

The Cost of IPC: an Architectural Analysis

Isaac Gelado, Javier Cabezas, Lluís Vilanova and Nacho Navarro
Departament d'Arquitectura de Computadors. Universitat Politècnica de Catalunya
C/ Jordi Girona 1-3, Campus Nord. 08034 Barcelona, Spain
e-mail: {igelado,jcabezas,vilanova,nacho}@ac.upc.edu

Abstract— μ -kernels based operating systems provide a modular design enabling scalability, extensibility, and portability not typically found in monolithic or conventional operating systems, and, even more important, they deliver fault-tolerance and security to computing environments. However, the low performance delivered has been the main barrier to an extended use of this design in operating systems.

μ -kernels heavily rely on sending IPC messages between processes, some of which implement memory management, device drivers and other abstractions such as filesystems or sockets. The high cost of IPC mechanisms is the key point to explain its poor performance.

In this paper we present an empirical evaluation of the evolution of the major performance limiting factors the computer architecture introduces in IPC: system calls, memory copy and address space switches. Our work shows the increasing architecture cost of these operations, especially in the privilege mode changes and address space switches, as a big challenge for future processors to deliver high performance and fault-tolerant secure systems based on a μ -kernel design.

I. INTRODUCTION

Security and fault tolerance are two major concerns on current systems. In the 80s, μ -kernel based operating systems were proposed. This paradigm showed to provide more secure and fault tolerant systems, at the cost of reducing the overall system performance. This is the main reason why currently few operating systems are based on μ -kernels.

In μ -kernel based operating systems, the kernel only provides few abstractions: threads (or any other processor-like abstractions) and address spaces. Filesystems, sockets, memory management or device drivers are implemented as user-level processes. Inter Process Communication (IPC) is thoroughly used in this kind of systems. Every system call usually involves several IPC messages. First, a running application sends a message to a certain server to request, for instance, reading a file. Then, the filesystem server, sends a message to the device driver to read the necessary data from the disk. Next, the disk device driver gets the data and sends it back to the filesystem server with another IPC message. Finally, the filesystem server sends yet another IPC message with the requested data to the application. Short time after μ -kernels were proposed, the high cost of IPC showed as the main factor of the poor performance of this kind of systems.

This work has been supported by the Ministry of Science and Technology of Spain and by the European Union (FEDER) under contract TIN2004-07739-C02-01, the HiPEAC European Network of Excellence and the SARC European Project. All products or company names mentioned herein are the trademarks of registered trademarks of their respective owners.

In 1994 Jochen Liedtke presented a study of different techniques to improve IPC [10]. He studied IPC from three different levels: architecture, algorithms and interfaces. Liedtke proposed a set of optimizations which speeded IPC up by a factor of 10x. Despite this huge performance improvement, only one of the three current most popular operating systems, MacOS X, is using a μ -kernel. The other two, Microsoft Windows and GNU/Linux, are still based on huge monolithic kernels. Nevertheless, the MacOS X implementation is based on a monolithic user-level server which implements all the aforementioned user-level services, thus not following the architectural design based on isolation that μ -kernels provide. Moreover, Mac OS X is based on the Mach μ -kernel, which incorporates some performance critical device drivers, as the disk ones, inside the kernel space [18].

Since Liedtke's work, processor architecture has radically changed. Current processors implement out-of-order execution, speculation and highly optimized branch predictors. The memory wall has impeded prefetching techniques and other architectural support. However, when Liedtke evaluated IPC, computers used in-order processors with few aggressive architecture optimization techniques. In this paper we present a study of the evolution of the three architectural aspects Liedtke identified as the main bottlenecks for IPC: privilege level switches, address space switches and memory copy. We also analyze the impact of current architecture techniques over the IPC.

The rest of this paper has the following structure: in section II we present the background knowledge and our motivation; Section III describes the architecture mechanisms involved on IPC; Section IV shows the experimental results of our work; Section V draws the conclusions.

II. BACKGROUND AND MOTIVATION

Most of the operating system bugs are located inside the code implementing device drivers or high-level abstractions [4]. μ -kernel based operating systems, such as Mach [1], Chorus [13], Minix [8] or L4 [11], execute traditional operating system services, as filesystems or device drivers, at user-level. Thus, these systems, due to its design, are more secure and fault tolerant than monolithic ones. They are more secure, because in case of an attack exploiting an operating system bug, the attacker is constrained to execute user-level code in a given server. μ -kernels are more fault tolerant, because if a

bug makes a device driver to execute incorrect code, it will be killed by the kernel and re-launched again.

A. The Importance of IPC

In μ -kernel based operating systems, applications use cross address-space IPC to request services to other servers (either in the Trusted Computing Base - TCB, the trusted code which runs in userspace - or outside of it), so the IPC performance is critical in these systems [2]. Several techniques were proposed to improve the performance of IPC for the first μ -kernels [3], [6], however μ -kernels still performed poorly. In 1994, Jochen Liedtke presented a set of optimization for IPC message passing which speeded IPC up by a factor of 5 to 10x [10]. To design these optimizations, Liedtke analyzed IPC from three different levels: architecture, algorithms and interfaces. In his analysis of the architecture level he showed that calling the kernel (privilege level switch) and address space switches were the most time consuming parts for a *null IPC*. He also showed that message copying from the sender process to the receiver was another important performance penalty. This work drove the design and implementation of the L4 μ -kernel [11].

