Introduction to Multiprocessors

Erik Hagersten
Uppsala University

DARK2 in a nutshell
1. Memory Systems (caches, VM, DRAM, microbenchmarks, …)
2. Multiprocessors (TLP, coherence, interconnects, scalability, clusters, …)
3. CPUs (pipelines, ILP, scheduling, Superscalars, VLIWs, embedded, …)
4. Future: (physical limitations, TLP+ILP in the CPU,...)

How do we get good performance?

- Creating and exploring:
  1) Locality
     a) Spatial locality
     b) Temporal locality
     c) Geographical locality
  2) Parallelism
     a) Instruction level
     b) Thread level

MP Taxonomy

Flynn’s Taxonomy

{Single,Multiple}Instruction + {Single,Multiple}Data

- SISD - Our good old simple CPUs
- SIMD – Vectors, “MMX”, DSPs, CM-2,...
- MIMD – TLP, cluster, shared-mem MP,...
- MISD – Can’t think of any...
**SIMD**

- **Program:**
  - ...
  - ...
  - ...

**SIMD: Thinking Machine**
- Connection Machine: CM1, CM2, CM200 (at KTH ~1990: CM200 "Bellman")
- One-bit ALU and a small local memory
- FP accelerator available
- Hypercube interconnect
- Programmed in "ASM", *C and *Lisp
- Hard to program (in my opinion...)

**Other Flavors of SIMD**
- MMX/AltiVec/VIS instructions/...
  - Divide register content into smaller items (e.g., bytes)
  - Special instructions operate on all items in parallel, e.g., BYTE-COMPARE...
- Some DSPs (Digital Signal Processors)
- Some Image Processors

**Vector architectures**
- CRAY, NEC, Fujitsu, ...
  - VECTory Processors
    - LD/ST operate on vectors of data
    - ALU Ops operate on vectors of data
  - Example:
    - 8 vector register contains 64 vector entries each
    - A single LD/ST instr loads/stores entire vectors
    - A single ALU instr V1 op V2 op V3
    - 64 bit mask vectors make execution conditional
    - Overlaps Mem and ALU ops
    - One form of "SIMD" -- Single Instruction Multiple Data

**MIMD: Message-passing**
- **SIMD**
- **MIMD**
  - **Message-passing**
  - **Shared Memory**
    - **UMA**
    - **NUMA**
    - **COMA**
  - **Fine-grained**
  - **Coarse-grained**

**Message-passing Arch MIMD**
- **Explicit Messages**
  - **Program:**
    - ...
    - ...
    - ...
  - **Message-passing**
Message-Passing HW

- Programmed in MPI or PVM (or HPFortran...)
- Thinking Machines: CM5
- Intel (!!): Paragon
- IBM: SP2
- Meiko (Bristol, UK!!): CS2
- Today: Clusters with high-speed interconnect

- Clusters can be used as message-passing HW, but is most often used as capacity computing (i.e., throughput computing)

Dataflow

- Often programmed in functional languages (e.g., ID)
- Compile program to Dataflow graph
- Operands + graph = executable
- Operation ready when the source operands are available

Dataflow Example:

```
X := A + B
Y := C + D
If (X > Y)
  output X
else
  output Y
```

Static Dataflow (Dennis)

Each operand executed exactly once per program
Location assigned for each input data

Dynamic Dataflow (Arvind)

- Allows for recursion and loops
- Each invocation is assigned a “color”
- Pairs of operands are matched dynamically
  - Based on {Color, Operation}
  - In the Waiting-Matching Section (i.e., a cache)
- One problem: too much parallelism in the wrong place
Carlstedts Elektornik
Gunnar Carlstedt, Staffan Truve' et al

- Processor “8601”
  - Gothenburg 1990-1997
  - Functional language “H”
  - Execution performed by a reduction a CAM memory
  - ALU rarely used
  - Many parallel processors on a wafer (Wafer-scale integration)
- CRT (Carlstedt Research Technology)

Today’s Topic

The era of the "Rocket Science Supercomputers” 1980-1995

- The one with the most blinking lights wins
- The one with the niftiest languages wins
- The more different the better!

