6th Workshop on Hot Topics in Operating Systems (HotOS)
May 5-6, 1997, Chatham (Cape Code), Massachusetts

Achieved IPC Performance

(Still The Foundation For Extensibility)

Jochen Liedtke * Kevin Elphinstone † Sebastian Schönberg ‡ Hermann Härtig‡
Gernot Heiser† Nayeem Islam* Trent Jaeger*

Abstract

Extensibility can be based on cross-address-space communication or on grafting application-specific modules into the operating system. For comparing both approaches, we need to explore the best achievable performance for both models. This paper reports the achieved performance of cross-address-space communication for the L4 μ-kernel on Intel Pentium, Mips R4600 and DEC Alpha. The direct costs range from 45 cycles (Alpha) to 121 cycles (Pentium). Since only 2.3% of the L1 cache are required (Pentium), the average indirect costs are not to be expected much higher.

1 Motivation: extensibility

"Extensibility" is a relatively new buzzword in OS research. Nevertheless, the requirement for extensibility is neither specific to operating systems nor new. Editors are extended by macros associating new functions to keys, programming languages are extended by libraries, database systems are extended by application-specific functions, word processing systems are extended by customized texts, *et cetera, et cetera.*

What makes extensibility an OS-specific topic?

Security and safety!

When extending an operating system by a new or modified service, we require that (a) the service can be introduced only for selected clients and that (b) a potential malfunction of the new service affects only those clients that use it. In accordance to (a), diffferent clients can, of course, use different services for the same event. (a) is difficult because the operating system controls central resources; (b) is difficult because (i) these resources are critical with respect to the correct functioning of the entire system and (ii) services need to be protected from each other making uncontrolled interference impossible.

The multiple-server approach

An obvious (and well-known) solution: use multiple servers, protect them by classical operating system mechanisms, i.e. address spaces, and make them freely attachable to applications. Basically, that is the μ-kernel approach, pioneered by Amoeba, Mach and Chorus, further developed by L4 [Liedtke 1995], Fluke [Ford et al. 1996] and others.

This method is best-suited to incorporate general, well-known software techniques for extensibility. Functionally, it is most flexible and most general.

However, good performance of the multiple-server technique requires that the direct and indirect costs of cross-address-space communication (including address-space switching) are sufficiently low. Unfortunately, years ago, IPC was considered to be expensive.

The grafting approach

A further solution is to graft additional modules into the monolithic server (the operating system). Early applications of this technique are widely used but insecure and/or of limited flexibility: mounting new file systems, adding new device drivers *et cetera.*

New research projects, in particular Spin [Bershad et al. 1995] and Vino [Seltzer et al. 1996] experiment with compile-time and run-time (compiler-supported) security for "grafted" kernel components. Spin [Bershad et al. 1995] inserts type-checked modules into the kernel; Vino [Seltzer et al. 1996] permits unsafe grafts and controls them by sandboxing and transactions.

Necula and Lee [1996] developed a very interesting method of controling grafts by mathmatical proofs. However, currently this method is probably not (yet?) applicable to non-toy grafts.

*IBM T. J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532, USA, jochen@watson.ibm.com
†School of Computer Science, University of New South Wales, Sydney, 2052, Australia, kevine@cse.unsw.edu.au
‡Department of Computer Science, Dresden University of Technology, Hans-Grundig-Str., Dresden, Germany, sebastian.schoenberg@inf.tu-dresden.de

Which approach should be preferred?

There are two scientific criteria for comparing the multiple-server against the grafting approach: functionality and performance. Liedtke [1995] showed that the μ-kernel-based approach (multiple servers in multiple address spaces communicate via IPC) is at least as flexible as to modify a monolithic server. This includes policy extensibility since a real μ-kernel is policy-free and permits to implement all policies at user-level. It is still not clear whether the reverse statement "modifying a monolithic server always gives the same flexibility" holds.

The second, and probably the more critical question, is performance.

Therefore, it is important to find the really achievable best performance of cross-address space communication. That is the topic of this paper. Section 2.1 reports the achieved best-case performance in the L4 μ-kernel on Intel Pentium, Mips R4600 and DEC Alpha systems. Section 2.2 analyzes indirect costs for average and worst cases.

Of course, this is only one side of the coin. Comparably substantiated and comparably analyzed performance results are also required for the grafting model. Currently, the reported numbers are *6 to 80 times worse* for grafting than the L4-based results (see section 3). However, there is no evidence how close the reported numbers are to the principally achievable performance.[1]

2 Achieved IPC performance

The L4 μ-kernel is currently implemented for Intel 486 and Pentium [Liedtke 1996], Mips R4600 and DEC Alpha 21164 processors [Schönberg 1996]. Intel versions of L4 are available since early 1996. In the meantime, Linux was ported to run on top of the 486 and Pentium L4 μ-kernels [Hohmuth et al. 1996]. According Mips and Alpha versions are forethcoming.