To the best of our knowledge, the performance of IPC from the architecture level has not been evaluated since Liedtke's work. However, current computer architectures are completely different from the in-order pipelined processors where IPC were evaluated more than ten years ago. We update that work by showing an evolution of the three architecture mechanisms Liedtke identified as main bottlenecks for the IPC performance: privilege level switches, address space switches and memory copy.

B. IPC Implementation

In μ -kernel based operating systems, user-level applications request services to the operating system or other servers by sending IPC messages. In this section we review how the architecture is involved in this process. Here, we will assume an IA-32 processor, since it is the most widely used. We use an example of an application sending an arbitrary message to drive our analysis, as can be seen on Figure 1.

First, the application builds the message, by copying the parameters to the processor registers for short messages or to a memory buffer for long messages (1). Usually, long messages are built as short messages with registers containing memory addresses of user-level buffers. This process does not differ from setting the parameters for a function call.

Then, the application executes a *system call*, which makes the processor to start executing kernel code at a predefined memory address (2,4). The processor provides special instructions to perform this action. In the IA-32 architecture it used to be implemented by the trap (or software interrupt) instruction `int`. Software interrupts in this architecture require accessing two different structures in memory: an interrupt table to get the new program counter and a descriptor table to get the new privilege level of execution. Due to the increasing cost of accessing memory, using `int` for this purpose introduces a huge performance penalty. To overcome these problems, Intel

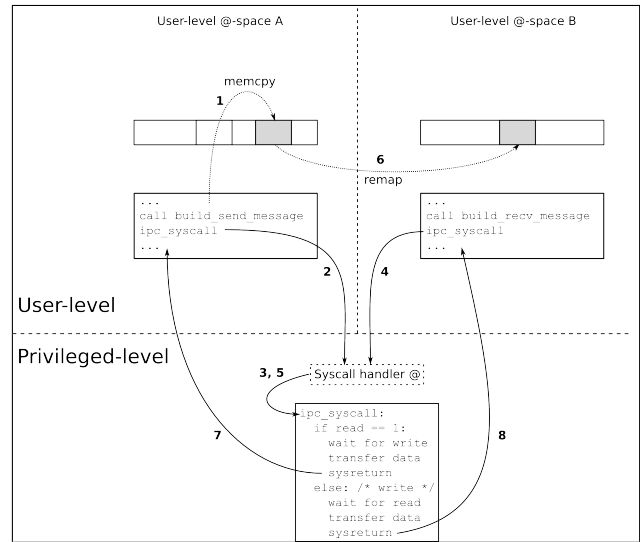


Fig. 1. Example of an IPC

introduced in its Pentium Pro chip the `sysenter` instruction to implement system calls. This instruction avoids one access to memory by getting the destination program counter and the privilege level from special purpose registers.

The kernel code executed after the trap saves the registers (3,5) containing the message and checks whether the destination is waiting for a message. Saving the sender registers is usually done by pushing their contents into the process kernel stack. However, architectures that support register windows, this operation simply moves the register frame, as any other function call. For long messages, the kernel copies the message from the sender address space to the receiver's one (6). This copy is usually done by temporary mapping the receiver's communication window inside the sender's address space, so a intermediate copy to the kernel memory is not necessary [10].

Then, the kernel switches from the sender address space to the receiver one and restores the contents of the registers. The address space is switched by changing the processor page table. Typically, this requires flushing the Translation Lookaside Buffer (TLB). Architectures that use virtual addresses in one or more levels of the memory hierarchy, require flushing the cache right after the address space switch to maintain memory protection.

Finally, the kernel switches again to user-level to start executing the sender's predefined routine for getting IPC messages (7). In the IA-32 architecture, it used to be done by executing the `iret` instruction, which also requires accessing twice to memory. The Pentium Pro chip introduced the `sysexit` instruction to return to the user-level without accessing to memory.

C. Capability-based OSes: an example of IPC stressing

As we have already said, μ -kernel operating systems provide better means for security and fault isolation, as well as a clearer modularity because each process must offer a given communication interface through its IPC mechanisms.

Capability-based operating systems enforce security by requiring applications to get a special token to access whatever resource in the system. This token designates an object and gives the program the authority to perform a specific set of actions on that object (i.e. reading or writing). This kind of token is called a capability, which was introduced in 1965 [5]. Additionally, more than one capability can point to the same object, but each conveying different access properties. A simplified way of understanding a capability is like a *file descriptor* when accessing a file in a UNIX-like system. In this example, using the system call `read (fd, buf, size)` is like performing the *read* operation on the file named by *fd* (the capability) using the arguments *buf* and *size*.

In this kind of systems, capabilities provide a substrate for strong security, as some have already proved in production systems [16] as well as in research ones [9]. The goal is to drive all the system parts to have a set of capabilities which is as smallest and specific as possible, because a program will not be able to have access beyond that conveyed by its capabilities. This is known as the *Principle of Least Privilege* (POLP) [14], [12]. Capabilities also render as unnecessary the existence of privileged users to access some special files or perform special tasks, as the running program has *all and only* the privileges to do its task (e.g. updating the user password or rebooting the system) and thus avoids having all the user's privileges (e.g. access to the whole home directory or the network) or the necessity to use `setuid` programs (a typical root of system vulnerabilities) as well as provide means for selectively revoking access to a given object.

In addition, this kind of systems have some interesting mathematical implications related to security. Systems can be designed to provide a static view of the whole authority relationship graph between applications even before running them, thus providing means for checking in advance the range of applications' permitted accesses through formal verification [17].

