The Three Questions

- What is the problem?
- What is new or different?
- What are the contributions and limitations?
ccNUMA Is Reality

Even in PCs...

PCIe

0 2 4 6

1 3 5 7

HyperTransport links

CPU L1
L2
CPU L1
L2
CPU L1
L2
CPU L1
L2

L3

RAM RAM
Memory Latency Varies...

Figure 2: The AMD 16-core system topology. Memory access latency is in cycles and listed before the backslash. Memory bandwidth is in bytes per cycle and listed after the backslash. The measurements reflect the latency and bandwidth achieved by a core issuing load instructions. The measurements for accessing the L1 or L2 caches of a different core on the same chip are the same. The measurements for accessing any cache on a different chip are the same. Each cache line is 64 bytes, L1 caches are 64 Kbytes 8-way set associative, L2 caches are 512 Kbytes 16-way set associative, and L3 caches are 2 Mbytes 32-way set associative.

These properties can already be observed in current multicore machines. Figure 2 summarizes the memory system of a 16-core machine from AMD. The machine has four quad-core Opteron processors connected by a square interconnect. The interconnect carries data between cores and memory, as well as cache coherence broadcasts to locate and invalidate cache lines, and point-to-point cache coherence transfers of individual cache lines. Each core has a private L1 and L2 cache. Four cores on the same chip share an L3 cache. Cores on one chip are connected by an internal crossbar switch and can access each others' L1 and L2 caches efficiently. Memory is partitioned in four banks, each connected to one of the four chips. Each core has a clock frequency of 2.0 GHz.

Figure 2 also shows the cost of loading a cache line from a local core, a nearby core, and a distant core. We measured these numbers using many of the techniques documented by Yotov et al. [31]. The techniques avoid interference from the operating system and hardware features such as cache line prefetching.

Reading from the local L3 cache on an AMD chip is faster than reading from the cache of a different core on the same chip. Inter-chip reads are slower, particularly when they go through two interconnect hops.

Figure 3 shows the performance scalability of locking, an important primitive often dominated by cache miss costs. The graph compares Linux's kernel spin locks and scalable MCS locks [21] (on Corey) on the AMD machine, varying the number of cores contending for the lock. For both the kernel spin lock and the MCS lock we measured the time required to acquire and release a single lock. When only one core uses a lock, both types of lock are fast because the lock is always in the core's cache; spin locks are faster than MCS locks because the former requires three instructions to acquire and release when not contended, while the latter requires 15. The time for each successful acquire increases for the spin lock with additional cores because each new core adds a few more cache coherence broadcasts per acquire, in order for the new core to observe the effects of other cores' acquires and releases. MCS locks avoid this cost by having each core spin on a separate location.

We measured the average times required by Linux 2.6.25 to flush the TLBs of all cores after a change to a shared page table. These "TLB shootdowns" are considerably slower on 16 cores than on 2 (18742 cycles versus 5092 cycles). TLB shootdown typically dominates the cost of removing a mapping from an address space that is shared among multiple cores.

On AMD 16-core systems, the speed difference between a fast memory access and a slow memory access is a factor of 100, and that factor is likely to grow with the number of cores. Our measurements for the Intel Xeon show a similar speed difference. For performance, kernels must mainly access data in the local core's cache. A contended or falsely shared cache line decreases performance because many accesses are to remote caches. Widely-shared and contended locks also decrease performance, even if the critical sections they protect are only a few instructions. These performance effects will be more damaging as the number of cores increases, since the cost of accessing far-away caches increases with the number of cores.
Figure 2: The AMD 16-core system topology. Memory access latency is in cycles and listed before the backslash. Memory bandwidth is in bytes per cycle and listed after the backslash. The measurements reflect the latency and bandwidth achieved by a core issuing load instructions. The measurements for accessing the L1 or L2 caches of a different core on the same chip are the same. The measurements for accessing any cache on a different chip are the same. Each cache line is 64 bytes, L1 caches are 64 Kbytes 8-way set associative, L2 caches are 512 Kbytes 16-way set associative, and L3 caches are 2 Mbytes 32-way set associative.

