4 IPC

When a 32-bit thread sends an IPC message to a 64-bit thread, the upper 32 bits of the destination message registers are zero-filled, treating the values as unsigned integers (except for the label, which is sign-extended). The resulting 64-bit values are correct for addresses, fpages, string items, and map/grant items, so no special treatment is necessary for any of these. When the kernel decodes the message to handle string and map/grant items, it reads the message registers from the UTCB using the access functions, and then operates on the values independently of the type of thread.

For simple register-only IPC, L4Ka/AMD64 offers a fast path bypassing the usual IPC system call function. If the source thread is a 32-bit thread, the fast path is not used, since there is a separate system call entry point for Compatibility Mode. However, if the source thread is a 64-bit thread and the destination thread is a 32-bit thread, the fast path may be entered accidentally unless special care is taken. The fast path implementation already contains checks for all non-standard conditions requiring slow path IPC, including a check for resource bits set in the source or destination thread. Resource bits indicate the need to perform specific actions on a thread switch, such as saving and restoring floating point registers. In the AMD64 port, setting a resource bit in one of the two threads involved unconditionally causes the slow path to be entered. Thus, adding a Compatibility Mode resource bit is sufficient to prevent fast path IPC from being used between 64-bit and 32-bit threads.

5.5 Miscellaneous

As the `ExchangeRegisters` system call uses more than two registers to return values, its implementation uses a special return path separate from the actual system call handler. In other words, the `sysret` instruction is performed inline inside of the system call function. If Compatibility Mode support is enabled, the return code needs to check whether the calling thread is a 32-bit thread, and use the appropriate registers and `sysret` prefix.

The kernel debugger decodes certain instructions to perform special actions such as debugger entry and console I/O. To trigger a kernel debugger feature, user or kernel-level code must execute a software interrupt followed by a specific instruction. The AMD64 kernel debugger interface uses the same instructions as the IA-32 interface, but operands are usually extended from 32 to 64 bits in the 64-bit instruction set of AMD64. Therefore, in the interrupt handler, the kernel debugger first needs to determine the type of thread and then read 32 or 64 bits from user memory, respectively.

The L4Ka kernel compiled for the AMD64 platform is substantially larger than the IA-32 kernel. The size becomes an issue when the root task is an IA-32 program because root tasks receive idempotent memory mappings from σ_0. Per default, 32-bit root tasks use base addresses that are occupied by the kernel on AMD64. The base addresses can be changed in the compile-time configuration, but actually such a conflict is hardly noticeable. Therefore we extended the boot loader utility to detect and report it.

6 Evaluation

In this section, we will evaluate the implementation in several ways. First of all, we will ensure that it provides correct functionality in the sense that programs compiled for L4/IA-32 can be executed and produce the same results as on the IA-32 kernel. Next, we will measure performance of IPC between all types of threads, as good IPC performance was one of the main design goals. Finally, we will analyze and justify the amount of additional complexity introduced into the kernel source code.

6.1 Correctness

The L4Ka distribution contains an L4 test suite covering a large part of the functionality defined in the L4 specification. The IA-32 version of L4Ka passes most of the tests. One exception is the return value of the IPC system call when called with a local destination ID. For some other tests, the expected result is not obvious, so it is unclear whether the tests succeed or fail.

Therefore, when evaluating the Compatibility Mode implementation, the goal was not for all tests to succeed, but to produce the same results when run on an AMD64 or IA-32 kernel. In this sense, the implementation has been tested successfully using the regular test suite compiled for the IA-32 platform.

The virtualization layer built on top of L4Ka is a complex L4 application using many advanced features of L4. Although a real application can not replace a test suite, it serves well as an addition. However, I was not able to get the virtualized Linux system to run completely even on an IA-32 microkernel. Shortly after the kernel has booted, the system hangs. When executed on top of the AMD64 microkernel on the same machine, it hangs directly after kernel boot-up, or even earlier if IDE support is enabled in the Linux kernel.

6.2 Performance

6.2.1 64/64-bit Performance

The most critical performance data to be measured is the speed of regular IPC between two 64-bit threads, more precisely whether the existence of Compatibility Mode support in the kernel has a noticeable impact on 64-bit IPC performance.

For performance analysis, we need to distinguish between two different types of IPC operations: Like the IA-32 target of L4Ka, the AMD64 target offers an optional fast path for IPC operations that meet certain criteria [12]: For example, the message may only contain untyped words (no string or map items), the destination thread must be waiting, and as a new requirement, it must be a 64-bit thread as well. More complex IPC operations are handled by the slow path. The fast path can be turned off entirely in the kernel configuration menu.

The fast path code, including all of the validity checks, is written entirely in Assembler and does not contain any calls to other parts of the kernel. The addition of Compatibility Mode support did not require any changes to this code. To prevent an IPC operation with a 32-bit target thread from entering the fast path, the implementation uses the existing functionality of “Resource Bits”, which is actually designed to inform the kernel about special resources to be saved and restored on thread switches (such as floating point registers). In the AMD64 implementation, the slow path is entered whenever one of the involved threads' resource bits is set. Consequently, the compiled fast path code of the Compatibility Mode kernel is instruction-identical to the present version. If there are any performance differences, they must be the result of alignment, caching, or TLB issues.

For performance measurements, an IPC ping-pong program is included with the L4Ka distribution. It starts two threads sending a given number of IPC messages to each other, consisting of a varying number of untyped words. For each message size, it calculates the average time a single IPC operation takes, and expresses the time in processor cycles as well. The user can choose whether the threads run in different addresses spaces and on different processors (if multiple processors are available).