As capabilities are implemented by the μ -kernel and are the *only* way of accessing a capability-protected object, every use of them requires entering the privileged mode (a pure capability-based OS would have a single system call, which is performing an operation on a given capability). This way, performing an operation on a kernel-implemented capability would require a privilege level switch, while performing an operation on a user-implemented capability would require a whole IPC round trip.

Figure 2 shows how a read operation is performed in a capability-based operating system. In the figure, each capability designates a recipient process and conveys certain access permissions to a recipient-designated internal object (non-dotted small circles are capabilities held by a running process, while the dotted ones represent that some process hold a capability to that object with some access permissions). This simple example results in 4 IPCs (which lead to 8 system calls).

This scheme can be improved by the use of *derived capabilities*, which shortcut this path on subsequent calls, thus

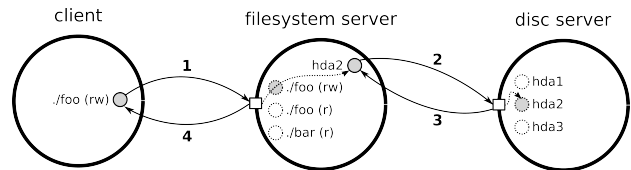


Fig. 2. Example of reading a file in a capability-based system

reducing the amount of necessary IPC calls if the operation is to be repeated.

This stressing use of IPC is one of the reasons why μ -kernels have been discouraged so many times, although it has been demonstrated that IPC in μ -kernels can be very fast [10], [7].

III. ARCHITECTURAL ANALYSIS

A. Privilege Level Switches

Current processors implement privilege level switches in different ways. In this section we review some of the processor design decisions that affect the performance of privilege level switches. Some processors implement privilege level switches as software traps, for instance the PowerPC, whereas others provide dedicated instructions to call the operating system, as the IA-32 processors.

When the processor executes a privilege level switch instruction, it starts fetching instructions that will be executed at a different privilege level. Two different implementation approaches are possible:

- Wait for all the current privilege level instructions to finish before start executing the new ones.
- Add a tag to every decoded instruction to label the privilege level at which it is being executed.

The former simplifies the pipeline implementation, since a single privilege level register is needed to check every instruction. The latter requires additional logic during the decode stage to mark new fetched instructions with the new execution privilege level. However, the big performance penalty due to the pipeline stall justifies using a privilege instruction tag for every instruction.

System calls have two main characteristics. First, a system call always implies a privilege level switch. Second, the operating system kernel usually implements a unique entry point for all system calls because the required actions after entering the system are independent of the code that will be later executed. Once these common instructions are executed, the kernel code calls one or another function depending on the system call number.

Typically, trap instructions are used to call the system. It means, the processor deals with hardware interrupts and system calls in the same way. However, hardware interrupts do not necessarily require a privilege level switch. For instance, a user-level device driver can serve the interrupts generated by the device that it is handling. We will show that using the same mechanism for hardware interrupts and system calls harms the system performance.

Several processors, as the PowerPC, assume that interrupts are only handled by kernel code. Thus, whenever an interrupt is received by the processor it jumps to a predefined memory address and starts executing code at the most privileged execution level. Such a design prevents the system from executing user-level code whenever an interrupt is received, so the kernel must get every interrupt and send an IPC message to the user-level process that actually handles it. Other architectures, as IA-32, allow handling interrupts directly at user-level by using an interrupt descriptor table, which is hosted in memory. Every time an interrupt is sent, the processor reads the interrupt descriptor from memory and determines the next instruction to execute and the privilege level. In this design, usually the processor incurs on a cache miss when it gets the interrupt descriptor from memory, to determine the privilege level, which is unnecessary for a system call.

Intel added the `sysenter` instruction to IA-32 processors to exploit the two characteristics of system calls we exposed. This instruction starts executing code at the most privileged level using as program counter the value stored in a special purpose register.

B. Address Space Switches

User-level applications have their own virtual address space. Whenever an IPC message is sent from one application to another, the kernel has to switch from the sender address space to the receiver's one. An address space switch is typically done by invalidating the TLB and changing the page table used by the processor for address translation.

Applications usually share portions of their address space, for instance shared libraries can be mapped on the same virtual addresses for all applications. In Linux, the kernel code and data is mapped on the same virtual addresses for every application. Processors exploit this high level of memory sharing by allowing certain page table entries to be pinned, thus a TLB invalidation does not evict these entries. This technique avoids unnecessary TLB misses during an address space switch because the entries mapping the code that implements the address space switch remains in the TLB. Prefetching techniques for the TLB have been proposed for user-level applications [15]. Although after an address space switch a huge number of TLB misses happen, no architecture support exists for TLB prefetching in this case. A prefetching technique for this situation would be based on a history buffer, so given the first TLB miss after an address space switch, the next ones can be effectively predicted.

Some processors implement tagged TLBs, where each TLB entry has a tag field that identifies a single address space. Usually, TLB tags are few bits long, so the operating system has to virtualize them. The operating system multiplexes the usage of the TLB tags among all the processes in the system. Whenever a process without a TLB tag is scheduled, the operating system selects an already used tag value (for instance using a LRU algorithm) from other process, marks this process as untagged, invalidates the TLB entries containing that tag and, finally, assigns that tag to the scheduled process. Tagged

TLBs reduce the cost of address space switches [11] but might increase the context switch time.

