

# Hybrid vs. Monolithic OS Kernels: A Benchmark Comparison

Dylan Griffiths and Dwight Makaroff  
University of Saskatchewan  
Saskatoon, Saskatchewan, S7N 5C9  
Canada

## Abstract

Operating Systems are huge, complex pieces of software that are difficult to design and maintain in both security and functionality. Microkernel-based operating system designs reduce implementation complexity and increase code modularity, but have had serious performance drawbacks. The increase in the speed of both processing and memory access has led some to reconsider the relative advantages and disadvantages in microkernel design [5].

In this paper, we investigate the performance of modern, widely-used workstation operating systems using a standard process-based benchmark. The experiments were executed on commodity hardware and the results indicate that hardware characteristics dominate the differences in the kernel architecture, lending support to the idea that microkernels may be poised to gain widespread deployment.

## 1 Introduction

As hardware has become more complex, OS kernels have also become more complex. Monolithic kernels have added such features as dynamic module loading, multithreading, and kernel locks to allow reentrant access to the data structures and code sequences inside. Another way of dealing with the complexity has been to structure the kernel of the operating system as a microkernel.

The overhead of coordinating the different parts of the microkernels limited the performance possible in early microkernels. Current

personal workstations have increased in power by orders of magnitude since initial microkernel research, particularly in memory size and bus speeds.

If the performance issues of early microkernels [1] were related to memory throughput, rather than the software design, it is possible that such a system could run on modern machines with minimal overhead.

Early monolithic kernels did not easily handle the large number of dynamic devices that could be added to a system, without requiring a restart. Microkernels [1, 4] break the operating system kernel up into discrete parts, isolated by memory protection barriers. The core microkernel handled only such basics as message passing, time slicing, and the initial bootstrapping of the hardware and software. Servers providing facilities such as network communication and filesystem access are connected via the main synchronizing microkernel.

Initially, message passing (IPC) overhead degraded performance. Subsequent research investigated hybridization to solve the performance problem of microkernels. IPC overhead was reduced by allowing strongly logically connected components to reside within the same memory space.

Related research into comparative performance of kernel designs (e.g. [3]) has not looked at commercial microkernel operating systems such as MacOS X and Windows. In this paper, we examine some performance characteristics of hybrid and monolithic kernels on commodity hardware using a standard benchmark.

---

Copyright © 2006 University of Saskatchewan. Permission to copy is hereby granted provided the original copyright notice is reproduced in copies made.

## 2 Methodology

We chose *lmbench3*, which has been widely used [2], as our benchmark program. Four tests were selected from *lmbench*'s suite to characterize some of the performance details of the processor, memory, and OS of the test systems. The first test was that of memory latency. Memory latency provides a baseline measurement for the hardware. The other three tests were related to combinations of the software and hardware: context switch overhead, memory read bandwidth, and file read bandwidth.

Context switch overhead is measured by passing a token between a number of concurrent processes via IPC, capturing IPC implementation efficiency. Memory read bandwidth is tested by allocating a block of memory, zeroing it, and then copying the contents of the first-half to the second-half. The file read bandwidth benchmark is very similar in concept: open a file and read it in 64 KB chunks.

Each operating system was installed and minimally configured to support compilation and execution of the *lmbench3* benchmark. The benchmark was executed 10 times on each combination of software and hardware.

## 3 Test Environment

Table 1 lists the operating systems tested and corresponding hardware for each test. Two systems have monolithic kernels (Linux, OpenBSD), while two are microkernel based with some hybrid features in each (MacOS X, Windows).

CPU/mode	Operating System
PowerPC	Linux (2.6.12.X), MacOS X (10.4.5), OpenBSD 3.8
Sempron	Linux, OpenBSD, Windows 2K (SP4)
Athlon64 32-bit	Linux, MacOS X, OpenBSD, Windows
Athlon64 64-bit	Linux, OpenBSD

Table 1: Test Configurations

As shown in Table 1, three processors were

tested. Two of these processors have a CISC interface, while the third is a RISC machine. Table 2 gives more detail regarding the hardware components. The bus speed referred to is the memory bus speed.

