Overview

- Multiprocessor OS (Background and Review)
  - How does it work? (Background)
  - Scalability (Review)
- Multiprocessor Hardware
  - Contemporary systems (Intel, AMD, ARM, Oracle/Sun)
  - Experimental and Future systems (Intel, MS, Polaris)
- OS Design for Multiprocessors
  - Guidelines
  - Design approaches
    - Share and Conquer (Slack, Translation)
    - Reduce Sharing (K42, Core, Linux, FlexSC, scalable commutativity)
    - No Sharing (Barre/Sch, fcq)

Uniprocessor OS

Multiprocessor OS

Key design challenges:
- Correctness of (shared) data structures
- Scalability (performance doesn’t suffer)
Correctness of Shared Data

- Concurrency control
  - Locks
  - Semaphores
  - Transactions
  - Lock-free data structures

- We know how to do this:
  - In the application
  - In the OS

Scalability

Speedup as more processors added

- Reality

\[ S(N) = \frac{T_1}{T_N} \]

Scalability and Serialisation

Remember Amdahl’s law

- Serial (non-parallel) portion: application not running on all cores
- Serialisation prevents scalability

```
T_s = 1 = (1 - P) + P
T_p = (1 - P) + P
S(N) = \frac{T_s}{T_p} = \frac{1}{(1 - P) + \frac{P}{N}}
S(\infty) \to \frac{1}{1 - P}
```

Serialisation

Where does serialisation show up?

- Application (e.g., access shared app data)
- OS (e.g., performing syscall)
- How much time is spent in OS?

Sources of Serialisation

- Locking (explicit serialisation)
  - Waiting for a lock \( \rightarrow \) stalls self
  - Lock implementation
- Atomic operations (load bus) \( \rightarrow \) stalls everyone waiting for memory
- Cache coherence traffic loads bus \( \rightarrow \) stalls others waiting for memory

Memory access (implicit)

- Relatively high latency to memory \( \rightarrow \) stalls self
- Processor stalled while cache line is fetched or invalidated
- Affected by latency of interconnect
- Performance depends on data size (cache lines) and contention (number of cores)
More Cache-related Serialisation

False sharing
- Unrelated data structs share the same cache line
- Accessed from different processors
  ➔ Cache coherence traffic and delay

Cache line bouncing
- Shared R/W on many processors
  ➔ E.g., bouncing due to locks: each processor spinning on a lock brings it into its own cache
- Cache coherence traffic and delay

Cache misses
- Potentially direct memory access ➔ stalls self
- When does cache miss occur?
  ➔ Application accesses data for the first time, Application runs on new core
  ➔ Cached memory has been evicted
  ➔ Cache footprint too big, another app ran, OS ran

Multi-What?

- Terminology:
  - core, die (chip), package (module, processor, CPU)
  - Multiprocessor, SMP
    - >1 separate processors, connected by off-processor interconnect
  - Multithread, SMT
    - >1 hardware threads in a single processing core
  - Multicore, CMP
    - >1 processing cores in a single die, connected by on-die interconnect
    - Multicore + Multiprocessor
    - >1 multicore dies in a package (multi-chip module), on-processor interconnect
  - Manycore
    - Lots (>100) of cores

Interesting Properties of Multiprocessors

- Scale and Structure
  - How many cores and processors are there
  - What limits of cores and processors are there
  - How are they organised (access to IO, etc.)
  - Interconnect
    - How are the cores and processors connected
  - Memory Locality and Caches
    - Where is the memory
    - What is the cache architecture
  - Interprocessor Communication
    - How do cores and processors send messages to each other

