Architecture of Scalable Operating Systems: Multikernel

A Paper for KvBK-Seminar
at Friedrich-Alexander-Universität Erlangen-Nürnberg

Rasmus Pfeiffer

ABSTRACT

This paper gives a brief overview of the multikernel operating system model and uses Barrelfish [5] as example. The multikernel model tries to provide a solution to the problem of ever faster changing hardware and with it its scalability issues. The multikernel model addresses the scalability issue by using message passing instead of shared memory.

1. INTRODUCTION

Computer hardware is changing faster than ever. To keep up with this trend, software needs to be more adaptable. CPUs are getting more and more cores. This introduces scalability issues like keeping the cache coherent. This is one reason why future CPUs probably wont have a cache-coherent system anymore. To solve this problem, the multikernel operating system model can be used. Which also helps solving the problem of hardware becoming more and more diverse. A multikernel OS can run on a heterogeneous CPU like in AMD's Skybridge project.

2. MOTIVATION

Current CPU architectures rely on shared memory for inter-core communication. This also means current operating systems use the concept of shared memory. There are already signs that this could change in the future. For example Intel's Xeon Phi processors do not use shared memory for inter-core communication any more. This is why a new OS model is needed.

2.1 Number of Cores Increases

In 1978 Lauer and Needham said that massage passing and shared memory systems were equal and it depends on the hardware architecture which one is faster [12]. A simple experiment shows that this is not true anymore. Today's model of shared memory does not scale well as figure 1 shows. The more cores a system has the more complex it gets to keep the shared memory consistent. Message passing on the other hand scales a lot better. The performance gain by using caches becomes a performance loss on multi-core or many-core systems. It takes more time to keep to cache coherent, than you save by using caches. To prevent this problem, the multikernel OS architecture uses message passing on every level possible.

2.2 Diverse Hardware

Hardware is also becoming more and more diverse. AMD's Skybridge project for example uses x86 and arm in one CPU. Operating systems need quite a few changes to run on such hardware. The multikernel model also acomodates such hardware and can easily adapted to new and yet unknown hardware.

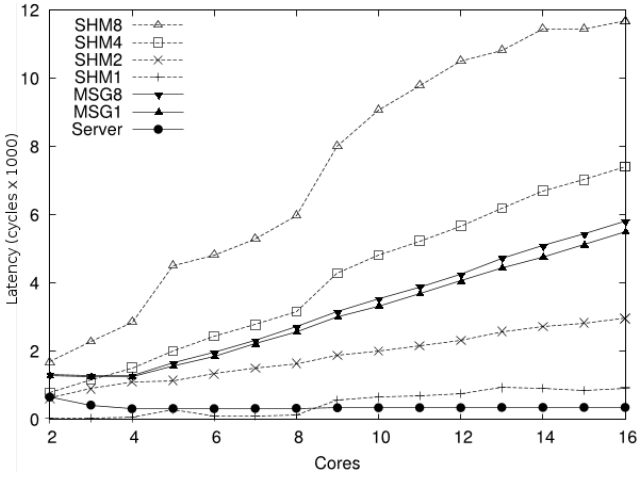

Figure 1: Comparison of the cost of updating shared state using shared memory and message passing [5].

3. SHARED MEMORY VS MESSAGE PASSING

A long time shared memory and message passing were seen as duals [12]. Although today, with the increasing number of CPU cores it becomes increasingly clear, that shared memory introduces problems, which were not considered to be that harmful to performance as they actually are.

ACM ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

3.1 Performance Comparison

Figure 1 plots the latency against number of cores for updates of various sizes on a 4x4-core AMD system. The shared memory cases (SHM1-8) show the latency per operation (in cycles) for updates, that directly modify 1, 2, 4, and 8 shared cache lines. The cost grows approximately linear. A single core can update the cache lines quite fast, but 16 cores take a lot longer, because most of the time the CPU is stalled, because of cache misses.
In the message passing case client threads (MSG1 and MSG8) call a lightweight remote procedure call [6] (which fits in single cache line). The cost varies little with the number of modified cache lines, because they remain in the server's local cache. The elapsed time per operation in the server thread grows linearly with the number of client threads.
The server curve shows time spend performing each update operation. This excludes queuing delay. This cost is mostly independent of the number of threads.
Overall the RPC latency is lower than shared memory access. Furthermore, with an asynchronous or pipelined RPC implementation the client processors can avoid stalling on cache misses. This figure also shows scalability issues with even a small number of cores. The performance issue of shared memory arises less from the necessity of using locks, but from data locality issues [4].
The number of cores in CPUs will grow and with that cache-coherence protocols will become increasingly expensive. There are already CPU architectures in development which use message passing at hardware level or a combination of both [13].