These properties can already be observed in current multicore machines. Figure 2 summarizes the memory system of a 16-core machine from AMD. The machine has four quad-core Opteron processors connected by a square interconnect. The interconnect carries data between cores and memory, as well as cache coherence broadcasts to locate and invalidate cache lines, and point-to-point cache coherence transfers of individual cache lines. Each core has a private L1 and L2 cache. Four cores on the same chip share an L3 cache. Cores on one chip are connected by an internal crossbar switch and can access each others' L1 and L2 caches efficiently. Memory is partitioned in four banks, each connected to one of the four chips. Each core has a clock frequency of 2.0 GHz.

Figure 2 also shows the cost of loading a cache line from a local core, a nearby core, and a distant core. We measured these numbers using many of the techniques documented by Yotov et al. [31]. The techniques avoid interference from the operating system and hardware features such as cache line prefetching.

Reading from the local L3 cache on an AMD chip is faster than reading from the cache of a different core on the same chip. Inter-chip reads are slower, particularly when they go through two interconnect hops.

Figure 3 shows the performance scalability of locking, an important primitive often dominated by cache miss costs. The graph compares Linux's kernel spin locks and scalable MCS locks [21] (on Corei) on the AMD machine, varying the number of cores contending for the lock. For both the kernel spin lock and the MCS lock we measured the time required to acquire and release a single lock. When only one core uses a lock, both types of lock are fast because the lock is always in the core's cache; spin locks are faster than MCS locks because the former requires three instructions to acquire and release when not contended, while the latter requires 15. The time for each successful acquire increases for the spin lock with additional cores because each new core adds a few more cache coherence broadcasts per acquire, in order for the new core to observe the effects of other cores' acquires and releases. MCS locks avoid this cost by having each core spin on a separate location.

We measured the average times required by Linux 2.6.25 to flush the TLBs of all cores after a change to a shared page table. These "TLB shootdowns" are considerably slower on 16 cores than on 2 (18742 cycles versus 5092 cycles). TLB shootdown typically dominates the cost of removing a mapping from an address space that is shared among multiple cores.

On AMD 16-core systems, the speed difference between a fast memory access and a slow memory access is a factor of 100, and that factor is likely to grow with the number of cores. Our measurements for the Intel Xeon show a similar speed difference. For performance, kernels must mainly access data in the local core's cache. A contended or falsely shared cache line decreases performance because many accesses are to remote caches. Widely-shared and contended locks also decrease performance, even if the critical sections they protect are only a few instructions. These performance effects will be more damaging as the number of cores increases, since the cost of accessing far-away caches increases with the number of cores.
...Makes Sharing Look Bad

*Example: Thread looping over `dup` and `close`
What to Do?

- Avoid shared data structures altogether
- Use fine-grained locking or wait-free primitives

- Above approaches fix sharing policy in OS
  - Some instance should be shared, others shouldn’t

- New Principle: Applications control sharing of OS data
  - By default, only one core uses data
  - OS interface lets apps control sharing
We Need New Abstractions

- Address ranges
  - Control which parts of address space are private vs shared
- Kernel cores
  - Control which parts of kernel run on which core
- Shares
  - Control which kernel objects are private vs shared

- What does “private” mean here?
Address Ranges

- Contains a range of virtual-to-physical mappings
- May also refer to another range
- Shared ranges imply shared hardware page tables
- See mappings inserted by “soft page faults” — wtf?

(a) A single address space.

(b) Separate address spaces.