First-level cache design also affects the performance of address space switches. Tags in the cache can be either physical or virtual memory addresses. The former approach does not require any address translation on every cache access. However, if the cache uses virtual addresses, the contents of the cache must be flushed to ensure memory coherence and protection whenever an address space is switched. Similar mechanisms to those previously described for the TLB could be used for architectures using virtual addressed first-level caches.

C. Memory Copy

Long-messages require the kernel to copy data from memory in the sender's address space to a buffer in the receiver's address space. In μ -kernels such as Mach, this process requires copying the data to a kernel buffer, then perform an address space switch and, finally, copying the data from within the kernel to the receiver. To avoid this intermediate data copy, the kernel temporarily maps the receiver's buffer inside the sender's address space [10]. Thus, in this process the actual data copy is the most time consuming part.

Depending on the size of the data to be copied and how it is aligned in memory, the kernel uses different strategies, such using SIMD instructions. Preemptible μ -kernels use the Direct Memory Access (DMA) controller to copy the data, while other threads are scheduled. Despite this optimizations, the memory barrier increases the weight of data copy in IPC.

The main bottleneck in this process are the cache misses produced while writing to the destination memory. The cost of these cache misses is later recovered by the receiver code, since it usually accesses immediately to this data when the cache uses physical addresses. Caches using virtual addresses can not recover this cost because a cache flush is needed after the address space switch to start executing the receiver code.

IV. EVALUATION

We have run a set of tests in a set of Intel processors, which try to cover a wide range of them, from the Pentium II to the Core 2 Duo. Table I shows the characteristics of the machines we have used in our experiments, sorted by launch date, which is the same order we will use on all our next figures. These processor families represent two different branches of microarchitectural development, as the Pentium IV family is a break with the previous families, and disappeared to get development back again into the Pentium III (where Pentium M and the Core evolved from).

To evaluate the current architecture support for IPC, we implemented a minimal kernel and set of user-level applications that do all the measurements. The reason to avoid using any currently available kernel is to minimize the undesirable effects introduced by code which is superfluous to our objectives,

¹These are off-chip caches working at 50% of CPU-speed

²Pentium IV processor use a Trace Cache

Code	CPU	Clock	Stages	Width	L1\$ (i+d)	L2\$	FSB	Mem	MClock	Year
PII	Pentium II Deschutes	400MHz	14	4	16+16KiB	512KiB ¹	100MHz	SDRAM	100MHz	1998
PIII	Pentium III Katmai	450MHz	14	4	16+16KiB	512KiB ¹	100MHz	SDRAM	133MHz	1999
PIV	Pentium IV Prescott 3.0E	3.00GHz	31	3	12 ² +16KiB	1MiB	800MHz	DDR	400MHz	2004
PM	Pentium M Dothan 740	1.6GHz	~14	3	32+32KiB	2MiB	533MHz	DDR2	533MHz	2005
PIV-2	Pentium IV Cedar Mill 631	3.00GHz	31	3	12 ² +16KiB	2MiB	800MHz	DDR2	533MHz	2006
CoreSolo	Core Solo Yonah T1300	1.66GHz	12	4	32+32KiB	2MiB	667MHz	DDR2	533MHz	2006
CoreDuo	Core Duo Yonah T2600	2.16GHz	12	4	32+32KiB	2MiB	667MHz	DDR2	667MHz	2006
Core2Duo	Core 2 Duo Conroe E6600	2.4GHz	14	4	32+32KiB	4MiB	1067MHz	DDR2	667MHz	2006

TABLE I
HARDWARE CHARACTERISTICS OF THE MACHINES USED FOR EVALUATION

which is to measure the bare metal costs of all the mechanisms involved in IPC.

The only noticeable overhead of our measurements is introduced by the timestamping routine, as well as any function call in the measured code which is introduced by the compiler (which we tried to minimize by using hand-written assembly code as well as inlined functions). Our timestamping routine is based on the unprivileged instruction `rdtsc` that incurs in a function call (which adds an overhead when timestamping at the end of the test) as well as its return and saving the 64bit result in a variable (which adds an overhead when timestamping at the start of the test). In addition, when measuring the memory transfer rates, we use the `cpuid` instruction (which compels us to save and restore a pair of registers out of the four it uses, as well as assigning a value to `eax`) just before the `rdtsc`, to ensure the timestamping comprises the cost of all the memory copy (otherwise, the processor can execute it before finishing the memory copies due to the lack of data dependencies).

All the shown results are the mean of a thousand of executions, and show three different values each for three different experiment configurations. First *hot* caches, which means that no special action despite from measuring the cost in cycles of the test is taken. The second one is *data cold* caches, where we have written a piece of code that flushes the data level one cache prior to the test. Finally, *cold* caches, which also flush the level one instruction cache. Table II shows the raw numbers of our tests.

Due to the different microarchitectures implemented in the processors we are evaluating, getting the number of cycles for a given operation provides little information. Our study is focused in the evolution of the architecture mechanisms respect to the rest of the processor. Thus, we first run the SPECint set of the SPEC2006 benchmarks to estimate the overall performance of each processor, as shown in Figure 3. From now on, when talking about a result, it is normalized to show its relation factor to this test ($result_cycles/test_cycles$), which we will refer to as the *performance ratio*.