On the Pentium processor, a simple IPC transfers up to 3 registers (plus sender id) from the sending to the receiving thread. R4600 and Alpha permit up to 8 registers. More complex communication can use application-specific memory sharing or the ability of IPC to copy longer messages between address spaces.

[1] Remember what happened in the IPC case: for years IPC was reported to cost about 100 μs (independent of the processor), then it improved to 5 μs, now 1 μs.

	cache lines used	*cycles required*	
Pentium	12 of 512	121	166 MHz: 0.73 μs
R4600	19 of 1024	86	100 MHz: 0.86 μs
Alpha	17 of 512	45	433 MHz: 0.10 μs

Table 1: *Simple IPC performance.*

2.1 Direct costs

Table 1 summarizes the direct costs (space and time) for a simple cross-address-space IPC. Address-space switch does not require a TLB flush in either system. Alpha and Mips have tagged TLBs, for Pentium, a segment-based technique is used to emulate a tagged TLB. The Pentium requires 121 cycles for a simple IPC, about 35 cycles more than Mips, 75 cycles more than Alpha. Mips R4600 is a single-issue processor while Pentium and Alpha are dual-issue machines (in the absence of floating point operations). The additional costs for the Pentium processor are due to its slow kernel-trap instruction.

These nevertheless small numbers show that IPC can be regarded as a simple, basic operation. In a way, it is similar to a complex microprogrammed instruction. This is corroborated by the small amount of first-level cache consumed by IPC. Tables 2 and 3 give a detailed breakdown of cycles and cache lines required.

2.2 Indirect costs

The simple IPC implementation is small enough to permit an in-depth performance analysis of the indirect costs. Currently, we have a detailed understanding what happens in the Pentium implementation.

None of the three mentioned implementations flushes the TLB on an address-space switch. So indirect TLB costs can only occur when the μ-kernel uses virtual memory. Note that the Pentium does not support unmapped memory, i.e. operating system code and data is also part of virtual memory. This needs 4 TLB entries (of 96 entries) per IPC. Two of the entries are also used per incoming hardware interrupt, one is associated with the currently running thread. So we consider 3 of them to be always present. The fourth one is associated with the destination thread and should be present with a high probability if the destination is frequently accessed and we have no TLB thrashing situation. Since a TLB miss takes approximately 25 cycles, the average TLB-related indirect costs should not exceed 5 cycles.

	Pentium		**R4600**		**Alpha**	
	instructions	*cycles*	*instructions*	*cycles*	*instructions*	*cycles*
enter kernel mode (trap)	1	52	23	25	1	5
ipc code	43	23	47	50	60	38
segment register reload	4	16	–	–	–	–
exit kernel mode (ret)	1	20	9	11	1	2
total	50	**121**	79	**86**	62	**45**
	166 MHz: 0.73 μs		100 MHz: 0.86 μs		433 MHz: 0.10 μs	

Table 2: *Simple IPC, cycle costs.*

		Pentium		**R4600**		**Alpha**	
		cache lines	*L1 cache usage*	*cache lines*	*L1 cache usage*	*cache lines*	*L1 cache usage*
kernel code	(I-cache)	6	2.3 %	14	2.7 %	13	5.1 %
global kernel data	(D-cache)	2	0.8 %	1	0.2 %	0	0.0 %
thread kernel data	(D-cache)	2×2	1.6 %	2×2	0.8 %	2×2	1.6 %
total	(I+D-cache)	**12**	**2.3** %	**19**	**1.9** %	**17**	**3.3** %

Table 3: *Simple IPC, cache costs.*

Costs related to cache misses might be higher. The worst possible case involves second-level cache misses and bus blocking due to write-back bursts. However, this worst case is very unlikely: Write-back overhead is usually hidden by write buffers, and the second-level cache of at least 256 K should contain the 12 lines – that is 0.15% of the L2 cache – used for IPC. For a reasonable *very bad* case, we assume that per IPC

— the L1 cache contains *no* IPC-related data at all, i.e., that *maximum* L1 misses occur,

\+ no second-level cache misses occur, and

\+ cache refill is never delayed due to pending write-back operations.

Furthermore, we define a *bad* case where half of the IPCs perform *very badly* while the other half are best-case. This is a reasonable worst-case approximation for very short remote procedure calls. We assume that the calling IPC is always *very bad* as described above. However, the short remote procedure body will usually not conflict with the previously loaded 12 cache lines so that the reply is a best-case IPC.