Using this program, we measured the performance of slow-path IPC between two 64-bit threads on L4Ka/AMD64, using three different kernels:

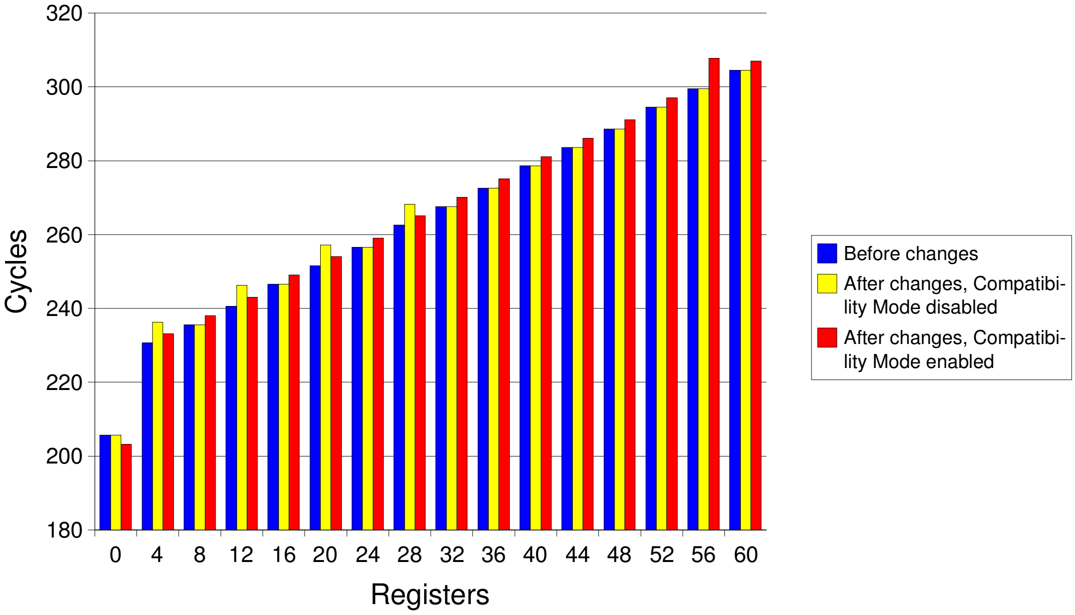
- the original kernel without any modifications related to Compatibility Mode (except for some general bug fixes),
- a kernel built from the modified source code, but with Compatibility Mode support disabled at compile time,
- and finally, the modified kernel with Compatibility Mode support enabled.

All kernels were built using the same compiler and executed on the same two machines: An AMD64 system with an AMD K8 processor and an EM64T system with an Intel P4 processor. All debugging facilities were excluded at compile time, and the AMD K8 Flush Filter was disabled for better accuracy. Since address space switches have relatively high performance costs which are dictated by the hardware and obviously not influenced by Compatibility Mode support, the most relevant results are those of two threads running in the same address space.

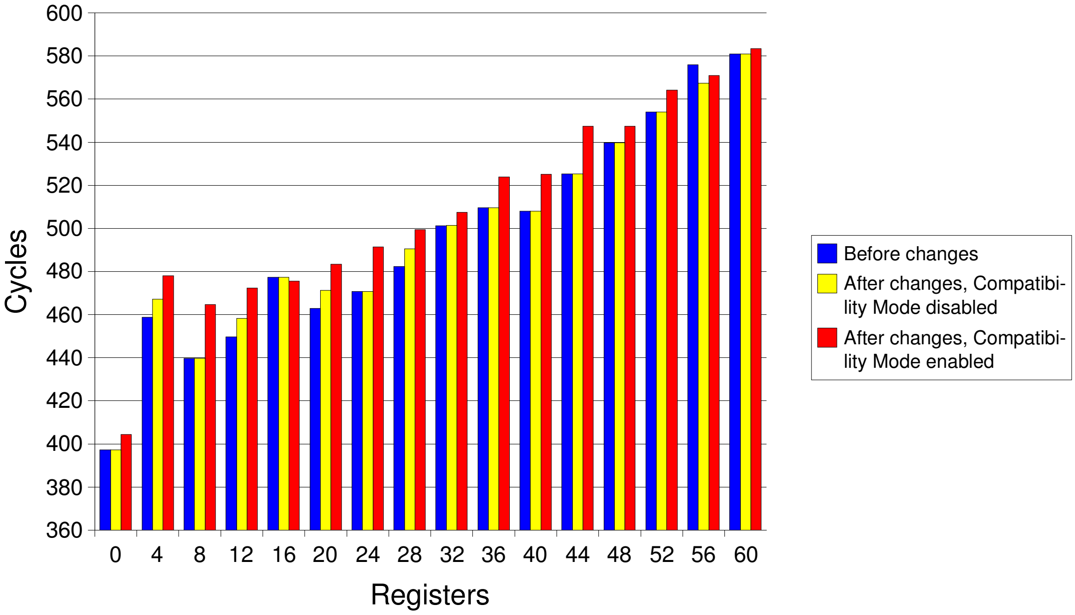
The actual results (see Figure 6) leave some room for interpretation, but a few conclusions can be drawn:

- IPC Performance of all three kernels differs by less than 25 processor cycles per IPC operation.

