Alternative system models

Tudor David
Outline

- The multi-core as a distributed system
- Case study: agreement
- The distributed system as a multi-core
The system model

- **Concurrency**: several communicating processes executing at the same time;

- **Implicit communication**: shared memory;
  - Resources – shared between processes;
  - Communication – implicit through shared resources;
  - Synchronization – locks, condition variables, non-blocking algorithms, etc.

- **Explicit communication**: message passing;
  - Resources – partitioned between processes;
  - Communication – explicit message channels;
  - Synchronization – message channels;

  Whatever can be expressed using shared memory can be expressed using message passing.
So far – shared memory view

• set of registers that any process can read or write to;
• communication – implicit through these registers;
• problems:
  — concurrency bugs – very common;
  — scalability - not great when contention high;

Scalability
Queue Lock Throughput

Mops/s

# cores
Alternative model: message passing

• more verbose
  — but – can better control how information is sent in the multi-core.

• how do we get message passing on multicores?
  — dedicated hardware message passing channels (e.g. Tilera)
  — more common – use dedicated cache lines for message queues
Programming using message passing

• System design – more similar to distributed systems;
• Map concepts from shared memory to message passing;

• A few examples:
  — Synchronization, data structures: flat combining;
  — Programming languages: e.g. Go, Erlang;
  — Operating systems: the multikernel (e.g. Barrellfish)
Barrelfish: All Communication - Explicit

- Communication – exclusively message passing
- Easier reasoning: know what is accessed when and by whom
- Asynchronous operations – eliminate wait time
- Pipelining, batching
- More scalable
Barrelfish: OS Structure – Hardware Neutral

Separate OS structure from hardware

Machine dependent components

• Messaging
• HW interface

Better layering, modularity

Easier porting
Barrelfish: Replicated State

- No shared memory => replicate shared OS state
- Reduce interconnect load
- Decrease latencies
- Asynchronous client updates
- Possibly NUMA aware replication
Consistency of replicas: agreement protocol

Implicit communication
(shared memory)

Explicit communication
(message passing)

Locally cached data

Replicated state

State machine replication → Total ordering of updates → Agreement

High availability, High scalability

How should we do message-passing agreement in a multi-core?
Outline

- The multi-core as a distributed system
- **Case study: agreement**
- The distributed system as a multi-core
First go – a blocking protocol

Two-Phase Commit (2PC)

1. broadcast Prepare
2. wait for Acks
3. broadcast Commit/Rollback
4. wait for Acks

Blocking, all messages go through coordinator
Is a blocking protocol appropriate?

Blocking agreement – only as fast as the slowest participant

- Scheduling?
- I/O?

Use a non-blocking protocol

“Latency numbers every programmer should know”

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 cache reference</td>
<td>0.5</td>
</tr>
<tr>
<td>Branch mispredict</td>
<td>5</td>
</tr>
<tr>
<td>L2 cache reference</td>
<td>7</td>
</tr>
<tr>
<td>Mutex lock/unlock</td>
<td>25</td>
</tr>
<tr>
<td>Main memory reference</td>
<td>100</td>
</tr>
<tr>
<td>Compress 1K bytes</td>
<td>3000</td>
</tr>
<tr>
<td>Send 1K bytes over 1 Gbps network</td>
<td>10000</td>
</tr>
<tr>
<td>Read 4K randomly from SSD</td>
<td>150000</td>
</tr>
<tr>
<td>Read 1MB sequentially from memory</td>
<td>250000</td>
</tr>
<tr>
<td>Round trip within datacenter</td>
<td>500000</td>
</tr>
<tr>
<td>Read 1MB sequentially from SSD</td>
<td>1000000</td>
</tr>
<tr>
<td>Disk seek</td>
<td>100000000</td>
</tr>
<tr>
<td>Read 1 MB sequentially from disk</td>
<td>200000000</td>
</tr>
<tr>
<td>Send packet CA-&gt;Netherlands-&gt;CA</td>
<td>150000000</td>
</tr>
</tbody>
</table>

Source: Jeff Dean
Non-blocking agreement protocols

**Consensus** ~ non-blocking agreement between distributed processes on one out of possibly multiple proposed values

**Paxos**
- Tolerates non-malicious faults or unresponsive nodes: in multi-cores, slow cores
- Needs a majority of responses to progress (tolerates partitions)

Lots of variations and optimizations: CheapPaxos, MultiPaxos, FastPaxos etc.

**Phase 1: prepare**
**Phase 2: accept**

**Roles:**
- Proposer
- Acceptor
- Learner

Usually – all roles on a physical node (Collapsed Paxos)
MultiPaxos

- Unless failed, keep same leader in subsequent rounds
Does MultiPaxos scale in a multi-core?

MultiPaxos, 3 replicas

Limited scalability in the multi-core environment
A closer look at the multi-core environment

Where does time go when sending a message?

Large networks:
Minimize number of rounds/instance

Multi-core:
Minimize the number of messages
Can we adapt Paxos to this scenario?

Using one acceptor significantly reduces the number of messages
1Paxos: The failure-free case

1. P2: obtains active acceptor A1 and sends prepare_request(pn)
2. A1: if pn - max. proposal received, replies to P2 with ack
3. P2 -> A1 accept_request(pn, value)
4. A1 broadcasts value to learners

Common case: only steps 3 and 4
1. P2 leader?

2. PaxosUtility: P2 proposes
   - A3 active acceptor
   - Uncommitted proposed values

3. P2 -> A3: prepare_request
1. A1 – active acceptor?

2. PaxosUtility: P3 new leader and A1 active acceptor

3. P3 → A1: prepare_request
Switching leader and acceptor

The trade-off:
while leader and active acceptor non-responsive at the same time
✔ safety
✖ liveness

Why is it reasonable?
• small probability event
• no network partitions
• if nodes not crashed, but slow → system becomes responsive after a while
Latency and throughput

Smaller # of messages - smaller latency and increased throughput
Agreement - summary

Multi-core – message passing distributed system, but distributed algorithm implementations different

Agreement in multi-cores
- non blocking
- reduced # of messages

Use one acceptor: 1Paxos
- reduced latency
- increased throughput
Outline

- The multi-core as a distributed system
- Case study: agreement
- The distributed system as a multi-core
Remote Direct Memory Access (RDMA)

Read/Write remote memory
NIC performs DMA requests

Great performance
Bypasses the kernel
Bypasses the remote CPU

Source: A. Dragojevic
FaRM: Fast Remote Memory (NSDI'14) – Dragojevic et al.

Source: A. Dragojevic