Contemporary Multiprocessor Hardware

- Intel:
  - Nehalem, Westmere: 10-core, QPI
  - Sandy Bridge, Ivy Bridge: 6-core, ring bus, integrated GH, L3, IO
  - Haswell (Broadwell): 10v-core, ring bus, transactional memory, slices (EP)
  - Skylake (DF) mesh architecture
- AMD
  - K10 (Opteron: Barcelona, Magny Cours): 32 core, Hypertransport
  - Bulldozer, Piledriver, Steambender (Opteron, FX)
    - 16 core, Clustered Multithread: module with 2 integer cores
    - Zero on die Notlik, CPU Complexe (CO)(bcore, private L2)
    - 2x 2-chiplets (FX/2) corelets, 61 die (incl mem controller)
- Oracle (Sun) UltraSparc T3, T2, T3, T4, T5 (Niagara), M5, M7,
  - T5: 16 core, 8 threads/core (2 simultaneous), 64k cache, 8 sockets, 5GHz
- ARM Cortex A9, A53 MPICore, big.LITTLE, DynamIQ
  - ARM Cortex A9, A53 MPICore, big.LITTLE, DynamIQ
  - 4-8 cores, big.LITTLE: A7 > A53, dynamicQ, A75 + A53

Scale and Structure

- ARM Cortex A9

Advanced Bus Interfaces Unit
- ARM AHB 3 clock domains: Optional APB and Address Filter
Interconnect (Bandwidth)

Node 0

Node 6

Node 7

30G/s

60G/s

120G/s

Unidirectional

Interconnect

- Oracle Sparc T2

Interconnect/Structure/Memory

Cluster on Die (COD) Mode

- Supported on 15 & 25 SKUs with 2 Home Agents (10+ cores)
- In memory directory bits & directory cache used on 25 to reduce coherence traffic and cache-to-cache transfer latencies
- Targeted at NOC/NSC optimized workloads where latency is more important than sharing across Caching Agents
- Reduces average LLC hit and local memory latencies
- Can route all requests from reduced set of threads potentially offering higher effective memory bandwidth
- OS/VM aware NUMA and process affinity decisions

Experimental/Future/Non-mainstream Multiprocessor Hardware

- Microsoft Beehive
  - Ring bus, no cache coherence
- Tilera (now Mellanox) Tile64, Tile-Gx
  - 100 cores, mesh network
- Intel Polaris
  - 80 cores, mesh network
- Intel SCC
  - 48 cores, mesh network, no cache coherence
- Intel MIC (Multi Integrated Core)
  - Knight’s Corner/Landing - Xeon Phi
  - 60+ cores, ring bus/mesh

Scale and Structure

Tilera Tile64 (newest: Mellanox TILE-Gx), Intel Polaris
**Cache and Memory and IPC**
- Intel SCC

**Interprocessor Communication**
- Beehive

**Interconnect**
- Intel MIC (Multi Integrated Core) (Knight's Corner/Landing - Xeon Phi)

**Skylake SP**

**Summary**
- Scalability
  - 100+ cores
  - Amdahl's law really kicks in
- Heterogeneity
  - Heterogeneous cores, memory, etc.
  - Properties of similar systems may vary wildly (e.g., interconnect topology and latencies between different AMD platforms)
- NUMA
  - Also variable latencies due to topology and cache coherence
- Cache coherence may not be possible
  - Can't use it for locking
  - Shared data structures require explicit work
- Computer is a distributed system
  - Message passing
  - Consistency and Synchronisation
  - Fault tolerance

**OS DESIGN for Multiprocessors**
**Optimisation for Scalability**

- Reduce amount of code in critical sections
- Increases concurrency
- Fine grained locking
- Lock data not code
- Tradeoff: more concurrency but more locking (and locking causes serialisation)
- Lock fine data structures
- Avoid expensive memory access
- Avoid uncached memory
- Access cheap (closed) memory

**Optimisation for Scalability**

- Reduce false sharing
- Pad data structures to cache lines
- Reduce cache line bouncing
- Reduce sharing
  - E.g: MCS looks up local data
- Reduce cache misses
  - Affinity scheduling: run process on the core where it last ran.
  - Avoid cache pollution