Figure 6: 64-bit Intra-AS IPC Performance

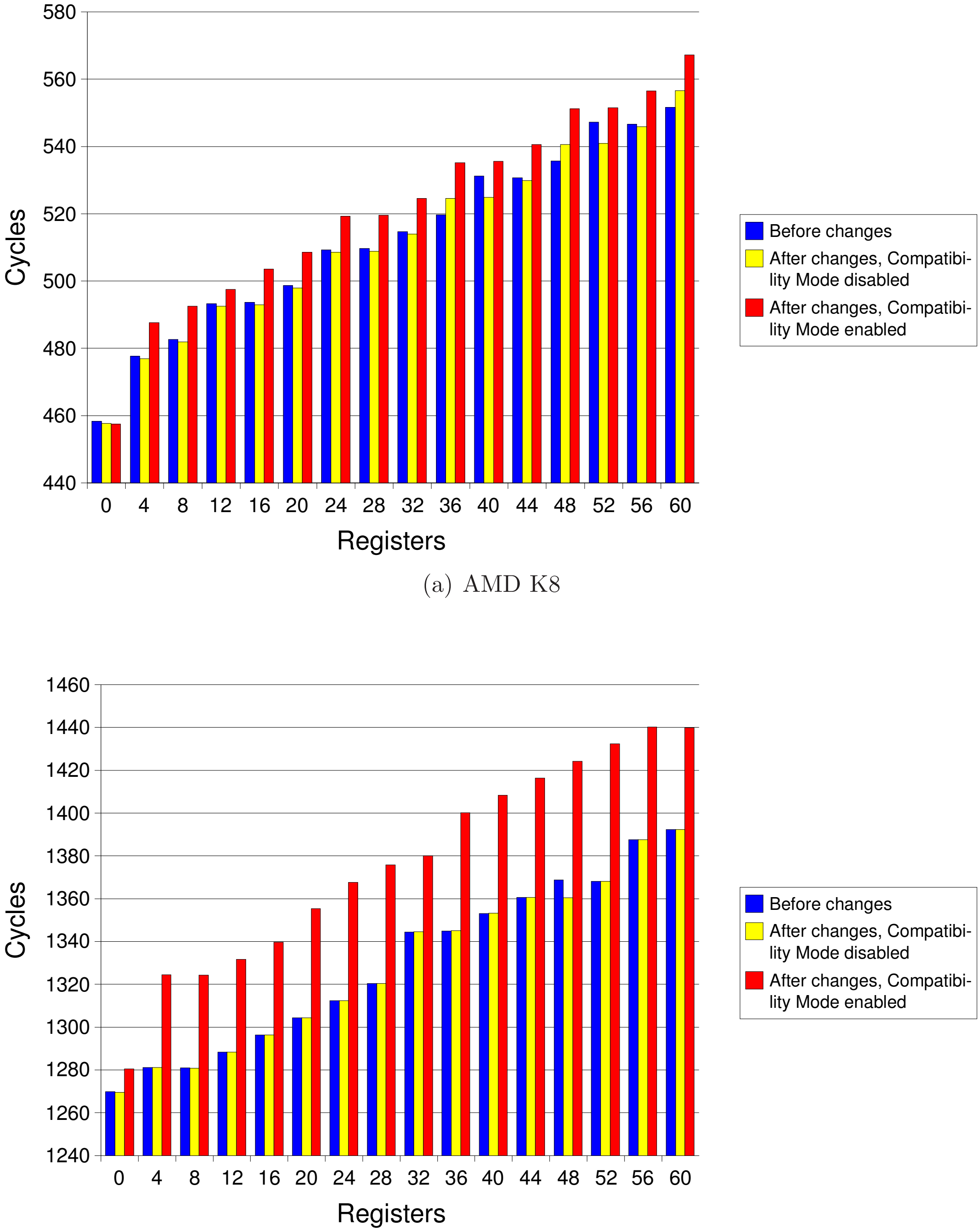


(a) AMD K8



(b) Intel P4

Figure 7: 64-bit Inter-AS IPC Performance



(a) AMD K8

(b) Intel P4

- Performance measurement using the ping-pong program is not accurate enough to provide exact quantitative data. For example, in theory, the Compatibility Mode kernel can never perform better than the original kernel, and a larger IPC message must always result in a longer transfer time. Some of the anomalies disappear when taking the average over multiple runs, others remain. In any case, exact cycle counts are an illusion because of caching and pipelining effects or interrupts.

- Nevertheless, it is evident that the Compatibility Mode kernel consumes a few more cycles per IPC operation than the original kernel. This is a predictable result because Compatibility Mode support introduces some additional instructions determining the type of thread at a few places in the code.

- The number of additional cycles can be regarded as constant with respect to the number of message registers. On the AMD K8 system, an IPC operation uses approximately 4 additional cycles if Compatibility Mode is enabled. Given the total number of cycles per IPC operation, this is an increase of 2% or less, depending on the number of message registers. (Note that the scale in the figures does not start at 0 cycles.) The Intel P4 data is very inaccurate, a rough estimate is 20 cycles or 5%.

