Milestone 7: another core

• Assignment:
  – Bring up the second Cortex-A9 core

• Today:
  – Multiprocessor operating systems
  – User-level RPC
  – Inter-core messaging in Barrelfish
Multiprocessor OSes

• Multiprocessor computers were anticipated by the research community long before they became mainstream
  – Typically restricted to “big iron”
• But relatively few OSes designed from the outset for multiprocessor hardware
• A multiprocessor OS:
  – Runs on a tightly-coupled (usually shared-memory) multiprocessor machine
  – Provides system-wide OS abstractions
Multics

• Time-sharing operating system for a multiprocessor mainframe
• Joint project between MIT, General Electric, and Bell Labs (until 1969)
• 1965 – mid 1980s
  – Last Multics system decommissioned in 2000
• Goals: reliability, dynamic reconfiguration, security, etc.
• Very influential
Multics: typical configuration

GE645 computer
Symmetric multiprocessor

Communication was by using “mailboxes” in the memory modules and corresponding interrupts (asynchronous).

to remote terminals, magnetic tape, disc, console reader punch etc
Multics on GE645

- Reliable interconnect
- No caches
- Single level of shared memory
  - Uniform memory access (UMA)
- Online reconfiguration of the hardware
  - Regularly partitioned into 2 separate systems for testing and development and then recombined
- Slow!
Hydra

• Early 1970s, CMU
• Multiprocessor operating system for C.mmp (Carnegie-Mellon Multi-Mini-Processor)
  – Up to 16 PDP-11 processors
  – Up to 32MB memory
• Design goals:
  – Effective utilization of hardware resources
  – Base for further research into OSes and runtimes for multiprocessor systems
**C.mmp multiprocessor**

Primary memory

- **Mp₀ (2MB)**
- **Mp₁₅ (2MB)**

Central processor (PDP-11)

- **Pc₀**
- **Mp**
- **Kc**

Control for clock, IPC

- **Clock**
- **Interrupt**

Address relocation hardware

Switch

Connections to secondary memory and devices
Hydra (cont)

• Limited hardware
  – No hardware messaging, send IPIs
  – No caches
    • 8k private memory on processors
  – No virtual memory support

• Crossbar switch to access memory banks
  – Uniform memory access (~1us if no contention)
  – But had to worry about contention

• Not scalable
Cm*

- Late 1970s, CMU
- Improved scalability over C.mmp
  - 50 processors, 3MB shared memory
  - Each processor is a DEC LSI-11 processor with bus, local memory and peripherals
  - Set of clusters (up to 14 processors per cluster) connected by a bus
  - Memory can be accessed locally, within the cluster and at another cluster (NUMA)
  - No cache
- 2 Oses developed: StarOS and Medusa
One Kmap per cluster

50 compute modules (CMs)
5 communication controllers (Kmaps)
Cm*

- NUMA
- Reliable message-passing
- No caches
- Contention and latency big issues when accessing remote memory
  - Sharing is expensive
  - Concurrent processes run better if independent
Medusa

- OS for Cm*, 1977-1980
- Goal: reflect the underlying distributed architecture of the hardware
- Single copy of the OS impractical
  - Huge difference in local vs non-local memory access times
    - 3.5us local vs 24us cross-cluster
- Complete replication of the OS impractical
  - Small local memories (64 or 128KB)
  - Typical OS size 40-60KB
Medusa (cont)

• Replicated kernel on each processor
  – Interrupts, context switching
• Other OS functions divided into disjoint utilities
  – Utility code always executed on local processor
  – Utility functions invoked (asynchronously) by sending messages on pipes
• Utilities:
  – Memory manager
  – File system
  – Task force manager
    • All processes are task forces, consisting of multiple activities that are co-scheduled across multiple processors
  – Exception reporter
  – Debugger/tracer
Medusa (cont)

• Had to be careful about deadlock, eg file open:
  – File manager must request storage for file control block from memory manager
  – If swapping between primary and secondary memory is required, then memory manager must request I/O transfer from file system
  → Deadlock

• Used coscheduling of activities in a task force to avoid livelock
Firefly

• Shared-memory, multiprocessor, personal workstation
  – Developed at DEC SRC, 1985-1987