(c) Two address spaces with shared result mappings.
Kernel Cores

- Pin the execution of certain system calls to a specific core
- In practice, that is, Corey:
  - Create pcore
  - Add system call pseudo-device
  - Run pcore
  - Physical core then polls object for pending events
Map user-level identifiers to kernel-level pointers
  - Including other shares

Each of an application’s cores has a private root share

Two cores can share same identifiers by
  - Creating a new shared share
  - Adding that share to some share reachable from root
Some More Details on Corey

- Kernel has a few more abstractions
  - Segments for physical memory
  - Pcores for physical cores
  - Devices for physical devices and software services

- System services include
  - `cfork`: extend execution to a new core
  - Network stack, which can be private or shared
  - Buffer cache, which shared
    - Uses lock-free tree, private allocators, scalable read-write lock to prevent freeing/reuse
Corey Evaluation
Address Ranges

* memclone
  * Each core allocates 100 MB and modifies each page
  * Pages are allocated on fault, in memory connected to chip

* mempass
  * Allocate shared 100 MB
  * Touch each page on one core and then hand off to next core
Kernel Cores

- **TCP service**
  - Accept, write 128 B, close
- **Dedicated configuration**
  - One core handles network
- **Polling configuration**
  - One core polls for receive and transmit events only
- **Both configurations**
  - Private TCP stacks, services

---

**8.3 Kernel cores**

This section explores an example in which use of the Corey kernel core abstraction improves scalability. The benchmark application is a simple TCP service, which accepts incoming connections, writing 128 bytes to each connection before closing it. Up to 15 separate client machines (as many as there are active server cores) generate the connection requests.

We compare two server configurations. One of them (called "Dedicated") uses a kernel core to handle all network device processing: placing packet buffer pointers in device DMA descriptors, polling for received packets and transmit completions, triggering device transmission, and manipulating corresponding driver data structures. The second configuration, called "Polling", uses a kernel core only to poll for received packet notifications and transmit completions. In both cases, each other core runs a private TCP/IP stack and an instance of the TCP service. For Dedicated, each service core uses shared-memory IPC to send and receive packet buffers with the kernel core. For Polling, each service core transmits packets and registers receive packet buffers by directly manipulating the device DMA descriptors (with locking), and is notified of received packets via IPC from the Polling kernel core. The purpose of the comparison is to show the effect on performance of moving all device processing to the Dedicated kernel core, thereby eliminating contention over device driver data structures. Both configurations poll for received packets, since otherwise interrupt costs would dominate performance.

Figure 8(a) presents the results of the TCP benchmark. The network device appears to be capable of handling only about 900,000 packets per second in total, which limits the throughput to about 110,000 connections per second in all configurations (each connection involves 4 input and 4 output packets). The dedicated configuration reaches 110,000 with only five cores, while Polling reaches 110,000 with 15 cores.

---

**Figure 8**

(a) Throughput.

(b) L3 cache misses.
** Shares**

- Add & remove segment to/from share
- Version 1: global share
- Version 2: per-core share

![Graphs showing throughput and L3 cache misses.](attachment:image.png)

Figure 9: Share microbenchmark results.
Applications

MapReduce

![Corey and Linux performance.]

![Corey improvement over Linux.]

![Connections per second vs. File size (Kbytes, log scale).]

As mentioned in the Introduction, Gough

An Intel white paper reports use of spatial multiplexing, but provides a new kernel that runs on commodity operating systems, such as Linux, which offers a strong foundation for many applications. We compare Metis on Corey and on Linux, using the same benchmark suite across both systems. The results above should be viewed as a case for the principle that applications should control sharing rather than a conclusive “proof”. Corey lacks many features and others to the operating system. Corey also uses sparsely managed by the kernel. The K42 and Tornado implementations based on sharing patterns. The Corey kernel is designed to primarily manipulate the sources managed by the kernel. The K42 and Tornado implementations use tricks inspired by RCU to implement efficient functions to cores through kernel cores.

Much work has been done on making widely-used operating systems run well on multiprocessors. The K42 and Tornado implementations, like Corey, are designed to automatically assign to cores in a way that minimizes contention. Applications can schedule work directly manipulate the data using techniques like computation migration [6, 14].

Figure 10 presents the absolute performance of webd. As argument, and returns the sum of the bytes in that file.