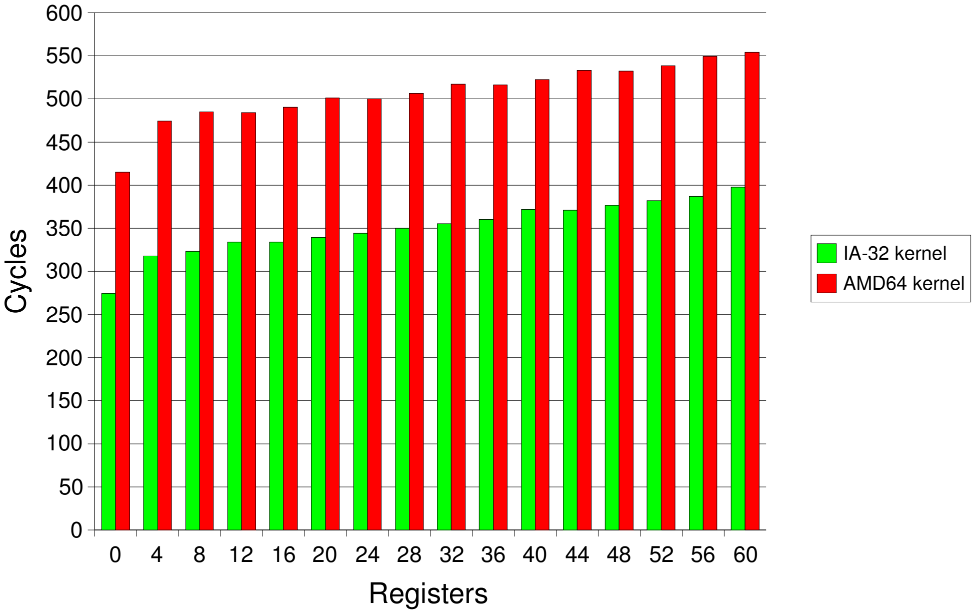
- The modified kernel with Compatibility Mode disabled cannot be proven to perform worse than the original kernel (which would, in fact, be surprising, since the differences between these two kernels are minimal).

The remaining question is whether a performance drop of 2% or even 5% is acceptable. Such an impact on performance could be avoided if one of the other design alternatives was chosen. However, it is important to remember that fast-path IPC is not affected. For this reason, we claim the performance drop to be acceptable.

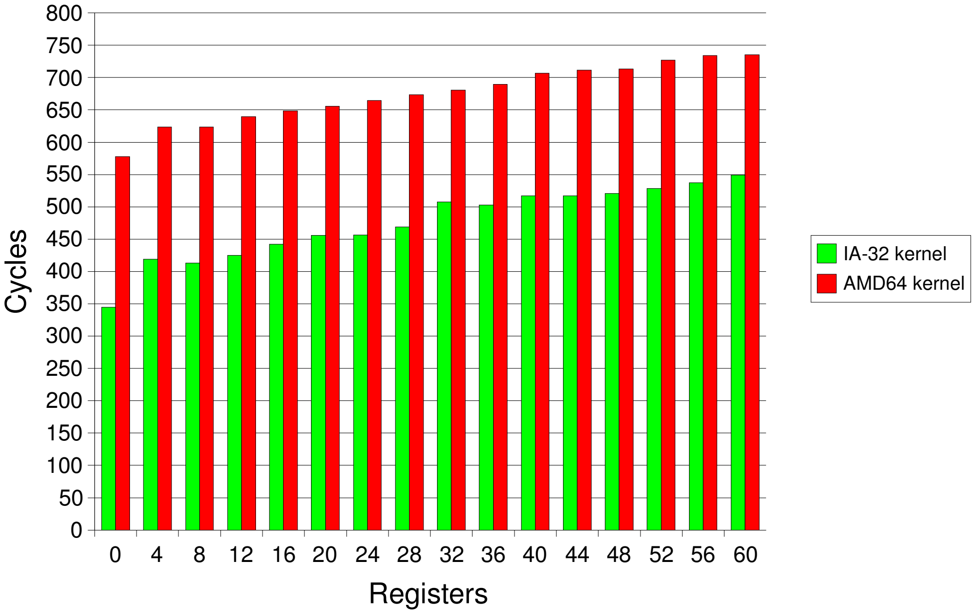
6.2.2 32/32-bit Performance

Since Compatibility Mode support was designed to run unmodified programs compiled for the IA-32 architecture, the IA-32 version of the ping-pong program can be used to measure the speed of IPC between two 32-bit threads on a native AMD64 kernel, and to produce an accurate comparison with an IA-32 kernel running on the same machine.