• Requirements:
  – Research platform (powerful, multiprocessor)
  – Built in a short space of time (off-the-shelf components as much as possible)
  – Suitable for an office (not too large, loud or power-hungry)
  – Ease of programming (hardware cache coherence)
Firefly (version 2)

- Primary processor: MicroVAX 78032
- Secondary processors: CVAX 78034 (typically 4)

I/O controllers (disk, network, display, keyboard, mouse)
Firefly

- UMA
- Reliable interconnect
- Hardware support for cache coherence
- Bus contention an important issue
  - Analysis using trace-driven simulation and a simple queuing model found that adding processors improved performance up to about 9 processors
Topaz

- Software system for the Firefly
- Multiple threads of control in a shared address space
- Binary emulation of Ultrix system call interface
- Uniform RPC communication mechanism
  - Same machine and between machines
- System kernel called the Nub
  - Virtual memory
  - Scheduler
  - Device drivers
- Rest of the OS ran in user-mode
- All software multithreaded
  - Executed simultaneously on multiple processors
Memory consistency models

If one CPU modifies memory, when do others observe it?

• **Strict/Sequential**: reads return the most recently written value

• **Processor/PRAM**: writes from one CPU are seen in order, writes by different CPUs may be reordered

• **Weak**: separate rules for synchronizing accesses (e.g. locks)
  – Synchronising accesses sequentially consistent
  – Synchronising accesses act as a barrier:
    • previous writes completed
    • future read/writes blocked
Memory consistency models

If one CPU modifies memory, when do others observe it?

- **Strict/Sequential**: reads return the most recently written value
- **Processor/PRAM**: writes from one CPU are seen in order, writes by different CPUs may be reordered
- **Weak**: separate rules for synchronizing accesses (e.g. locks)
  - Synchronising accesses sequentially consistent
  - Synchronising accesses act as a barrier:
    - previous writes completed
    - future read/writes blocked

Important to know your hardware!
- x86: processor consistency
- PowerPC: weak consistency
Example: **MOESI** protocol

- Every cache line is in one of five states:
  - **Modified**: dirty, present only in this cache
  - **Owned**: dirty, present in this cache and possibly others
  - **Exclusive**: clean, present only in this cache
  - **Shared**: present in this cache and possibly others
  - **Invalid**: not present

- May satisfy read from any state
- Fetch to shared or exclusive state
- Write requires modified or exclusive state;
  - if shared, must invalidate other caches
- Owned: line may be transferred without flushing to memory
Hive

• Stanford, early 1990s
• Targeted at the Stanford FLASH multiprocessor
  – Large-scale ccNUMA
• Main goal was fault containment
  – Contain hardware and software failure to the smallest possible set of resources
• Second goal was scalability through limited sharing of kernel resources
Stanford FLASH architecture

- Processor
- Memory
- Coherence Controller
- 2nd-Level Cache
- Net
- I/O
Stanford FLASH

- Reliable message-passing
  - Nodes can fail independently
- Designed to scale to 1000’s of nodes
- Non-Uniform Memory Access
  - Latency increases with distance
- Hardware cache coherence
  - Directory-based protocol
  - Data structures occupy 7-9% of main memory
Hive (cont)

• Each “cell” (ie kernel) independently manages a small group of processors, plus memory and I/O devices
  – Controls a portion of the global address space
• Cells communicate mostly by RPC
  – But for performance can read and write each other’s memory directly
• Resource management by program called Wax running in user-space
  – Global allocation policies for memory and processors
  – Threads on different cells synchronize via shared memory
Hive: failure detection and fault containment

• Failure detection mechanisms
  – RPC timeouts
  – Keep-alive increments on shared memory locations
  – Consistency checks on reading remote cell data structures
  – Hardware errors, e.g., bus errors

• Fault containment
  – Hardware firewall (an ACL per page of memory) prevents wild writes
  – Preemptive discard of all pages belonging to a failed process
  – Aggressive failure detection
    • Distributed agreement algorithm confirms cell has failed and reboot it
Disco
Running commodity OSes on scalable multiprocessors [Bugnion et al., 1997]