The server market 1995

![Server Size vs. Price Chart]

The target of the rocket science supercomputers

Models of parallelism

- Processes (fork or & in UNIX)
  - A parallel execution, where each process has its own process state, e.g., memory mapping
- Threads (thread chreate in POSIX)
  - Parallel threads of control inside a process
  - There are some thread-shared state, e.g., memory mappings.
- Sverker will tell you more.

Coherent Memory

Erik Hagersten
Uppsala University
Programming Model:

Shared Memory

Thr Thr Thr Thr Thr Thr Thread

Adding Caches: More Concurrency

Shared Memory

Thr Thr Thr Thr Thr Thr Thread

Thr Thr Thr Thr Thr Thr Thr Thread

Inv

Thread

Read A

Read B

Read A

Read A

Summing up Coherence

There can be many copies of a datum, but only one value.

The Cache Coherent Memory System

Shared Memory

A:

Thread

Read A

Read A

Read B

Write A

B:

Read A

Read A

Read B

Read A

There is a single global order of value changes to each datum.

Too strong definition!
Implementation options for memory coherence

- Two coherence options
  - Snoop-based
  - Directory-based
- Different memory models
- Varying scalability

Example: Bus Snoop MOSI

BUSrts: ReadToShare (reading the data with the intention to read it)
BUSrtw, ReadToWrite (reading the data with the intention to modify it)
BUSwb: Writing data back to memory
BUSinv: Invalidating other caches copies

Example: CPU access MOSI

CPUwrite: Caused by a store miss
CPUread: Caused by a load miss
CPUrepl: Caused by a replacement

"Upgrade" in snoop-based
Directory-based coherence: per-cacheline info in the memory

"Upgrade" in dir-based

Cache-to-cache in dir-based

Cache-to-cache in snoop-based

A New Kind of Cache Miss
- Capacity – too small cache
- Conflict – limited associativity
- Compulsory – accessing data the first time
- Communication (or "Coherence") [Jouppi]
  - Caused by downgrade (modified→shared)
    "A store to data I had in state M, but now it's in state S" 😊
  - Caused my invalidation (shared→invalid)
    "A load to data I had in state S, but now it's been invalidated" 😊

False sharing
Communication misses even though the threads do not share data "the cache line is too large"
Why snoop?  
- A "bus": a serialization point helps coherence and memory ordering  
- Upgrade is faster [producer/ consumer and migratory sharing]  
- Cache-to-cache is much faster [i.e., communication...]  
- Synchronization, a combination of both  
  ...but it is hard to scale the bandwidth

Why directory-based  
- P2P messages → high bandwidth  
- Suits out-of-the-box coherence  
- Note:  
  - Dir-based can be used to build a uniform-memory architecture (UMA)  
  - Bandwidth will be great!!  
  - Memory latency will be OK  
  - Cache-to-cache latency will not!

Update Instead of Invalidate?  
- Write the new value to the other caches holding a shared copy (instead of invalidating...)  
- Will avoid coherence misses  
- Consumes a large amount of bandwidth  
- Hard to implement strong coherence  
- Few implementations: SPARCCenter2000, Xerox Dragon

Update in MOSI snoop-based

Memory Ordering (aka Memory Consistency)  
-- tricky but important stuff

Erik Hagersten  
Uppsala University  
Sweden

Q: What value will get printed?

Initially A = B = 0  

...  

A := 1  

...  

B := 1  

...  

Print A
Dekker’s Algorithm

Initially A = B = 0

fork

A := 1
if (B == 0) print("A won")
B := 1
if (A == 0) print("B won")

Q: Is it possible that both A and B wins?

Memory Ordering

- Defines the guaranteed memory ordering
- Is a “contract” between the HW and SW guys
- Without it, you cannot say much about the result of a parallel execution

Memory Ordering

- Defines the guaranteed memory ordering
- Is a “contract” between the HW and SW guys
- Without it, you cannot say much about the result of a parallel execution

In which order were these threads executed?