The figure shows performance improvements between successive families of processors. The most significant performance jump is found between the PIII and the PIV (probably caused by the inclusion of the L2 cache into the chip³), whereas there is a slowdown in terms of performance gain between the PIV and the CoreSolo processors. From the

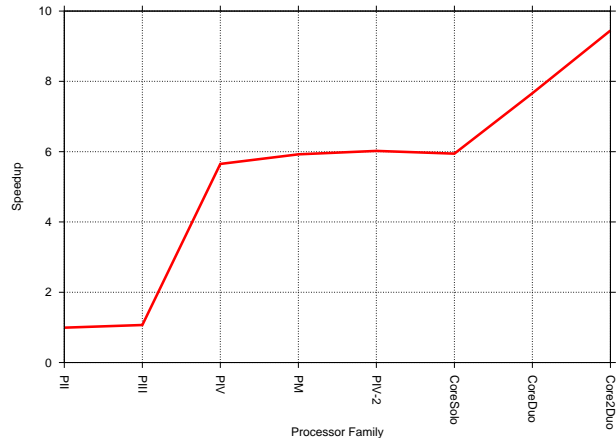


Fig. 3. Performance on SPEC2006int benchmarks

CoreSolo to the Core2Duo a linear speedup can be noticed.

A. Two flavors for Privilege Level Switches

We have measured four different magnitudes to evaluate the evolution of privilege level switches. These are, respectively, the cost of a software interrupt (`int`) and returning from it (`iret`) and the cost of a system call (`sysenter`) and system exit (`sysexit`).

This set of tests first flush the pertinent caches (depending on the type of test - *hot*, *data cold*, *cold* -), get a timestamp, perform the desired system entry or exit instruction, and once changed the privilege level (user \rightarrow system for `int` and `sysenter`; system \rightarrow user for `iret` and `sysexit`), a timestamp is once again retrieved. The time lapse between those two timestamps is the value used to calculate our results in Figure 4.

As we can see, the *data cold* and *cold* lines are much closer on the system entry/exit figures than in the software interrupt ones, due to the fact that the latter access memory to get both the entry information (privilege level and `eip` from the interrupt descriptor table, as explained in Section III-A) as well as the exit information (return `eip` and `esp` are stored in the stack). Meanwhile, the former (system entry/exit) use registers to avoid those accesses on entering (special-purpose ones) as well as when exiting (`eax` and `edx`), thus the *data cold* and *cold* lines are almost equal, as only a single memory access is required. Nevertheless, the big difference between

³The Pentium III Coppermine included the L2 cache into the chip, but we have not had access to such processor in this evaluation

Code	Experiment	int	iret	sysenter	sysexit	@space switch	memcpy 4KB	memcpy 8KB	memcpy 16KB	SPEC2006int (secs)
PII	<i>hot</i>	160	219	84	107	118	1045	2255	24420	85.7418
	<i>data cold</i>	275	324	84	162	118	9321	18572	30661	
	<i>cold</i>	316	475	111	339	118	9335	18586	30675	
PIII	<i>hot</i>	160	219	82	104	114	1057	2179	25433	79.6555
	<i>data cold</i>	281	331	82	166	117	10151	20374	32826	
	<i>cold</i>	321	493	111	344	114	10165	20328	32825	
PIV	<i>hot</i>	521	645	206	233	508	2154	3584	6272	15.040
	<i>data cold</i>	584	689	207	233	528	2060	3446	6111	
	<i>cold</i>	604	753	240	311	512	2032	3408	6039	
PM	<i>hot</i>	179	263	96	113	153	1017	1659	3083	14.3487
	<i>data cold</i>	197	290	96	114	144	2471	4720	9250	
	<i>cold</i>	247	378	130	157	144	2471	4724	9492	
PIV-2	<i>hot</i>	515	645	205	215	540	2014	3347	6224	14.1151
	<i>data cold</i>	584	696	220	232	548	2100	3647	6684	
	<i>cold</i>	635	790	245	315	542	2107	3625	6693	
CoreSolo	<i>hot</i>	292	354	180	184	230	1126	1763	3231	14.3030
	<i>data cold</i>	302	400	180	184	230	1805	3157	5873	
	<i>cold</i>	364	502	226	238	230	1804	3172	5870	
CoreDuo	<i>hot</i>	292	354	180	184	230	1121	1761	3223	11.095
	<i>data cold</i>	302	400	180	184	230	1788	3144	5898	
	<i>cold</i>	364	502	225	239	230	1789	3143	5896	
Core2Duo	<i>hot</i>	314	361	150	163	247	896	1472	2797	8.9976
	<i>data cold</i>	325	397	150	163	246	1599	2852	5416	
	<i>cold</i>	326	441	162	209	247	1606	2860	5421	

TABLE II
RAW NUMBERS FOR MEASUREMENTS (IN CYCLES)