Figure 8: 32-bit Intra-AS IPC Performance

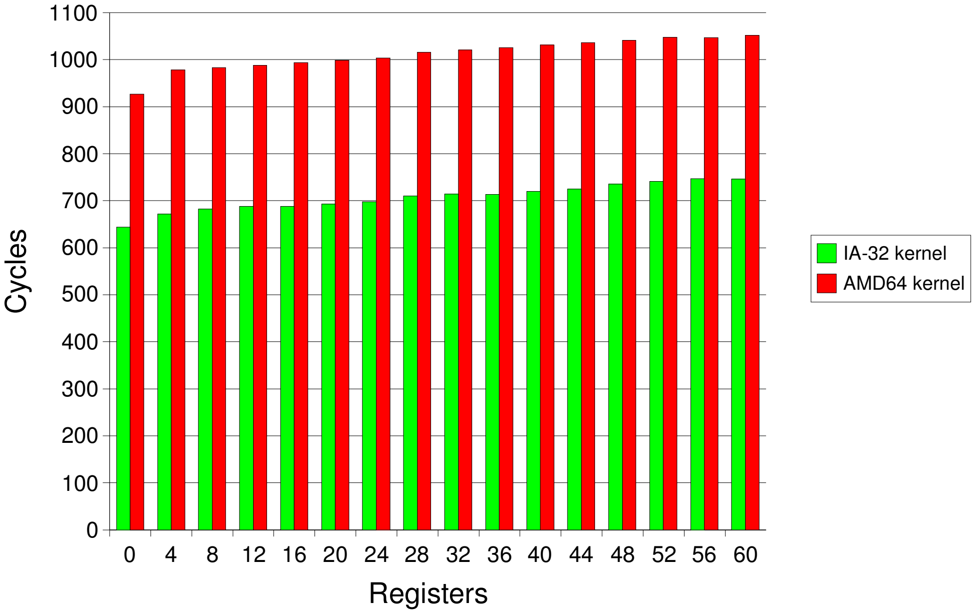


(a) AMD K8

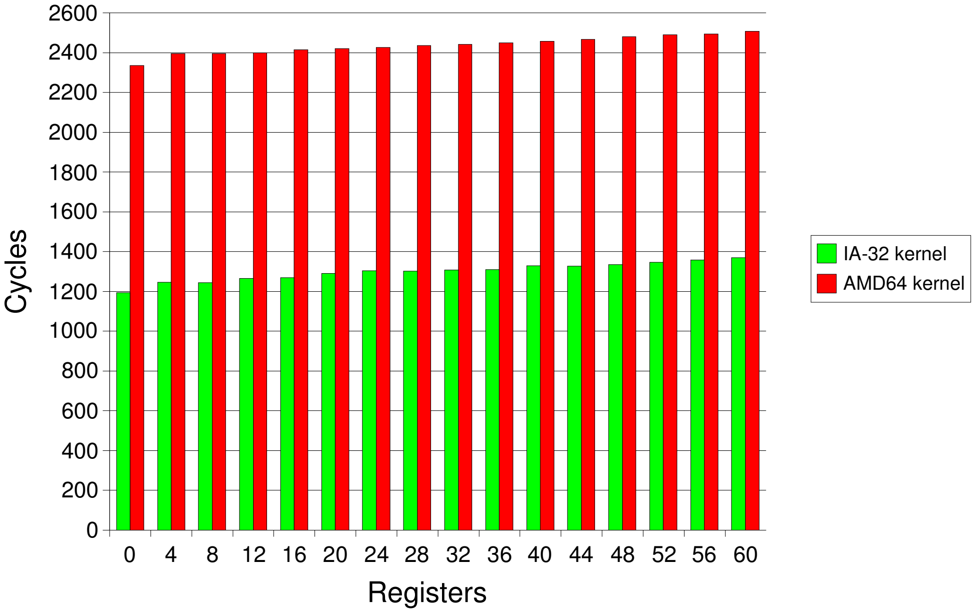


(b) Intel P4

Figure 9: 32-bit Inter-AS IPC Performance



(a) AMD K8



(b) Intel P4

The results (see Figure 8) are as inaccurate as the 64-bit results, but a quotient of less than 1.7 between the number of cycles used by the AMD64 kernel and the IA-32 kernel is common to all measurements. (On the AMD K8 system, it is between approximately 1.5 for empty messages and 1.4 for messages with 60 untyped words. The Intel P4 system has a quotient near 1.7 for empty messages, but 1.5 as well for 4 untyped words. The quotient between execution times is always the same as the quotient between cycles, which shows that cycles are not measured differently depending on the kernel).

Whether this value is high or low is a matter of opinion. In general, a certain unavoidable performance overhead is induced by the conversion of kernel data structures such as thread IDs. For a complete evaluation, the alternative user-level design of Compatibility Mode support would need to be implemented, in order to get comparable results.

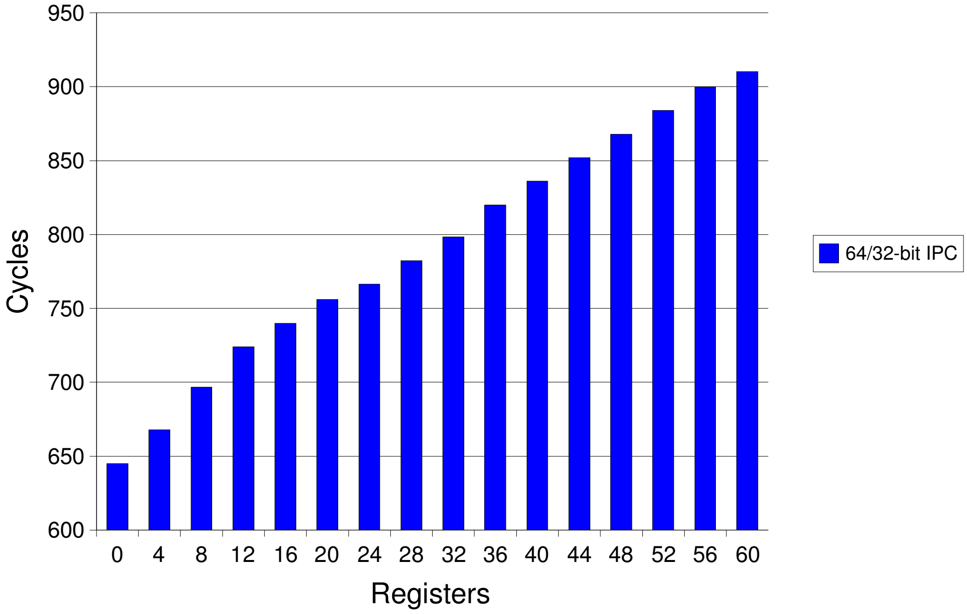
Unfortunately, the diploma thesis describing the IA-64 implementation does not contain a comparison with a native IA-32 kernel. The author does compare 32-bit and 64-bit IPC, to discover that 32-bit IPC is slower by a factor of 2.1 [15, Section 5.2]. Results on AMD64 show approximately the same factor (compare Figure 6 and 8), but the relevance of such a comparison is questionable.

For example, on AMD64, 32-bit IPC on an IA-32 kernel already turns out to be considerably slower than 64-bit IPC on an AMD64 kernel. The AMD64 architecture has the advantage over IA-32 of having a larger set of processor registers, which plays an important role in L4 IPC performance.