An Intel white paper reports use of spatial multiplexing, but provides a new kernel that runs on commodity operating systems, such as Linux, which offers a strong foundation for many applications. We compare Metis on Corey and on Linux, using the same benchmark suite across both systems. The results above should be viewed as a case for the principle that applications should control sharing rather than a conclusive “proof”. Corey lacks many features and others to the operating system. Corey also uses sparsely managed by the kernel. The K42 and Tornado implementations based on sharing patterns. The Corey kernel is designed to primarily manipulate the sources managed by the kernel. The K42 and Tornado implementations use tricks inspired by RCU to implement efficient functions to cores through kernel cores.

Figure 10 presents the absolute performance of webd. As argument, and returns the sum of the bytes in that file.

As mentioned in the Introduction, Gough
What’s Missing From Corey Evaluation?
Is This Enough?
Not Really.
Enter the Multikernel
Motivation

- Systems are increasingly diverse
  - Different hardware requires different optimizations
- Cores are increasingly diverse
  - Large cores, small cores, different ISAs
- The interconnect matters
  - Message passing hardware replaces shared interconnect
Motivation (cont.)

- Message cost less than shared memory
- Cache coherence is not a panacea
  - As core # increases, cc becomes harder
- Messages are getting easier
  - Programmers already think in terms of cccc messages
  - Libraries and languages help
Three Principles

- Make inter-core communication explicit
- Make OS structure hardware neutral
- View state as replicated
Explicit Communication

- Enables reasoning about use of interconnect
  - What state, accessed when and by who
  - May be automated, formal reasoning
- Enables well-known optimizations
  - Notably, pipelining and batching
- Enables isolation and resource management
  - Even across heterogeneous or non-cc cores
- Enables split-phase, i.e., async, communication
Hardware Neutral Structure

- Limits scope of porting OS to new hardware
  - Optimizations only necessary in messaging implementation
  - But no changes necessary for distributed algorithm
- Enables late binding between protocol and messaging
  - Dynamically adjust to workloads
Replicated State

- Reduces load on interconnect, contention for memory, and overhead for synchronization
- Supports systems that do not share memory
- Supports dynamic changes to hardware
  - E.g., hot-plug or shut down processor
- Supports limited sharing
That’s Nice, But Not Very Concrete. Let’s Apply the Theory.
**Barrelfish Structure**

- **CPU driver**: similar to exokernel
  - Protection, timeslicing, hardware access, messaging
- **Monitor**
  - Much like a traditional OS
Process Structure

- Collection of dispatcher objects
  - One per core
- Scheduled by CPU driver
  - Upcall, aka scheduler activation
- Traditional user-level threads built on top
Inter-Core Communication

- Variant of user-level RPC (Remote Procedure Call)
  - Looks like a regular procedure call
  - Control passes through driver (kernel)
  - Data passes through pairwise shared memory
- Implementation relies on polling
  - Receiver polls for a bit before relinquishing core
- Implementation exploits interconnect
  - Write message sequentially into cache line
  - Poll on last word of the line
Memory Management

- Driver only protects memory through capabilities
- User-level code allocates & manages page tables etc.

- In hindsight: capabilities are too complex, simple memory manager would have sufficed
  - But uniform operations: mostly copy or retype capability

- Agree across monitors
  - Page (re)map: one-phase commit
  - Capability retyping: two-phase commit
Barrelfish still allows for single address space running many threads across cores

Page table may be shared or replicated
  - Replication reduces TLB invalidations

Capabilities can be shared across cores as well
  - Requires support by Barrelfish

Threads can also be shared (migrated) across cores
  - Needs to be implemented by thread scheduler
Problem: which is the right system mechanism?
  * Think how do I chose the right messaging implementation

Solution: system knowledge base
  * Hardware, incl. ACPI tables, PCI bus probes, CPUID data
  * Performance, including URPC latency
  * Facts, including interconnect topology, quirks
  * All expressed in subset of first order logic
This approach might just work
What Do You Think?