To get an impression about the influence of the various memory systems, we measured *bad*-case and *very-bad*-case costs on a 90-MHz Thinkpad *without* an L2 cache, a 133-MHz Server with 256-K L2 cache and a 166-MHz PC with also 256-K L2 cache. For the bad cases, instruction cache and data cache were systematically flooded prior to each IPC so that no IPC-related code and data were left in the L1 cache. The according flooding overhead was subtracted from the total time. For all measurements, the cpu-internal clock register which is incremented per processor cycle was used. Table 4 shows the resulting IPC costs for the *ideal*, *bad* and *very bad* case. Surprisingly, even the Thinkpad without L2 cache shows reasonable performance, 166 cycles overhead for the *very bad* case. The 166-MHz machine shows a *bad* case overhead of 74 cycles. So we conclude that in general, short IPC mostly takes between 120 and 200 cycles.

Pentium						
	Thinkpad 760C		IBM PCS 320		IBM PC 750	
clock rate	90 MHz		133 MHz		166 MHz	
L2 cache	–		256 K		256 K	
ideal	121	1.34 μs	121	0.91 μs	121	0.73 μs
bad	204	2.27 μs	208	1.56 μs	195	1.18 μs
very bad	287	3.19 μs	295	2.22 μs	269	1.63 μs

Table 4: *Cycles per IPC, ideal and bad cases.*

A nice behaviour is that cache overhead decreases when IPC is used frequently. As a rule of thumb, the average cache-miss rate is expected to change by $x^{-\frac{1}{2}}$ when the cache size changes by a factor of x. Applying this rule to the 2.3% of cache consumption by IPC, would predict an increase of the cache-miss rate by a factor of $\frac{12}{1000}$. Assume that a system of programs communicating via IPC has a cache-miss rate of 5%. Then this rule would say that about $\frac{12}{1000} \times 5\% = 0.06\%$ are due to IPC. Although this is a rule of thumb and thus wrong in many concrete cases, it gives us some impressions about the order of non-magnitude.

Weird Ideas: There might be a problem that some particular software has *systematic* conflicts with the cache lines used by IPC. Since the IPC code is so small, it could be replicated for various cache lines. The μ-kernel could then from time to time randomly switch between them.

3 Comparison to grafting

		costs per pagefault	
L4	*Pentium 133 MHz*	4.5 μs	592 cycles
Spin	*Alpha 21064 133 MHz*	29.0 μs	3,857 cycles

Table 5: Simple Pager, Spin Versus L4. Experiment: A user program accesses an unmapped page. The page fault is sent to a user-level pager which simply maps an existing page (no paging). Costs include all hardware, kernel and user-level operations required to resolve the page fault.

	overhead per graft invocation	
L4 *Pentium 133 MHz* 2× IPC, including address-space switch	2...3 μs	240...400 cycles
Vino *Pentium 120 MHz*		
Read-ahead Graft	102 μs	12,240 cycles
Page-Eviction Graft	156 μs	18,720 cycles
Scheduling Graft	113 μs	13,560 cycles
Encryption Graft	251 μs	30,120 cycles

Table 6: Vino Grafts Versus L4 IPC. Vino-graft overhead includes sandboxing, transactions and locking. Result-checking and graft-functionality costs are not included.

4 Conclusion

We have some substantiated ideas about the architectural costs of IPC, i.e. the ideally achievable performance. In practice, no more than 100 to 200 cycles and 2.3% to 3.3% of the L1 cache are required. IPC is an order of magnitude faster than the reported costs of grafting kernels or servers. To decide whether grafting is a relevant technique, we need similar optimization efforts and analysis for the grafting approach.

References

Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., and Chambers, C. 1995. Extensibility, safety and performance in the Spin operating system. In *15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, pp. 267–284.

Ford, B., Hibler, M., Lepreau, J., Tullman, P., Back, G., and Clawson, S. 1996. Microkernels meet recursive virtual machines. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, pp. 137–152.

Hohmuth, M., Wolter, J., Baumgartl, R., and Borriss, M. 1996. Porting Linux to L4. http: //os.inf.tu-dresden.de/ L4/LinuxOnL4/ LiOnL4.html.

Liedtke, J. 1995. On μ-kernel construction. In *15th ACM Symposium on Operating System Principles (SOSP)*, Copper Mountain Resort, CO, pp. 237–250.

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

Necula, G. C. and Lee, P. 1996. Safe kernel extensions without run-time checking. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, pp. 229–243.

Schönberg, S. 1996. The L4 microkernel on Alpha – design and implementation. Cambridge University Technical Report 407.

Seltzer, M. I., Endo, Y., Small, C., and Smith, K. A. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *2nd USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Seattle, WA, pp. 213–228.