( A’ denotes a modified value to the data at addr A)

Thread 1

LD A
ST B’
LD C
ST D’
LD E
...

Thread 2

LD B’
ST C’
LD D
ST E’
...

LD A’
ST B’
LD C
ST D’
LD E
...

(ST A’ happend before ST A)
Sequential Consistency (SC) Violation

→ Dekker: both wins

Access graph

Both Left and Right wins → SC violation

Cyclic access graph → Not SC (there is no global order)

SC is OK if one thread wins

Only Right wins → SC is OK

Not cyclic graph → SC

One implementation of SC in dir-based
(....without speculation)

“Almost intuitive memory model”
Total Store Ordering (P. Sindhu)

Global order achieved by interleaving all store accesses from different threads

"Programmer’s intuition is maintained"

• Store causality? Yes
• Does Dekker work? No
• Unnecessarily restrictive ==> performance penalty

TSO HW Model

Stores are moved off the critical path
Coherence implementation can be the same as for SC
**TSO**

- Flag synchronization works
  
  ```c
  A := data while (flag != 1) {}
  flag := 1 X := A
  ```

- Provides causal correctness

**Dekker’s Algorithm**

Initially $A = B = 0$

```
“fork”
A := 1
Membar #StoreLoad if (B==0) print(“A won”)
B := 1
Membar #StoreLoad if (A == 0) print(“B won”)
```

Q: What value will get printed? Answer: 1

**Weak/release Consistency** (M. Dubois, K. Gharachorloo)

- Most accesses are unordered
- “Programmer’s intuition is not maintained”
  - Store causality? No
  - Does Dekker work? No
- Global order only established when the programmer explicitly inserts memory barrier instructions
  - ++ Better performance!!
  - --- Interesting bugs!!

**Example 1: Causal Correctness Issues**

Q: What value will get printed? Answer: 1
Example 1: Causal Correctness Issues

Shared Memory

Thread

Thread

Thread

Read A

A := 1

Write B

While (A == 0) {}

Read A

While (B == 0) {}

Print A

INV

What is the value of A?

It depends...

A: if store causality → "1" will be printed

What is the value of A?

It depends...
Dekker’s Algorithm

Initially A = B = 0

```
fork

A := 1
if (B == 0) print("A won")

B := 1
if (A == 0) print("B won")
```

Q: Is it possible that both A and B win?
A: Only known if you know the memory model

Learning more about memory models


RFM: Read the F****n Manual of the system you are working on!
(Different microprocessors and systems supports different memory models.)

Issue to think about: What code reordering may compilers really do?
Have to use “volatile” declarations in C.

Implementing Coherence (and Memory Models...)

Erik Hagersten
Uppsala University
Sweden

Common Cache States
- M – Modified
  My dirty copy is the only cached copy
- E – Exclusive
  My clean copy is the only cached copy
- O – Owner
  I have a dirty copy, others may also have a copy
- S – Shared
  I have a clean copy, others may also have a copy
- I – Invalid
  I have no valid copy in my cache

Some Coherence Alternative
- MSI
  Writeback to memory on a cache2cache.
- MOSI
  Leave one dirty copy in a cache on a cache2cache
- MOESI
  The first reader will go to E and can later write cheaply

Snoop-based Protocol Implementation
The Cache Coherent Memory System

Upgrade – the requesting CPU

Upgrade – the other CPUs

Modern snoop-based architecture -- dual tags

"Upgrade" in snoop-based

The Cache Coherent Cache-to-cache
**Cache2cache – the requesting CPU**

- **CPUwrite**: Caused by a store miss
- **CPUread**: Caused by a load miss
- **CPUrepl**: Caused by a replacement

---

**Cache-to-cache – the other CPU**

- **BUSrts**: Read to Share (reading the data with the intention to read it)
- **BUSrtw**: Read to Write (reading the data with the intention to modify it)
- **BUSwb**: Writing data back to memory
- **BUSinv**: Invalidating other caches copies

---

**Cache-to-cache in snoope-based**

- Shared Memory

---

**Yet Another Cache-to-cache**

- **BUSrts**: Read to Share (reading the data with the intention to read it)
- **BUSrtw**: Read to Write (reading the data with the intention to modify it)
- **BUSwb**: Writing data back to memory
- **BUSinv**: Invalidating other caches copies

---

**Revisiting Memory Models**

- What memory model is supported?