• Context: ca. 1995, large ccNUMA multiprocessors appearing
• Problem: scaling OSes to run efficiently on these was hard
  – Extensive modification of OS required
  – Complexity of OS makes this expensive
  – Availability of software and OSes trailing hardware
• Idea: implement a scalable VMM, run multiple OS instances
• VMM has most of the features of a scalable OS, e.g.:
  – NUMA-aware allocator
  – Page replication, remapping, etc.
• VMM substantially simpler/cheaper to implement
• Run multiple (smaller) OS images, for different applications
Disco architecture

[Bugnion et al., 1997]
Disco Contributions

• First project to revive an old idea: virtualization
  – New way to work around shortcomings of commodity Oses
  – Much of the paper focuses on efficient VM implementation
  – Authors went on to found VMware

• Another interesting (but largely unexplored) idea: programming a single machine as a distributed system
  – Example: parallel make, two configurations:
    1. Run an 8-CPU IRIX instance
    2. Run 8 IRIX VMs on Disco, one with an NFS server
  – Speedup for case 2, despite VM and vNIC overheads
K42

• OS for cache-coherent NUMA systems
• IBM Research, 1997–2006ish
• Successor of Tornado and Hurricane systems (University of Toronto)
• Supports Linux API/ABI
• Aims: high locality, scalability
• Heavily object-oriented
  – Resources managed by set of object instances
Why use OO in an OS?

Traditional System

User-level requests

System paths & data structures used to satisfy requests

- much sharing

OO Decomposed System

- much less sharing
- better performance

[Appavoo, 2005]
Clustered Objects
Example: shared counter

- Object internally decomposed into processor-local representatives
- Same reference on any processor
  - Object system routes invocation to local representative
  ⇒ Choice of sharing and locking strategy local to each object
- In example, \textit{inc} and \textit{dec} are local; only \textit{val} needs to communicate
Clustered objects

Implementation using processor-local **object translation table**:  

![Diagram showing reference to object translation table, object instance, and virtual function table with code](diagram.png)
Challenges with clustered objects

• Degree of clustering (number of reps, partitioned vs replicated) depends on how the object is used

• State maintained by the object reps must be kept consistent

• Determining global state is hard
  – Eg How to choose the next highest priority thread for scheduling when priorities are distributed across many user-level scheduler objects
Concrete example: VM objects

- OO decomposition minimizes sharing for unrelated data structures
  - No global locks \(\Rightarrow\) reduced synchronization

- Clustered objects system limits sharing within an object
K42 Principles/Lessons

• Focus on **locality**, not concurrency, to achieve scalability
• Adopt distributed component model to enable consistent construction of locality-tuned components
• Support distribution within an OO encapsulation boundary:
  – eases complexity
  – permits controlled/manageable introduction of localized data structures
Clear trend....

- Finer-grained locking of shared memory
- Replication as an optimization of shared memory

These are research OSes or Supercomputers. So why would you care?
Further reading

- Multics: www.multicians.org
- “HYDRA: The kernel of a multiprocessor operating system”, W. Wulf et al, Comm. ACM, 17(6), June 1974
- “Policy/Mechanism Separation in Hydra”, R. Levin et al, 5th SOSP, Nov 1975
- “Medusa: An Experiment in Distributed Operating System Structure”, John K. Ousterhout et al, CACM, 23(2), Feb 1980
- “The duality of memory and communication in the implementation of a multiprocessor operating system”, Michael Young et al, 11th SOSP, Nov 1987 [Mach]
- “K42: Building a Complete Operating System”, 1st EuroSys, April, 2006
User-level RPC
User-level RPC (RPC)

• Arguably, URPC is to Scheduler Activations what LRPC was to kernel threads
  – Send messages between address spaces directly \textcolor{red}{\textit{no kernel involved!}}
  – Eliminate unnecessary processor reallocation
  – Amortize processor reallocation over several calls
  – Exploit inherent parallelism in send / receiving

• Decouple:
  – notification (user-space)
  – scheduling (kernel)
  – data transfer (also user space).
URPC operation

Application
  Stubs
  URPC
  Scheduler activations

Application
  Stubs
  URPC
  Scheduler activations

Channel

Reallocate processor

Kernel
How is the kernel not involved?