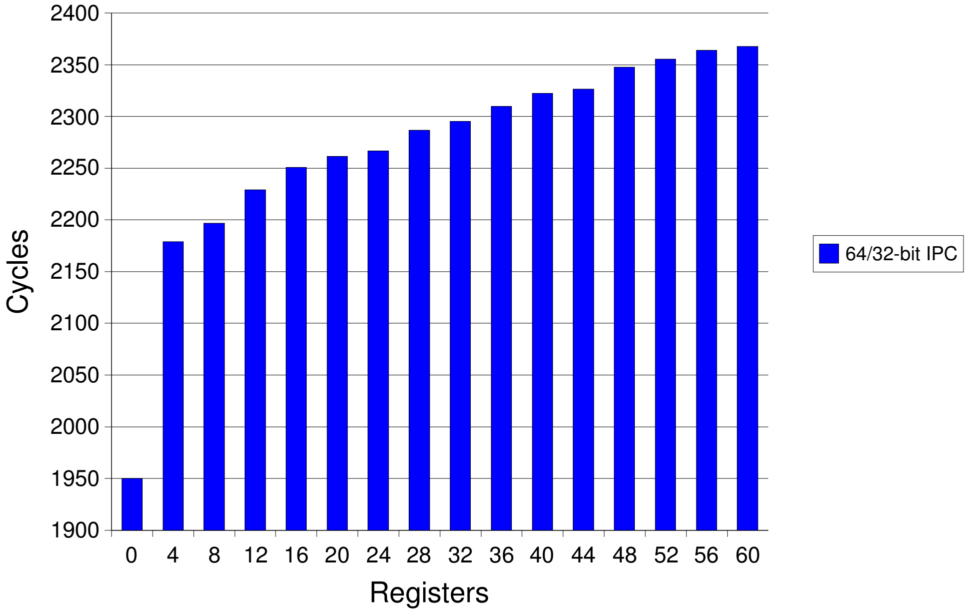
Moreover, depending on the exact hardware details, a system call from a 32-bit thread into a 64-bit kernel can take more cycles than a system call executed by a 64-bit thread. Hardware can also be optimized for 64-bit code in general, making 32-bit code run more slowly.

Performance of 32-bit IPC could be improved drastically by adding a fast path similar to the existing fast path for 64-bit IPC, which handles empty IPC messages in 122 cycles on the AMD K8 system. Such a fast path could use 32-bit thread IDs directly without the need for converting them at first, which would eliminate a lot of unnecessary branches in the code. However, since 32-bit applications are usually not required to run at full speed on 64-bit systems, the additional complexity of another fast path was deemed unnecessary.

Figure 10: 64/32-bit Inter-AS IPC Performance



(a) AMD K8



(b) Intel P4

6.2.3 64/32-bit Performance

To test the speed of IPC between a 32-bit and a 64-bit thread, the AMD64 version of ping-pong was extended by a feature to replace one of the two ping-pong threads with a 32-bit thread. The 32-bit code is not fully optimized, so the results may be a few cycles too high. Since 64-bit and 32-bit threads cannot run in the same address space, only Inter-AS data exists.

The results are not comparable to the other types of IPC operations. They are included for comparison with the results of future studies.

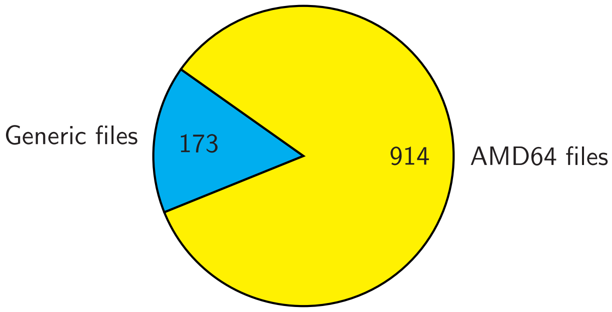
6.3 Complexity

Since the kernel-level design of Compatibility Mode support is likely to introduce more additional complexity in the kernel than the user-level solution, an analysis of this complexity is necessary to justify the design decision. Complexity can be analyzed in a variety of ways, both quantitatively and from the perspective of readability and maintainability. We will try to answer the following research questions:

- How many lines of code were changed or added in the kernel sources for the purpose of Compatibility Mode support?
- How many of these are local to the AMD64 target, and how many involve files that are used globally?
- How many of them would still be necessary if data conversion happened at user level? Would there be any need for additional functionality?
- Are the changes to global files applicable to other multi-architecture systems?
- Do any of the changes affect the readability of the code?
- Do any potential changes in the future involve more work if Compatibility Mode support is included?

In general, affected lines of code are a good measurement if white space and comments are excluded. Even if different programming styles result in fewer or more lines given the same code, they are a good indication, since every change bears the potential of adding complexity.

Figure 11: Changes in generic and AMD64-specific files



However, a few changes involved renaming certain identifiers at all places where they were used in a file, and counting every single instance of the identifiers would not be fair (especially since there is no actual complexity involved). In these cases, we count the entire renaming operation as a single change.