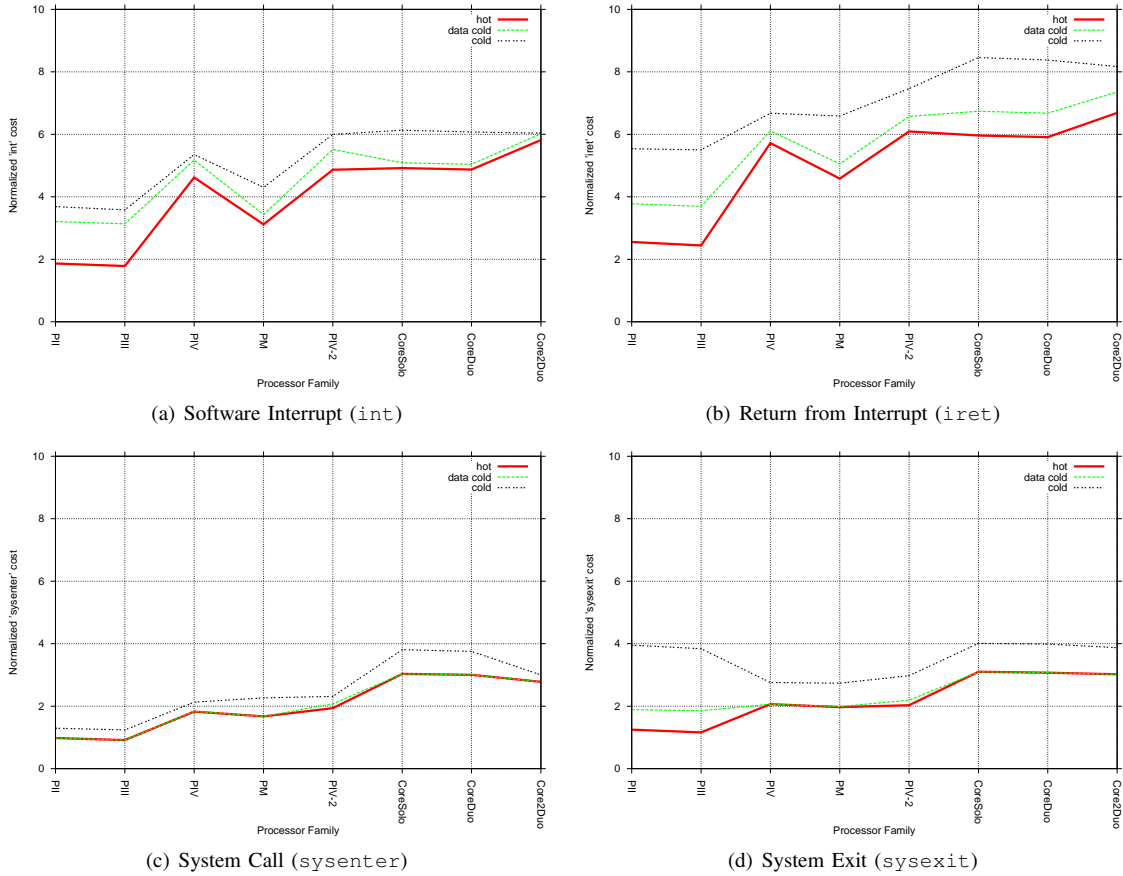


Fig. 4. Cost of entering and exiting the system

hot and *data cold* in PII and PIII in Figures 4(a), 4(b) are due to the fact that the L2 cache is outside the chip, and it has to be accessed after flushing the L1 data cache in the *data*

cold case. On the other hand, the anomaly between *hot* and *data cold* in PII and PIII in Figure 4(d) is due to the fact that before returning to user-level, we have to call a function to

save the timestamp into our ring of stored timestamps, which adds some extra misses to the data cache, which has to go outside the chip.

As we can see, the cost of performing these operations generally tends to increase on every processor family at the same pace of the increase on the stages of the pipeline, as all these operations flush the pipeline when executed (thus the PIV family - PIV and PIV-2 - has higher costs per cycle compared to the others). It should be noted the little difference between the relative performance of privilege level switches between PIV-2 and CoreSolo. The Core microarchitecture implements a less deeper pipeline than the PIV, so the processor is stalled during less number of cycles on a privilege level switch. However the PIV-2 runs at twice the frequency than the CoreSolo, so less time is spent during the privilege level switch. Finally, another interesting result is the cost of `int` and `iret` between the CoreDuo and the Core2Duo. The latter has two more stages in the pipeline and runs at a higher frequency, but the frequency has relatively increased less than the number of stages. Thus, the relative cost for both instructions is higher.

The results for the *cold* experiments show that the privilege level switch depends on the state of the L1 cache. After flushing the instruction cache, all the code that is going to be executed needs to be fetched again to the L1 instruction cache. The only difference in the *cold* results is in the PII and PIII when exiting the system (`iret` and `sysexit`), where like aforementioned we have to call a C function before exiting to save the timestamp that marks the starting time of the exit, so there are some overhead accesses to the L2 cache which are not present when entering the system (those timestamps can be saved in a temporary user-level variable and later stored in the timestamp ring outside the timestamping path).

As a summary, it is clear that the system entry and exit mechanisms have not evolved in the micro-architecture at the same pace of the brute force performance. The `sysentry` and `sysexit` have relatively slowed down up to 2x, while the relative slow-down for `int` and `iret` is almost 6x.

B. Caches and TLBs

This test consists of a change of the page table directory (write to register `cr3`), which has some important consequences, followed by timestamp retrievals, as well as the corresponding cache flushes before the starting timestamp retrieval.

The consequences of switching the page table are:

- 1) Wait for all the in-flight instructions in the pipeline before the switch to commit.
- 2) Flush all the instructions in the pipeline which come after the switch.
- 3) Flush all the entries in the TLB which have not been pinned.

As is usual on all types of kernels, we have pinned all the kernel-level pages with the *global* bit in the page table entries, thus the page table switch only flushes the TLB page entries owned by the process previously executing.

The results shown in Figure 5 have a clear relation with the number of stages of the pipeline, as the larger it is, the more costly it is to flush and commit the in-flight instructions (which is specially notable in the Pentium IV family).