3.2 Concerns

There are legitimate concerns with a shared nothing model as the multikernel model. There are two main concerns. The first is, not being able to access shared data. The second is, the event-driven programming style resulting from asynchronous messaging.
Shared data makes developing applications easier. Although, by using shared data, you run in performance and scalability issues. Experts are already very careful about details such as lock granularity. By fine-tuning code at low level one can reduce the performance and scalability issue. Although current operating systems already use a lot of low level optimizations, it is getting more and complex.
The second concern does not apply to modern operating systems, as they already use an event driven model. Event-driven models are also already used in other programming domains like graphical user interface and network services.

4. INTER-PROCESS COMMUNICATION

Efficient inter-process communication is central for the design of operating systems [7, 10]. One way of efficient inter-process communication is the User-Level Remote Procedure Call (URPC) [6].

4.1 User-Level vs Kernel-Based

Kernel-based inter-process communication (IPC) is limited by the cost of invoking the kernel and reallocating a processor from one address space to another. On a multi-core CPU messages can be passed directly between address spaces, by running the different address space on different cores, which can be done in URPC.
In URPC messages are sent directly between address spaces, without invoking the kernel. This avoids unnecessary processor reallocation. This preserves cache and TLB context across calls. When processor reallocation is needed, its overhead is reduced, because it can be used for several independent calls.

4.2 Assumptions

URPC makes two optimistic assumptions. First, the client has other work to do. Second, the server has, or will have, a CPU core with which it can service a message. The first assumption makes it possible to do an inexpensive context switch between user-level threads during the blocking phase of a cross-address call. The second assumption enables a URPC to execute in parallel. The combination of both assumptions minimizes the cost of processor reallocation, by using one reallocation for multiple URPCs.

5. MULTIKERNEL MODEL

In the multikernel model the OS is structured as distributed system. Each CPU core runs its own kernel. The cores communicate using message passing and have no shared memory. The three main principles of the multikernel model are:

1. Make all inter-core communication explicit.
2. Make OS structure Hardware-neutral.
3. View state as replicated instead of shared.

5.1 Make all inter-core communication explicit

A multikernel OS uses message passing for inter-core communication. That is why no memory is shared between the CPU cores, except if it used for message passing. This model resembles a network. Because of that we can use well known network optimizations techniques, like pipelining and batching.

5.2 Make OS structure Hardware-neutral

Because each CPU runs its own kernel, each kernel can be different. This allows us to only make a small part of the system hardware dependent. This small part is easily exchanged for a different hardware dependent code. This is the reason, that there is very little work to be done, to port a multikernel OS to new hardware. Which certainly helps with the ever faster changing world of available hardware than ever.

5.3 View state as replicated instead of shared

Traditional operating systems use shared data structures to keep things that need to be accessible to multiple CPU cores. This data structure is usually protected by locks. However, in a multikernel OS with no shared memory the global state of the system needs to be replicated across all CPU cores. By replicating data across multiple cores the scalability is improved, because it reduces the overhead of synchronization. If we apply techniques from distributed systems, we can even power off parts of the system while keeping a consistent global state. Which can reduce power consumption.

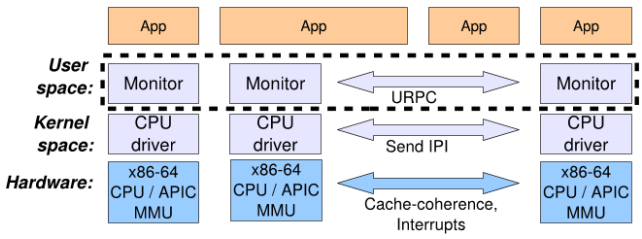

Figure 2: Barrelfish structure [5]

6. BARRELFISH

Barrelfish is one implementation which follows the multikernel model. It implements the multikernel model by running a CPU driver in privileged-mode and a monitor in user mode on each CPU core as shown in figure 2.

6.1 CPU driver

The CPU driver is single threaded and controls its locally connected hardware like APIC, MMU, etc. It shares no state with other cores, this is why it can be completely event driven. Which also means it is easier to write and also has less lines of code. The CPU driver is heavily specialized for the CPU architecture it is running on. It implements a lightweight, asynchronous same-core inter-process communication, which delivers a message to a process and unlocks it.