**OS Design Guidelines for Modern (and future) Multiprocessors**

- Avoid shared data
  - Performance issues arise less from lock contention than from data locality
- Explicit communication
  - Region control own communication costs (and predictability)
  - Sometimes it’s the only option
- Tradeoff: parallelism vs synchronisation
  - Synchronisation introduces serialisation
  - Make concurrent threads independent: reduce crit sections & cache misses
- Allocate for locality
  - E.g: provide memory local to a core
- Schedule for locality
  - With cached data
  - With local memory
- Tradeoff: uniprocessor performance vs scalability

**Design approaches**

- Divide and conquer
  - Divide multiprocessor into smaller bits, use them as normal
  - Using virtualisation
  - Using exokernel
- Reduced sharing
  - Brute force & Heroic Effort
  - Find problems in existing OS and fix them
  - E.g: Linux machetching: I/O vs fine-grained locking
  - By design
  - Avoid shared data as much as possible
- No sharing
  - Computer is a distributed system
  - Do extra work to share!

**OS Design Guidelines for Modern (and future) Multiprocessors**

- Avoid shared data
  - Performance issues arise less from lock contention than from data locality
- Explicit communication
  - Region control own communication costs (and predictability)
  - Sometimes it’s the only option
- Tradeoff: parallelism vs synchronisation
  - Synchronisation introduces serialisation
  - Make concurrent threads independent: reduce crit sections & cache misses
- Allocate for locality
  - E.g: provide memory local to a core
- Schedule for locality
  - With cached data
  - With local memory
- Tradeoff: uniprocessor performance vs scalability

**Design approaches**

- Divide and conquer
  - Divide multiprocessor into smaller bits, use them as normal
  - Using virtualisation
  - Using exokernel
- Reduced sharing
  - Brute force & Heroic Effort
  - Find problems in existing OS and fix them
  - E.g: Linux machetching: I/O vs fine-grained locking
  - By design
  - Avoid shared data as much as possible
- No sharing
  - Computer is a distributed system
  - Do extra work to share!

**Divide and Conquer**

**Disco**
- Scalability is too hard!
- Contrast:
  - ca. 1985, large ccNUMA multiprocessors appearing
  - Scaling聆 requires extensive modifications
- Idea:
  - Implement a scalable VMM
  - Run multiple OS instances
- VMM has most of the features of a scalable OS:
  - Activation/exec
  - Page iteration, remapping, etc.
- VMM substantially simpler/cheaper to implement
- Modern incarnations of this:
  - Virtual servers (Eucalyptus, etc.)
  - Research (Cerberus)

**Disco Architecture**

- Modern scalability on scalable multiprocessors: Bugnion et al., 2007
Space-Time Partitioning

Tessellation
- Space-Time partitioning
- 2-level scheduling
- Context:
  - 2009—highly parallel multicore systems
  - Berkeley Par Lab
- Tessellation OS (space-time partitioning)
  - Multicore Hardware
  - Disk
  - Research Core

Reduce Sharing

K42
- Context:
  - 1997-2006: OS for ccNUMA systems
  - IBM, S' Torrata (Tornado, Hurricane)
- Goals:
  - High locality
  - Scalability
- Object Oriented
  - Fine-grained objects
  - Clustered (Distributed) Objects
  - Data locality
- Deferred deletion (RCU)
  - Avoid locking
  - NUMA aware memory allocator
  - Memory locality

K42: Fine-grained objects

K42: Clustered objects

- Globally valid object reference
- Resolves to
  - Processor local representative
- Sharing, locking strategy local to each object
- Transparency
  - Eases complexity
  - Controlled introduction of locality
- Shared counter
  - inc, dec: local access
  - vol communication
- Fast path
  - Access mostly local structures
K42 Performance

Linux Brute Force Scalability

Scalability of the API

Corey

Linux Brute Force Scalability

Scalable Commutativity Rule
**Commuter: An Automated Scalability Testing Tool**

- Symbolic model
- Commutativity conditions
- Test cases
- Linux, Mach, VMM

**FlexSC**

- Asynchronous system calls
  - Batch system calls
  - Run them on dedicated cores
- FlexSC-Threads
  - M on N
  - M + N