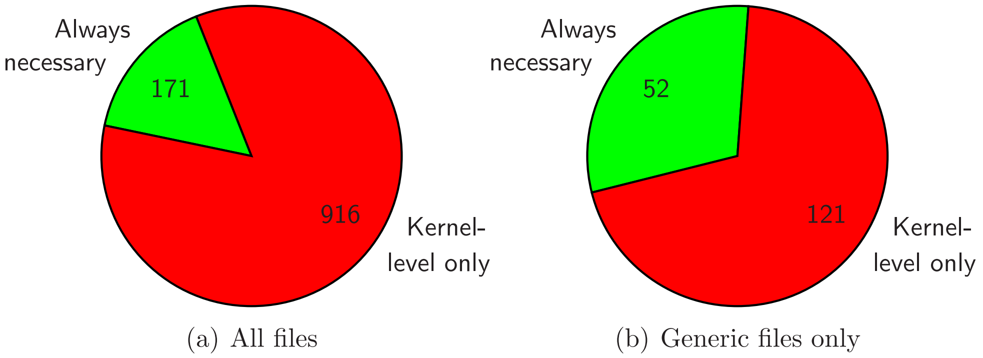
Some changes only involve user-level code, for example user header files, σ_0, the boot loader (which is technically not executed at user level but written using the same infrastructure), and also the system call stubs that are mapped into the applications' address spaces. These were excluded as well, since only the complexity of the kernel is to be investigated. If they were included, then a user-level solution would be regarded as complex, when in fact it is the reference for minimum complexity.

The remaining changes involve 1087 lines of code, 914 of which are local to the AMD64 target (see Figure 11). The other 173 lines are modifications to files that are used by other targets as well.

Out of the 1087 lines that were changed, only 171 would remain in a user-level solution (see Figure 12). The rest is related to tasks like system call argument conversion, 32-bit UTCB access, providing a 32-bit KIP, etc., which would be handled at user level. Code that would remain is either related to the hardware (for example, setting the segment descriptors) or to the general feature of running 32-bit code (for example, instruction emulation).

Therefore, a large number of changes could be avoided in a user-level solution. A reasonable conclusion might be that a user-level solution would require virtually no changes outside of the AMD64 target. However, this is not the case: A configuration menu needs to be created, the boot protocol must be extended to include the thread type of the root servers, the IPC system call needs code to inform the target thread about the sending thread, etc. These tasks make up 52 of the 173 lines of changes to generic files.

Figure 12: Changes that would still be necessary in a user-level solution vs. changes specific to the kernel-level solution



(a) All files

(b) Generic files only

The question about additional kernel functionality required by a user-level solution cannot be answered as clearly. What kind of functionality is required depends heavily on the details of the design. For example, if no 32-bit system call interface is provided by the kernel, then system calls need to be executed by an intermediary layer running in 64-bit mode. But in that case the kernel needs to provide facilities for 32-bit user programs to call the intermediary layer and thereby switch to 64-bit mode. On the other hand, a 32-bit system call interface introduces more complexity in the kernel as well, even if arguments are converted on the user side.

Another problem is that user programs use a certain processor instruction to obtain the address of the KIP on both L4/IA-32 and L4/AMD64. The instruction raises a hardware exception that is handled directly by the kernel without a chance of intervention from user mode (e.g. from a 32/64-bit compatibility layer). However, if the 32-bit KIP is handled by the user-mode compatibility layer, the AMD64 kernel does not have any information about its location in the 32-bit thread's address space. Therefore, special functionality needs to be implemented in the AMD64 target to query the compatibility layer for the address of the 32-bit KIP, or to forward the KIP address request to the compatibility layer in some way.

In theory, the UTCB creates a similar problem: In user mode, the `GS` segment register is used indirectly to obtain the address of the current thread's UTCB. A user-mode compatibility layer must therefore either be allowed to change the value of this segment register (unlike other user-mode code), or the kernel needs to load a different value on switches between the 32-bit application and the compatibility layer. In practice, this problem is solved by the fact that the `GS` segment descriptor is truncated to 32 bit in Compatibility Mode, which means that the segment starts a different location in 32-bit programs.

All in all, the additional functionality required for a user-level solution must not be neglected.

Readability and maintainability are an issue mainly for the changes to generic files. Most of the changes are normal C++ code, for example the addition of another method or field. However, there is one exception: The changes to KIP and UTCB code that were necessary to produce 64-bit and 32-bit data structures from the same source files contain macros. A default definition for each macro is provided in the file itself, and another definition is provided by AMD64-specific code. In general, the use of macros may affect readability. The cases where new macros were introduced are rare though, most of the re-definable macros replace fixed macros which were there before.

It should be noted that producing 64-bit and 32-bit data structures from the same files is not strictly necessary. The source files could just be duplicated, which would eliminate the need for additional macros. The copies could actually be placed in the part of the source tree that is local to the AMD64 target. However, duplicated code introduces an obvious maintainability problem when it needs to be changed. On the other hand, macros can be a maintainability issue as well. However, the macros only need to be introduced once for all architectures where 64/32-bit compatibility is implemented. Duplication, however, would be necessary for every architecture with support for backward compatibility.

7 Additional Work

The way Compatibility Mode support is designed, the kernel's task is to provide an environment compatible to L4/IA-32 to 32-bit threads, and to provide basic facilities for communication between 32-bit and 64-bit threads. It does not address the fact that the actual data sent and received by 32-bit threads often has a different format than the data used by 64-bit threads. Data conversion, if required, has to be done at user level because the kernel does not have any information about the data format. Therefore, 64-bit programs need to be designed specifically to be able to communicate with 32-bit threads.

7.1 User-Level Support Code

Apart from the user-level code necessary to detect that a certain IPC message originated from a 32-bit thread, the L4 user-level header files should provide assistance for the data conversion process. For the data types used in L4, we have added a separate header file containing 32-bit data types and conversion functions. The thread ID type is especially important since it is frequently used in communication. It is a bit field using different bit sizes in the 32-bit and 64-bit versions.