6.2 Monitors

The monitor is processor-agnostic and manages the system-wide state. It does most of the work a traditional kernel does. It keeps the global state of the system replicated across all CPU-cores means of an agreement protocol.

6.3 Inter-core communication

The multikernel model dictates that all inter-core communication is explicit. Which means that all communication between cores is via messages. To problem with current hardware is that the only mechanism for inter-core communication is cache-coherent memory. This is why Barrelfish uses a variant of user-level RPC (URPC) [6].
In Barrelfish URPC is implemented by letting the sender write a message sequentially into a cache line. The receiver polls on the last word of the line. This way it is ensured, that if the receiver reads the cache line while the sender is writing it, that the receiver never reads a partial message. This technique needs two round trips across the interconnect: one when the sender starts to invalidate the line in the receiver's cache, and one when the receiver fetches the line from the sender's cache.
Receiving messages is performed by polling for cartain time and then asking the monitor the notify the dispatcher if a message is available. Because the whole system depends on the performance of the message passing protocol the developers of Barrelfish compared different protocols for TLB shootdown, which is a worst case scenario. Figure 3 shows their result on a 8x4-core AMD system. They compared the following four protocols.
Broadcast protocol: the master monitor sends a URPC message to all cores and then waits for the acknowledgement by all other cores. This does not scale well due to the cache-coherence protocol used by AMD64 [8].

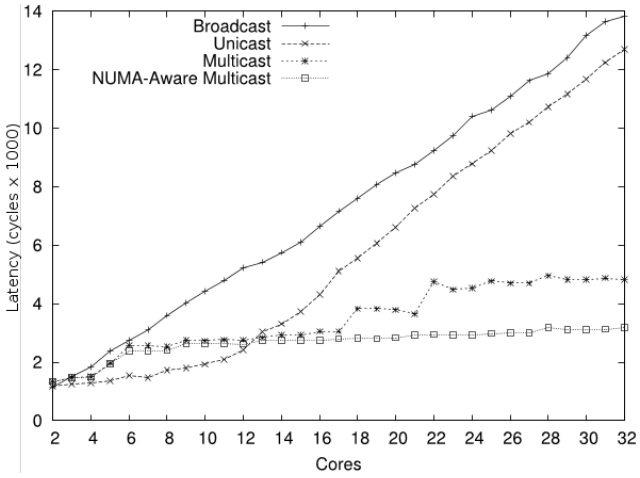

Figure 3: Comparison of TLB shootdown protocols [5]

Unicast protocol: the master sends individual messages to each core. This performs better, especially on a lower number of cores and is even faster CPUs with 4-12 cores, but still scales linear.
Multicast protocol: the master sends a message to the first core of each processor, which is then forwarded to the other cores of the processor. This scales in comparison to broadcast or unicast.
The NUMA-aware multicast protocol uses URPC buffers from memory local to the multicast aggregation nodes. The master sends the first message to the node with the highest latency. This protocol scales very well. It only shows steps, when the number of levels in the multicast tree increases.

6.4 Process structure

In Barrelfish a process consists of dispatcher objects, one on each core it might execute on. Communication on Barrelfish is between dispatchers. The dispatchers are scheduled by the local CPU driver. The CPU driver invokes an upcall interface, which is provided by each dispatcher. The threads package of Barrelfish provides user level API similar to POSIX threads. It provides support for implementing a traditional application sharing a single process address space across multiple cores.

6.5 Memory management

Global resources, such as physical memory, must be managed consistently. In the case of physical memory, the system has to ensure, that no two processes can acquire the same region of memory. Barrelfish uses a capability system, modeled after seL4 [11]. In this model, all memory management is performed explicitly through system calls that manipulate capabilities, which are user-level references to kernel objects or regions of physical memory. It also removes all memory allocation from the CPU driver, so that it only needs to check the correctness of operations that manipulate memory regions.
In Barrelfish all virtual memory management is performed by user-level code. For allocating memory, a user process first must acquire enough physical memory to store the required page tables. It then inserts the page table into the root page table. The CPU driver only checks for the correct-

ness of those operations. After that the process can allocate more memory and inserts it in its page table.
The monitors coordinate the global consistency of the page tables. This is implemented by one-phase commit operation between all monitors [5]. One exception is inserting the page table into the root page table. This needs to be done in a two-phase commit.