• Shared memory channels, mapped pairwise between domains
• Queues with non-spinning TAS locks at each end
• Threads block on channels entirely in user space
• Messaging is *asynchronous* below thread abstractions
• Can switch to another thread in same address space
  – rather than block waiting for another address space
• Big win:
  Multiprocessor with concurrent client and server threads
URPC latency

C = #cores for clients
S = #cores for server
URPC throughput

\[ C = \text{#cores for clients} \]
\[ S = \text{#cores for server} \]
URPC performance

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Operation</th>
<th>Performance</th>
</tr>
</thead>
<tbody>
<tr>
<td>URPC</td>
<td>Cross-AS latency</td>
<td>93 µs</td>
</tr>
<tr>
<td></td>
<td>Inter-processor overhead</td>
<td>53 µs</td>
</tr>
<tr>
<td></td>
<td>Thread fork</td>
<td>43 µs</td>
</tr>
<tr>
<td>LRPC</td>
<td>Latency</td>
<td>157 µs</td>
</tr>
<tr>
<td></td>
<td>Thread fork</td>
<td>&gt; 1000 µs</td>
</tr>
<tr>
<td>Hardware</td>
<td>Procedure call</td>
<td>7 µs</td>
</tr>
<tr>
<td></td>
<td>Kernel trap</td>
<td>20 µs</td>
</tr>
</tbody>
</table>

All on a Firefly (4-processor CVAX). Irony:
- LRPC, L4 seek performance by optimizing kernel path
- URPC gains performance by bypassing kernel entirely
Discussion

• L4, LRPC:
  – Optimize for synchronous, null RPC performance bypass scheduler
  – Hard to perform accurate resource accounting

• URPC:
  – Integrate with the scheduler
  – Decouple from event transmission
    ⇒ slightly slower null RPC times when idle
    ⇒ higher RPC throughput
    ⇒ lower latency on multiprocessors
Interprocess communication in Barrelfish
Communication stack

- Consensus
  - Replica maintenance

- Multihop routing
  - Multicast tree construction

- Portability layer (C API)
  - Generated from IDL

- Low-level messaging
  - Highly exposed, fixed MTU
  - User-space where possible
  - Polled or event-based

- Typically IPI
  - Adjunct to Interconnect Drivers

Group communication

Routing

Smart stubs

Interconnect drivers

Notification drivers

User-space dispatcher

CPU driver

CPU
Barrelfish communication stack

- Consensus
- Replica maintenance
- Multihop routing
- Multicast tree construction
- Portability layer (C API)
- Generated from IDL
- Low-level messaging
  - Highly exposed, fixed MTU
  - User-space where possible
  - Polled or event-based
- Typically IPI
- Adjunct to Interconnect Drivers

Group communication
Routing
Smart stubs
Interconnect drivers
Notification drivers

User-space dispatcher
CPU driver
CPU
Interconnect Drivers

• *Barely* abstract an inter-core message channel
  – Expose message size (e.g. 64 bytes)
  – Don’t implement flow control, etc. unless it’s there
  – Don’t require privilege, unless you have to
    • May not be able to send capabilities over this channel!
  – Separate out notification, unless it’s coupled
  – Interface can be either polled or event-driven

• General philosophy:
  – Don’t cover up anything
  – Expose all functionality
  – Abstract at a higher layer: in the stubs
CC-UMP Interconnect Driver

• Cache-coherent shared memory
  – inspired by URPC
• Ring buffer of cache-line sized messages
  – 64 bytes or 32 bytes
  – 1 word for bookkeeping; last one written (end of line)
• Credit-based flow control out of band
• One channel per IPC binding (not shared)
CC-UMP: cache-coherent user-space messaging

- Ack pointer advanced via messages in reverse path
- Sender fills in message in next cache-line sized slot
- Unidirectional channels
- Fixed-size circular buffer
- All messages are 64-byte cache lines
- Receiver polls for update at end of message
CC-UMP: cache-coherent user-space messaging

Sending core’s write buffer

Sending core’s cache: invalid => no copy of the line

Receiver’s cache: shared => read-only copy of the line
CC-UMP: cache-coherent user-space messaging

1. Sender starts to write message; h/w combines writes in write buffer

- Sending core’s write buffer
- Sending core’s cache: invalid => no copy of the line
- Receiver’s cache: shared => read-only copy of the line
CC-UOMP: cache-coherent user-space messaging