Feature	AMD64	Sempron	PPC G4
CPU	2.0 GHz	1.75 GHz	1.5 GHz
Memory	1024 MB	768 MB	512 MB
Bus Speed	800 MHz	266 MHz	333 MHz
L1 Cache	64 KB	64 KB	32 KB
L2 Cache	512 KB	256 KB	512 KB

Table 2: Hardware Characteristics

## 4 Results

Preliminary analysis of the result graphs identified interesting performance behaviour. Detailed results for Windows are not shown as *lmbench* interacted with the *Cygwin* environment in unpredictable ways.

Fig. 1 shows the memory latency for the A64 in 32-bit mode for Linux. It is typical of all the graphs on this architecture. The stride size is the offset into a fixed array for each access. The small memory size operations were very fast and did not differ between operating systems or hardware platforms. This corresponds to data being held in registers and/or L1 cache. Once the memory accesses reached the boundary of the L2 cache, latencies went up by a fixed amount representing the time required to service the request by issuing a read command, the delay of waiting for the data to become ready, and the bus transit time.

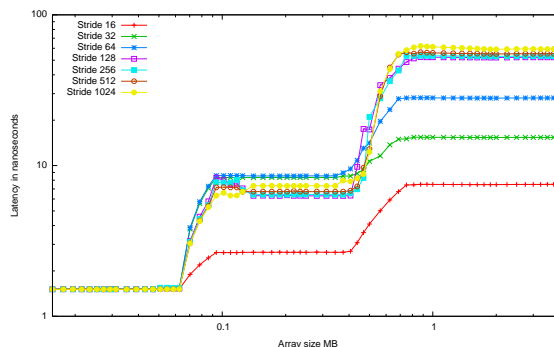


Figure 1: Memory Latency (A64-32 bit Linux)

Some stride sizes have a higher initial latency as the cache refills with data, settling to a lower overall latency. This pattern was not found on any other machine, and improves latency of access patterns inside of L2 cache. Once the working set size exceeded the L2 cache size, latencies again increase. Differences can be observed between all stride lines on all architectures, with smaller strides generally gaining the most benefit from prefetching and caching.

Context switching latency is very important in a system, given its frequent occurrence. The data transferred consists of the registers and some state information. This is stored in memory while another context is restored from memory onto the processor. It appeared that there was a direct relationship between the speed of the memory subsystem and the context switch latency. However, there was one case where there was not a linear relationship between bus speed and context switch latency: OpenBSD's A64-64bit port.

The OpenBSD context switch latency does not level off as it does under Linux, as shown in Fig. 2. It appears to increase in terms of both the number of processes active, and the size of the token sent via IPC. Each line illustrates this well; each line has a steep slope early, and a shallow, but non-zero slope later on.

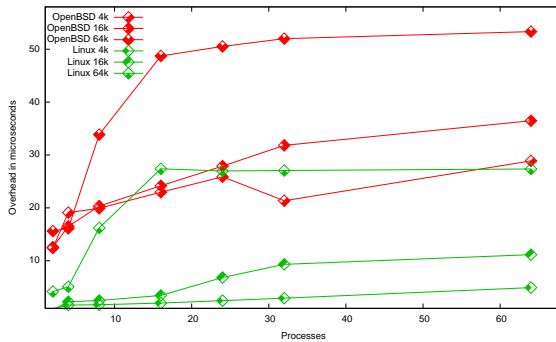


Figure 2: Context Switch Latency (A64)

Context switch latency under MacOS X is similar in performance to Linux. In Fig. 3, MacOS X appears to have 4 ns of additional latency. This is possibly due to the microkernel context switch overhead. This also holds for the Power PC, where MacOS X's latency shows

roughly a 10 ns increase.

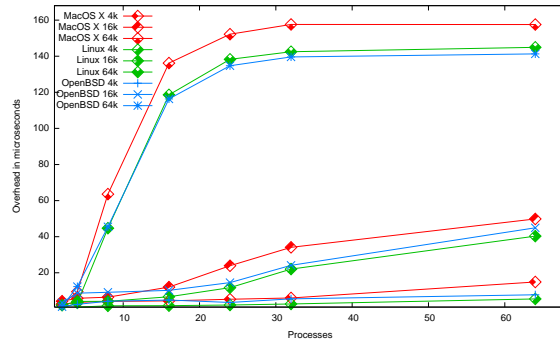


Figure 3: Context Switch Latency (PPC)

Table 3 summarizes the memory bus speed. Memory performance for the PPC and Sempron are affected by more than DRAM memory latency. The PPC context switch latency is worse than the Sempron's because its faster RAM does not compensate for the additional general purpose registers (32 vs. 8).