6.6 Performance

6.7 Compute-bound workloads

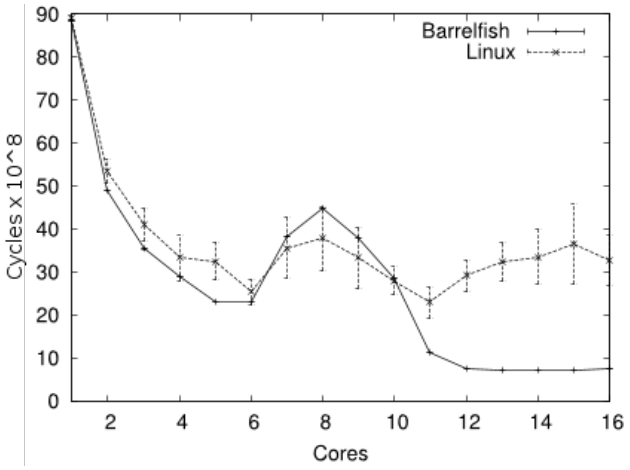

Figure 4: OpenMP conjugate gradient [5]

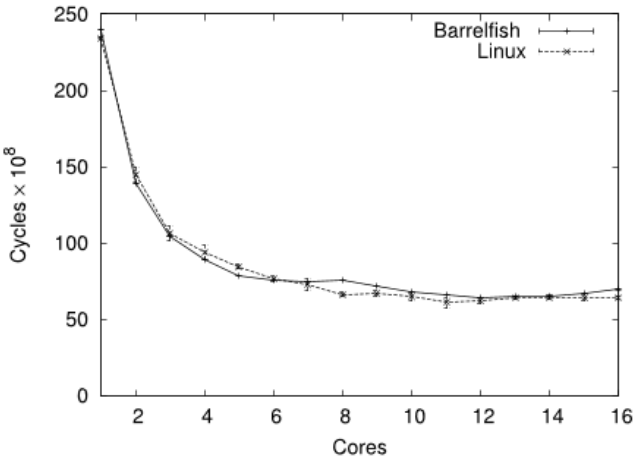

Figure 5: OpenMP 3D fast Fourier transformation [5]

Figures 4-8 show a comparison between Barrelfish and Linux using the NAS OpenMP [9] benchmark suite and SPLASH-2 [3] parallel application test suite. The difference between Linux and Barrelfish in figures 4 and 6 are explained by the different implementation of thread libraries. Linux use a in-kernel implementation and Barrelfish uses a user-space implementation.
Although those result show a similar performance between Linux and Barrelfish further testing with real world applications is need to determine if Barrelfish has comparable

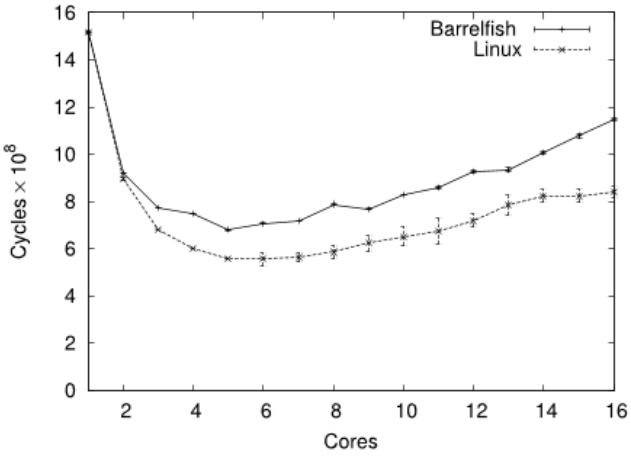

Figure 6: OpenMP integer sort [5]

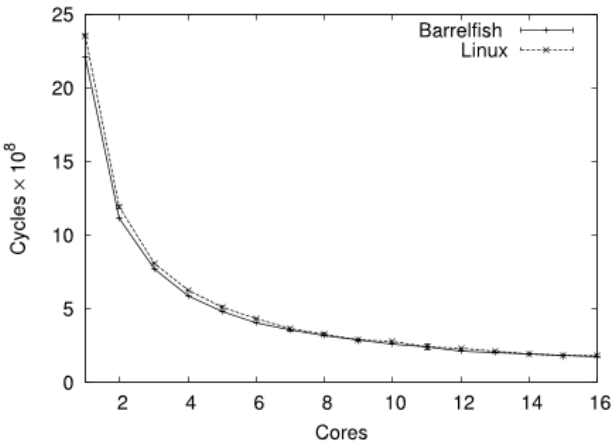

Figure 7: SPLASH-2 Barnes-Hut [5]

performance to Linux or other multipurpose operating systems. Those tests only indicate that the development of Barrelfish is on the right way.