1. Sender starts to write message; h/w combines writes in write buffer

- Sending core’s write buffer
- Sending core’s cache: invalid => no copy of the line
- Receiver’s cache: shared => read-only copy of the line
CC-Ump: cache-coherent user-space messaging

1. Sender starts to write message; h/w combines writes in write buffer

   Sending core’s write buffer

   Sending core’s cache: invalid => no copy of the line

   Receiver’s cache: shared => read-only copy of the line
CC-UMP: cache-coherent user-space messaging

1. Sender starts to write message; h/w combines writes in write buffer

2. Write buffer fills: fetch target cache line in exclusive (E) state

Sending core’s write buffer
Sending core’s cache: exclusive => clean, writable copy
Receiver’s cache: invalid => out-of-date copy of the line
CC-UMP: cache-coherent user-space messaging

1. Sender starts to write message; h/w combines writes in write buffer

2. Write buffer fills: fetch target cache line in exclusive (E) state

3. Drain buffered writes into cache line, change to modified (M) state

Sending core’s write buffer

Sending core’s cache: modified => dirty r/w copy

Receiver’s cache: invalid => out-of-date copy of the line
CC-UMP: cache-coherent user-space messaging

1. **Sender starts to write message; h/w combines writes in write buffer**
   - Sending core’s write buffer
   - Sending core’s cache: modified => dirty r/w copy

2. **Write buffer fills: fetch target cache line in exclusive (E) state**
   - Receiver’s cache: invalid => out-of-date copy of the line

3. **Drain buffered writes into cache line, change to modified (M) state**
   - Reader’s cache: invalid => out-of-date copy of the line

4. **Reader polls again; own cache is invalid (I) so needs to fetch fresh read-only copy (S)**
   - INVALIDATE
   - PROBE
   - Receiver’s cache: invalid => out-of-date copy of the line

Legend:
- S: Shared
- M: Modified
- I: Invalid
- R: Read-only
- XL: Read-only copy
Conventional wisdom

• Stub compilers are a solved problem (Flick)
  – Optimizing stub compilers compute most efficient way to copy values into a buffer
  – Buffers are assumed to be “big enough”
  – Marshalling code separate from send/receive code
  – Marshalling code doesn’t handle fragmentation/reassembly

• But:
  – Interconnect drivers don’t have a buffer abstraction
  – Transmission units are small (cache lines, registers)
  – Efficient packing varies across interconnect drivers
Flounder and specialized stubs

• **Different** backend code generator for each ICD

• Lots of engineering, but:
  – Haskell makes this easier
  – Code reuse where possible
  – Filet-o-Fish would help more
    (but we don’t use it here)

• Highly optimized: performs final specialization
Stub performance really matters

CC-UMP Interconnect driver
64-byte (16-word) MTU
Nehalem-EX 64-core system
(between packages)
Where does the time go?
Where does the time go?

- Null message
- Intel Nehalem-EX
- AMD Shanghai
- CC-UMP Interconnect driver
Communication binding

1. bind_client(iref, cframe)
2. remote_bind_req(iref, cframe)
3. bind(cframe)
4. ack(sframe)
5. remote_bind_reply(sframe)

Monitors route binding requests and replies
Name service

1. alloc_iref()
2. register(iref, name)
3. query(name)

Name server is orthogonal to IPC system
Intra-OS routing

• Routing within a single machine?
  – Hardware may not give full-mesh connectivity
  – Some Interconnects must be multiplexed in software (tunnelled)
    • E.g. PCIe channel to SCC
    • E.g. Ethernet

• Monitors and library provide *intra-OS routing*
Multicast

• Even with full routes, may need routing for group communication
  – High cost of dropping into software for a hop
  – Balanced with parallelism from e.g. tree topologies

• Routing library provides efficient construction of dissemination trees for specific hardware
  – Built at runtime from on-line measurements
Example: radix tree multicast
Summary

• Multiprocessors are different
  – Real concurrency
  – Exploit parallelism in message send/receive

• Use shared memory to bypass kernel
  – Scheduling decisions decoupled from messages
  – *Spatial* scheduling increasingly important

• Lowest level of a complete stack
  – Stubs, routing, multicast, etc. – almost a network...