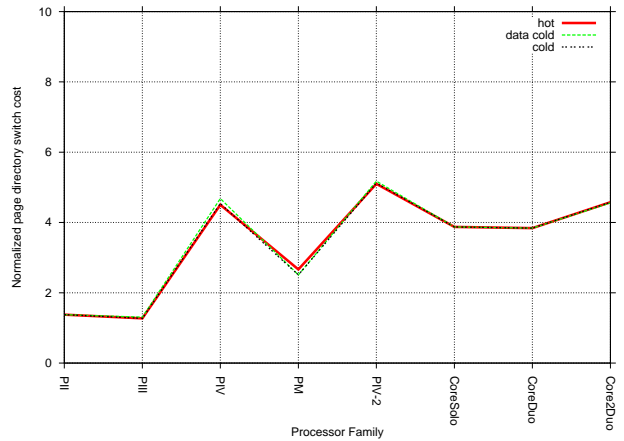


Fig. 5. Cost of changing address-space

This way, apart from the PIV and PIV-2 peaks, with a similar number of stages in the other family tests, the *performance ratio* has increased. The PM has about 14 stages in the pipeline, but it has a smaller number of functional units (a smaller width). Thus, the relative cost for the PM is lower than the CoreSolo, CoreDuo and Core2Duo.

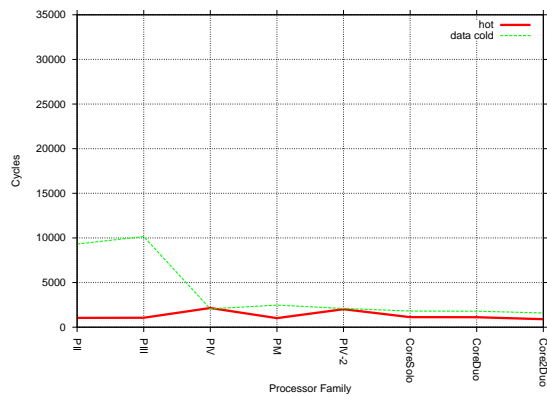
The differences between the three test types are not significant, as the page table directory switch is based on a single write to the special register `cr3`, and this is independent of the state of the cache. The only thing that could make a slight difference is the type and number of in-flight instructions at the time of the switch.

C. Memory Copy

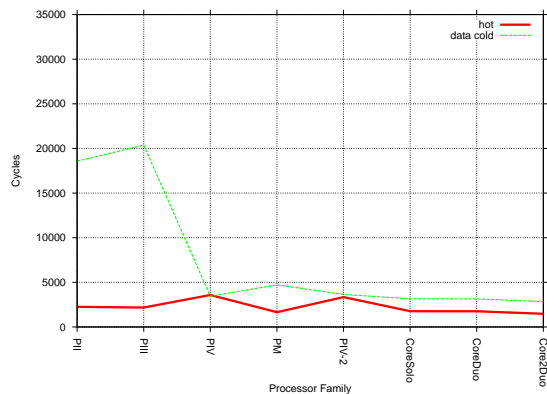
The IPC messages sent between processes can have different sizes. Small enough messages can be transferred in the general purpose registers of the processor. The cost of this communication is tied to the system entrance and page directory switching costs which have been evaluated previously. Bigger ones, are packed into memory and copied to the receiver's address space. Thus, we have evaluated the cost of memory copies in the studied platforms. We have chosen message sizes from 4 to 16KiB because those are the usual sizes read from a disk or network connection.

This set of tests first flush the pertinent caches (depending on the type of test - *hot*, *data cold*, but not *cold* as this test is not affected by the instruction cache -), get a timestamp, perform the desired message copy from one memory buffer to another, and a timestamp is once again retrieved. The time lapse between those two timestamps is the value used to calculate our results in Figure 6. We have used the special instruction `cpuid` before retrieving the second timestamp; this guarantees that all the memory instructions have finished.

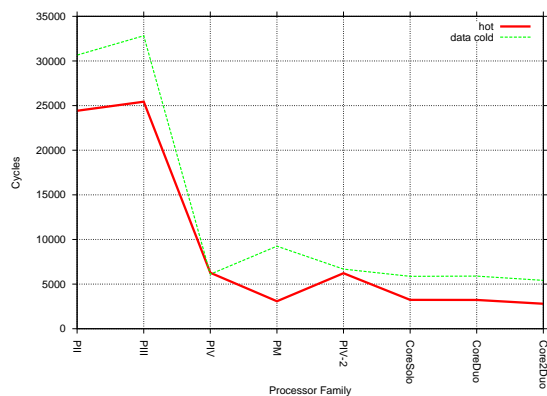
In these tests we are not using normalized numbers, but the cycles spent in each test run. We do not normalize these



(a) memcopy of 4KiB



(b) memcopy of 8KiB



(c) memcopy of 16KiB

Fig. 6. Cost of copying 4, 8 and 16KiB

measurements because the memory copy is more related to the main memory hierarchy than to the processor performance.

The memory hierarchy is the main factor to be considered in message copies. If a message fits in the first level cache the cost is bounded by its latency, which is usually very low. Bigger messages are partially written to the second level of the cache, which is much slower. The main memory technology would be another important factor for messages bigger than the second level cache.

As we can see in the three graphs in Figure 6, with a *hot* cache, the cost of 4 and 8KiB copies (Figures 6(a) and (b)) is almost flat, since the message fits in the L1 cache. However, for 16KiB (Figure 6(c)), the Pentium II and Pentium

III processors show much worse results since they have to access their (off-chip) L2 cache. The Pentium IV processor, which has a 16KiB data cache too, performs noticeably worse than the more recent processors in all the cases due to its small L1 data cache and higher cache latency.