**FlexSC Results**

- Apache
- FlexSC batching, syscall code redirect

**No sharing**

- Multikernel
  - Barrelinux
  - PVS: factored operating system

**Barrelinux**

- Context:
  - 2407 large multikernel machines appearing
  - 100s of cores on the horizon
  - NUMA (pc and non-pc)
- Approach:
  - Structure OS as distributed system
- Design principles:
  - Interprocessor communication is explicit
  - OS structure framework neutral
- Goals:
  - Support and manage heterogeneous hardware

**Context:**

- 2010, commodity multicores
- U Torontno

**Goal:**

- Reduce context switch overhead of system calls

**Syscall context switch:**

- Usual mode switch overhead
- But: cache and TLB pollution!

**Results:**

- User performance (IPC)
- Syscall execution time (cycles)

<table>
<thead>
<tr>
<th>Syscall</th>
<th>Instructions</th>
<th>Cycles</th>
<th>IPC</th>
<th>cache</th>
<th>d-cache</th>
<th>L1d</th>
<th>L1I</th>
<th>L2</th>
<th>TLB</th>
</tr>
</thead>
<tbody>
<tr>
<td>elipt</td>
<td>4979</td>
<td>15385</td>
<td>0.97</td>
<td>33</td>
<td>186</td>
<td>113</td>
<td>2899</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>printf</td>
<td>7789</td>
<td>15385</td>
<td>0.94</td>
<td>33</td>
<td>186</td>
<td>113</td>
<td>2899</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>malloc</td>
<td>3767</td>
<td>15385</td>
<td>0.94</td>
<td>33</td>
<td>186</td>
<td>113</td>
<td>2899</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>malloc+</td>
<td>3767</td>
<td>15385</td>
<td>0.94</td>
<td>33</td>
<td>186</td>
<td>113</td>
<td>2899</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>malloc-</td>
<td>3767</td>
<td>15385</td>
<td>0.94</td>
<td>33</td>
<td>186</td>
<td>113</td>
<td>2899</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>malloc-M</td>
<td>3767</td>
<td>15385</td>
<td>0.94</td>
<td>33</td>
<td>186</td>
<td>113</td>
<td>2899</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>malloc-D</td>
<td>3767</td>
<td>15385</td>
<td>0.94</td>
<td>33</td>
<td>186</td>
<td>113</td>
<td>2899</td>
<td>21</td>
<td></td>
</tr>
</tbody>
</table>
Barrelfish: Replication

- Kernel + Monitor:
  - Only memory shared for message channels
  - Monitor:
    - Collectively coordinate system-wide state
  - System-wide state:
    - Memory allocation tables
    - Address space mappings
    - Capability lists
  - What state is replicated in Barrelfish
  - Consistency and Coordination
    - Retype: two-phase commit to globally execute operation in order
    - Page (re)mapping: one-phase commit to synchronise TLBs

Barrelfish: Communication

- Different mechanisms:
  - Intra-core
    - Kernel endpoints
  - Inter-core
    - URPC
  - URPC:
    - Uses cache coherence + polling
      - Shared buffer
        - Sender writes a cache line
        - Receiver polls on cache line
        - [last word is no part message]
      - Polling:
        - Cache only changes when sender writes, so poll is cheap
        - Switch to block and IP if wait is too long

Barrelfish: Results

- Broadcast vs Multicast

- TLB shootdown
Summary

- Trends in multicore
  - Scale (100+ cores)
  - NUMA
  - No cache coherence
  - Distributed systems
  - Heterogeneity
- OS design guidelines
  - Avoid shared data
  - Explicit communication
  - Locality
- Approaches to multicore OS
  - Partition the machine (Skizo, Tessellation)
  - Reduce sharing (H8, Corey, Linux, FlexSC, scalable commutativity)
  - No sharing (Barrelfish, Fos)