6.8 IO workloads

The performance of an OS is not only determant by its computing power, but also by how fast it can handle IO. In today's systems the most stressful IO is network IO.
The Barrelfish developers first measured the UDP throughput by comparing Barrelfish to Linux. They used an application, that echos the recieved UDP packet back to the sender. They then measured the throughput by using the ipbench daemon [14] on e1000 NICs. Barrelfish reached a throughput of 951.7 Mbit/s and Linux 2.6.26 951 Mbits/s. Which is almost the maximum the NIC can reach.
Secondly they used a web server serving dynamic and static content. On Barrelfish they put the webserver on core 3, the e1000 driver on core 2, and the other system services on core 0. On Linux they did run lighttpd [1]. Barrelfish was able to answer 18697 requests per second with static content and Linux 8924 requests per second. The performance difference is mainly due to Barrelfish avoiding kernel-user crossing, by running entirely in user space. For the dynamic content they

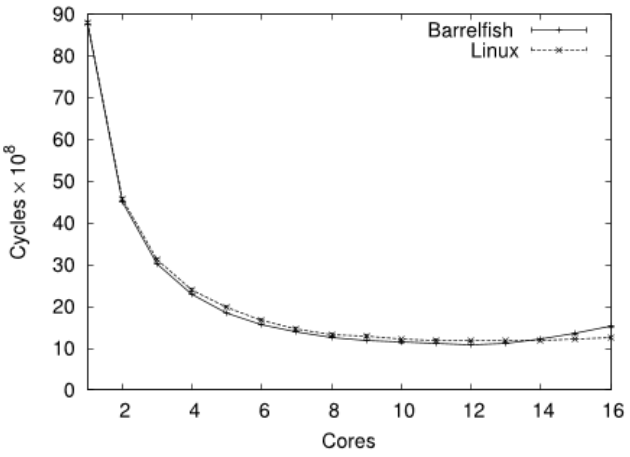

Figure 8: SPLASH-2 radiosity [5]

did run SQLite [2] on core 1 and reached 3417 requests per second. The bottleneck in this case was the SQLite server.

7. CONCLUSION

The multikernel model is an interesting idea, to solve the scalability problems of current operating systems. It already has an implementation with Barrelfish, which shows that it is possible, that the multikernel model can perform reasonably well in comparison to current operating systems.
Barrelfish is not the only implementation of the multikernel model. The Invasive Computing project [13] for example goes a step further and also designs its own hardware.
As good as those concepts may be, they are still concepts at the moment and we will have to see, if they are accepted by the computer industry. It will take quite a few years until anyone can predict, if such a fundamental change for operating systems will be accepted.

8. ACKNOWLEDGEMENT

Most of this paper was derived from The multikernel: a new os architecture for scalable multicore systems by A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. [5].

9. REFERENCES

[1] lighttpd webserver. http://www.lighttpd.net.

[2] Sqlite database server. http://www.sqlite.org.

[3] Stanford parallel applications for shared memory (splash-2). http://www-flash.stanford.edu/apps/SPLASH/.

[4] J. Appavoo, D. D. Silva, O. Krieger, M. Auslander, M. Ostrowski, B. Rosenburg, A. Waterland, R. W. Wisniewski, J. Xenidis, M. Stumm, et al. Experience distributing objects in an smmp os. *ACM Transactions on Computer Systems (TOCS)*, 25(3):6, 2007.

[5] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: a new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 29–44. ACM, 2009.

[6] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems (TOCS)*, 9(2):175–198, 1991.

[7] D. Cheriton. The v distributed system. *Communications of the ACM*, 31(3):314–333, 1988.

[8] A. M. Devices. Amd64 architecture programmer's manual volume 2: System programming, 2006.

[9] H.-Q. Jin, M. Frumkin, and J. Yan. The openmp implementation of nas parallel benchmarks and its performance. 1999.

[10] M. B. Jones and R. F. Rashid. *Mach and Matchmaker: Kernel and language support for object-oriented distributed systems*, volume 21. ACM, 1986.

[11] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[12] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *ACM SIGOPS Operating Systems Review*, 13(2):3–19, 1979.

[13] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting. Invasive computing: An overview. In *Multiprocessor System-on-Chip*, pages 241–268. Springer, 2011.

[14] I. Wienand and L. Macpherson. ipbench–a framework for distributed network benchmarking. *AUUGN*, page 163, 2004.