The experiments done with cold data caches show a similar behavior. The Pentium II and Pentium III spend much more cycles than the other processors since they always have to access the (off-chip) L2 cache. The *data cold* results for the two tested Pentium IV processors are the same numbers as those using *hot* caches. We can speculate that this effect is due to the prefetching technique implemented by the PIV chip. The rest of processors show a significant gap between the different configurations, due to the forced access to the L2 cache.

To sum up we can say that memory copies have been improved in the last processors due to the increased size of on-chip memory caches. From now on the copy of messages sized from 4 to 16KiB could only get faster if lower latency caches are developed. Bigger L1 caches would be needed in order to make bigger messages fit in them when being copied using *hot* caches while current L2 caches (2 or 4MiB) are big enough for holding the messages in copies performed with a cold cache.

V. CONCLUSIONS

In this paper we have evaluated the evolution of the architectural mechanisms used during IPC communication. Despite algorithmic optimizations, IPC heavily relies on the architecture support for privilege and address space switches and in a less important manner, on memory copy. The architecture should implement these mechanisms efficiently in order to allow operating systems to implement fast IPC.

IPC is a critical part of μ -kernel based operating systems and μ -kernels are necessary to build more secure and fault tolerant systems. The importance of IPC is even bigger in future capability-based operating systems, which will really bring security to user desktop computers.

Our experiments show that the trend is that the cost of privilege level and address space switches is increasing compared to the general horsepower. Although applications' performance is still far from dependant on the system calls, if this relative cost continues increasing, the performance of certain applications will be limited by the cost of system calls. Memory copy during IPC benefits from architectural improvements as data prefetching. Any advance to solve the well-know memory wall will improve IPC as well.

A new set of architecture optimizations and mechanisms is required to have an efficient IPC implementation. The work we present in this paper shows a problem that our future research will try to solve.

ACKNOWLEDGEMENTS

We would like to thank all the anonymous reviewers for the insightful hints, which helped to improve this paper. We would also like to thank all the people at the GSO and the HPC groups at the Departament d'Arquitectura de Computadors at

Universitat Politècnica de Catalunya, as well as people at the Barcelona Supercomputing Center for their valuable help.

REFERENCES

- [1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: a new kernel foundation for unix development. In *Summer 1986 USENIX Conference*, pages 93–113. USENIX Association, 1986.
- [2] T. E. Anderson, H. M. Levy, B. N. Bershad, and E. D. Lazowska. The interaction of architecture and operating system design. *ACM SIGARCH Computer Architecture News*, 19(2):108–120, 1991.
- [3] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure calls. *ACM Transactions on Computer Systems*, 8(1):37–55, 1990.
- [4] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler. An empirical study of operating system errors. *ACM SIGOPS Operating Systems Review*, 35(5):73–88, 2001.
- [5] J. B. Dennis and E. C. V. Horn. Programming semantics for multi-programmed computations. Technical Report MIT/LCS/TR-23, Massachusetts Institute of Technology, 1965.
- [6] R. P. Draves, B. N. Bershad, R. F. Rashid, and R. W. Dean. Using continuations to implement thread management and communication on operating systems. *SIGOPS Operating System Review*, 25(5):122–136, 1991.
- [7] H. Härtig, M. Hohmuth, J. Liedtke, S. Schönberg, and J. Wolter. The performance of μ -kernel-based systems. In *Sixteenth ACM Symposium on Operating System Principles*, pages 66–77. ACM Press, 1997.
- [8] J. N. Herder, H. Bos, and A. S. Tanenbaum. A lightweight method for building reliable operating systems despite unreliable device drivers. Technical Report IR-CS-018, Dept. of Computer Science, Vrije Universiteit, Amsterdam, The Netherlands, January 2006.
- [9] B. Kauer and M. Volp. *LA.Sec Preliminary Microkernel Reference Manual*. Technische Universität Dresden, 1995.
- [10] J. Liedtke. Improving ipc by kernel design. In *SOSP'93: Proceedings of the ourteenth ACM Symposium on Operating System Principles*, pages 175–188. ACM Press, 1993.
- [11] J. Liedtke. On micro-kernel construction. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 237–250. ACM Press, 1995.
- [12] M. Miller and J. Shapiro. Paradigm regained: Abstraction mechanism for access control. In *Asian Computing Conference*, December 2003.
- [13] M. Rozier, V. Abrossimov, F. Armand, I. Boule, M. Gien, M. Guillemon, F. Herrman, C. Kaiser, S. Langlois, and P. Leonard. The chorus distributed operating system: some design issues. In *Workshop on Micro-kernels and Other Kernel architectures*, pages 39–70. USENIX Association, 1988.
- [14] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. In *Proceedings of the IEEE 63(9)*, page 12781308, September 1975.
- [15] A. Saulsbury, F. Dahlgren, and P. Stenstrom. Recency-based tlb preloading. *SIGARCH ACM Computer Architecture News*, 28(2):117–127, 2000.
- [16] J. S. Shapiro, J. M. Smith, and D. J. Farber. EROS: a fast capability system. In *Symposium on Operating Systems Principles*, pages 170–185, 1999.
- [17] J. S. Shapiro and S. Weber. Verifying the EROS confinement mechanism. In *Symposium on Security and Privacy*, pages 166–176, 2000.
- [18] A. Tanenbaum. A comparison of three microkernels. *The Journal of Supercomputing*, 9(1):7–22, 1995.