7.2 IDL^4

Data conversion usually needs to be done by hand, but it can be automated if enough information is available. User-level L4 IPC code is often generated from higher-level interface descriptions by the IDL^4 program. IDL^4 Interfaces consist of functions with parameters that all have specific data types. Thus, the IDL^4 program can check for signed integer and thread ID types in parameters, and insert the appropriate conversion code into the generated IPC stubs.

However, word and thread ID types have to be imported from the L4 header files, which implies that they are treated as user-defined types unknown to IDL^4. Therefore we modified the IDL^4 compiler in two steps: At first, we introduced new built-in data types for unsigned/signed words and thread IDs, which map exactly to the respective L4 types. Then we added an option to use custom marshaling for these types.

Actually, the IDL^4 compiler did already use word and thread ID types internally, for example when the user specified an interface name as a type. However, there were no specific classes in the IDL^4 type system for these types; instead the word type was actually a normal integer type, and the thread ID type was created dynamically as a custom type. To be able to use custom marshaling, we extended the IDL^4 class hierarchy with new classes for these types.

Both types use the same new marshaling class, which was derived from the „simple copy" marshaling class used for integer and custom types. Marshaling and unmarshaling is performed by calling a function whose name depends on the type name. Such functions are defined in the IDL^4 header files; they use the L4 functionality to check for 32-bit threads and convert the data appropriately. Unsigned words do not need any special treatment, signed words need to be sign-extended when they are converted from 32 to 64 bits, and thread IDs need to be converted using the support code from the L4 header files. Luckily, other types offered by IDL^4, such as strings and fpages, do not need to be converted.

To ensure that 32-bit and 64-bit tasks using the same IDL^4 specification are really compatible, the new „compatibility" option of IDL^4 needs to do more than just activate marshaling for word and thread ID types. By default, interfaces are not compatible, since IDL^4 ignores the register structure of L4 IPC messages, and treats the message registers as a block of memory instead. Data transfer between 32-bit and 64-bit threads is done on a per-register basis, creating or deleting "holes" in the message, respectively. Therefore, if the compatibility option is activated, all data types must have machine word size, so each register holds exactly one argument. No custom data types or integers of arbitrary size may be used. Structures and sequences could be supported in theory if they only use elements of word size, but this is not implemented at the moment.

As a test case, we ported the resource monitor of the L4Ka virtualization project to AMD64. It communicates with a wedge installed in the address space of the guest operating system via IDL^4. The ported monitor can load an unmodified 32-bit wedge and guest operating system and serve all IDL^4 requests issued by the wedge.

8 Conclusions and Future Work

The objective of the study thesis, implementing AMD64 Compatibility Mode support in L4Ka, was completed successfully. Some minor issues related to the virtualization project remain. Performance and additional complexity added to the kernel are in an acceptable range, but performance could be improved further by implementing a fast path for 32-bit IPC.

There are multiple alternatives for the design of this support. The choice of a complete kernel-level solution is unusual for microkernel design, but has some major advantages. Comparison of all alternatives was outside of the scope of this study thesis, but can be an interesting ground for future research.

Kernel support for Compatibility Mode is not sufficient to achieve compatibility between 32-bit and 64-bit applications. Conversion of communication data must happen at a higher level. IDL^4 has been adapted to fulfill this task, and tested by porting the virtualization resource monitor to AMD64. Other systems based on L4 may require more complex data conversion facilities, thus the modifications to IDL^4 leave some room for further development.

References

[1] Advanced Micro Devices, Inc. AMD64 Computing Platform.
http://www.amd.com/us-en/Processors/TechnicalResources/0,,30_182_869_875,00.html

[2] Intel Corporation. Intel® 64 Architecture.
http://www.intel.com/technology/intel64/index.htm

[3] Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 1: Application Programming. Revision 3.12, September 2006
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24592.pdf

[4] Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 2: System Programming. Revision 3.12, September 2006
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24593.pdf

[5] Advanced Micro Devices, Inc. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. Revision 3.12, September 2006
http://www.amd.com/us-en/assets/content_type/white_papers_and_tech_docs/24594.pdf

[6] Intel Corporation. IA-32 Intel® Architecture Software Developer's Manual Volume 3: System Programming Guide. January 2006
http://www.intel.com/design/Pentium4/manuals/253668.htm

[7] Intel Corporation. Intel® Itanium® Architecture Software Developer's Manual Volume 1: Application Architecture. Revision 2.2, January 2006
http://www.intel.com/design/itanium/manuals/245317.htm

[8] L4.
http://l4hq.org/

[9] Universität Karlsruhe System Architecture Group. L4 eXperimental Kernel Reference Manual Version X.2. Revision 6, August 2006
http://l4hq.org/docs/manuals/l4-x2-20060810.pdf

[10] Universität Karlsruhe System Architecture Group. The L4Ka Project.
http://l4ka.org/

[11] Universität Karlsruhe System Architecture Group. L4Ka Virtual Machine Technology.
http://www.l4ka.org/projects/virtualization/

[12] Universität Karlsruhe System Architecture Group. L4Ka::Pistachio/amd64.
http://l4ka.org/projects/pistachio/amd64/

[13] NetBSD/amd64.
http://www.netbsd.org/Ports/amd64/

[14] Linux.
http://www.kernel.org/

[15] Ovidiu Dobre. Multi-Architecture Operating Systems. October 2004
http://i30www.ira.uka.de/teaching/thesisdocuments/l4ka/2004/dobre_dt_multi-architecture-os.pdf

[16] Jochen Liedtke. On μ-Kernel Construction. December 1995
http://i30www.ira.uka.de/research/documents/l4ka/1995/ukernel-construction.pdf