System	Latency
333MHz MPX PowerPC	140 - 160 ns
266MHz EV6 Sempron	100 - 120 ns
800MHz HT 64-bit A64	30 ns
800MHz HT 32-bit A64	25 - 30 ns

Table 3: Bus and context switch latency

A subset of the memory throughput performance for the A64 in 32-bit mode is shown in Fig. 4. Memory accesses less than L1 cache size have constant speed. As working set sizes increase, the hardware becomes more of a limiting factor. Moving from L1 to L2 reduces the achieved bandwidth substantially.

The filesystem performance builds directly on the memory performance. If the kernel is able to align its accesses and be smart about page management for the disk-cache, a significant performance improvement can be gained. The effect of the L1 and L2 caches on the graphs (Fig. 5) are noticeable. Within the L2 cache size, algorithms which invalidate fewer cache lines have a higher throughput. Once the L2 cache boundary is exceeded, most systems experience nearly identical throughput -

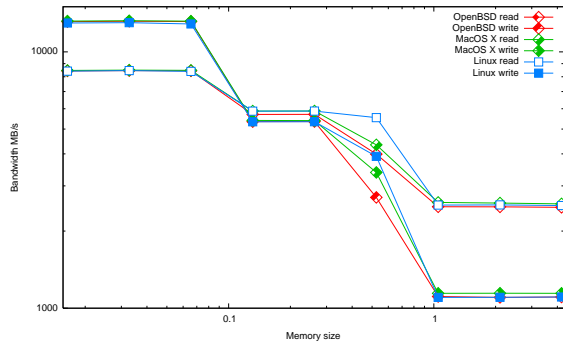


Figure 4: Memory Read (A64-32 bit)

being primarily limited by the hardware, not their cache management algorithms.

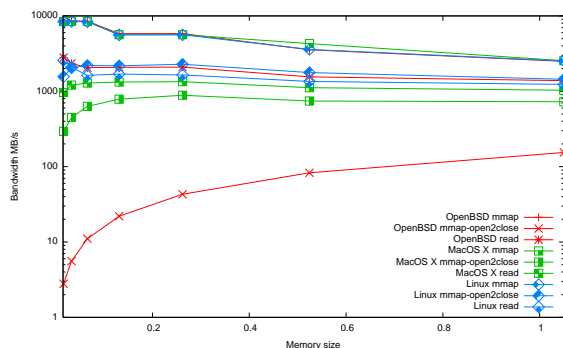


Figure 5: File Read (A64-32 bit)

Within the L1 cache boundary, Linux outperformed MacOS X by an order of magnitude on some operations, while within the boundaries of the L2 cache they perform within a few percentage points of each other. Once the working set exceeded the size of the L2 cache, Linux and MacOS X performed similarly.

OpenBSD was the only operating system to exhibit different behaviour. We see that the file *mmap open2close* line starts off much lower than any other operating system, and does not appear to be affected by caching.

## 5 Conclusions

Based on our limited set of experiments, differences between microkernels and monolithic

kernels are still present, but hardware seems to be the dominant factor. Linux appears to have a performance advantage on some operations, but this advantage is marginal once the working set exceeds the L2 cache size. MacOS X's results indicate reasonable performance with a hybrid microkernel design.

Modern workstations execute applications that are extremely network intensive. An efficient network protocol stack is essential to make meaningful comparisons between the two kernel architectures. Examination of these results in further detail, as well as how to tune the operating systems for better performance is part of continuing work.

## References

- [1] M. Accetta, R. Baron, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. In *Summer 1986 USENIX Technical Conference*, June 1986.
- [2] A. Brown and M. Seltzer. Operating system benchmarking in the wake of lmbench: a case study of the performance of netbsd on the intel x86 architecture. In *ACM SIGMETRICS*, pages 214–224, Seattle, WA, USA, 1997.
- [3] H. Hartig, M. Hohmuth, J. Liedtke, S. Schonberg, and J. Wolter. The Performance of uKernel-Based Systems. *Operating Systems Review*, 31(5):66–77, 1997.
- [4] J. Liedtke. On  $\mu$ -Kernel Construction. In *SOSP*, pages 237–250, 1995.
- [5] A. Tanenbaum, J. Herder, and H. Bos. Can We Make Operating Systems Reliable and Secure? *IEEE Computer*, 39(5):44–51, May 2006.