---

**What Approach to Shared Memory**

- (a) Shared cache
- (b) Direct-mapped (cache memory)
- (c) Distributed memory
- (d) Striped (cache memory)
- (e) Distributed memory
- (f) Numa (cache memory)

---

**Yet Another Cache-to-cache**

- **BUSrts**: Read to Share (reading the data with the intention to read it)
- **BUSrtw**: Read to Write (reading the data with the intention to modify it)
- **BUSwb**: Writing data back to memory
- **BUSinv**: Invalidating other caches copies
All three RISC CPUs in a MOSI shared-memory sequentially consistent multiprocessor executes the following code almost at the same time:

```plaintext
while(A != my_id){}; /* this is a primitive kind of lock*/
B := B + A * 2;
A := A + 1; /* this is a primitive kind of unlock */
<some other execution replaces A and B from the caches, if still present>
<after a long time>

Initially, CPU1 has its local variable my_id=1, CPU has my_id=2 and CPU3 has my_id=3 and the globally shared variables A is equal to 1 and B is equal to 0. CPU2 and 3 are starting slightly ahead of CPU1 and will execute the first while statement before CPU1. Initially, both A and B only reside in memory.

The following four bus transaction types can be seen on the snooping bus connecting the CPUs:
• RTS: ReadToShare (reading the data with the intention to read it)
• RTW, ReadToWrite (reading the data with the intention to modify it)
• WB: Writing data back to memory
• INV: Invalidating other caches copies

Show every state change and/or value change of A and B in each CPU's cache according to one possible interleaving of the memory accesses. The parallel execution is done for all of the CPUs, the cache lines still in the caches will be replaced. These actions should also be shown. For each line, also state what bus transaction occurs on the bus (if any) as well as which device is providing the corresponding data (if any).

CPC action | RTS (A) | RTS (B) | WB | INV
---|---|---|---|---
CPU1: replace A | - | - | - | -
CPU2: replace B | - | - | - | -
CPU3: | - | - | - | -

Example of a state transition sheet:

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
```

```
An E6000 Proc Board

80 signals = addr, uid, arb, ...

288 signals = 256 data + ECC

An I/O Board

80 signals = addr, uid, arb, ...

288 signals = 256 data + ECC

Split-Transaction Bus

- Split bus transaction into request and response sub-transactions
- Separate arbitration for each phase
- Other transactions may intervene
  - Improves bandwidth dramatically
  - Response is matched to request
  - Buffering between bus and cache controllers

Gigaplane Bus Timing

Electrical Characteristics of the Bus

- At most 16 electrical loads per signal
- 8 boards from each side (ex. 15 CPU+1 I/O)
- 20.5 inches "centerplane"
- Well controlled impedance
- ~350-400 signals
- Runs at 90/100 MHz

Address Controller

 addr arb aid did

ctrl
Dual State Tags

Timing of a single read trans
bd 1 reading from mem 2

Protocol tuned for timing

Foreign and own transactions queue in IQ
State Change on Address Packet

• Data “A” initially resides in CPU7’s cache
• CPU1: Issues a store request to “A”
• CPU1: Read-To-Write req, ID=d, (i.e., “write request”)
• CPU13: LD “A” -> Read-To-Shared req, ID=e
• CPU15: ST “A” -> RTW req , ID=f

mRTO stored in IQCPU1
Own read IQtrans retired when data arrives
Later requests for A queued in IQCPU1 behind mRTO
IQCPU1 will eventually store: <mRTWreq, fRTSw, fRTWreq>
A cascade of "write requests"

- Initially resides in CPU7's cache
- CPU1: RTO, ID=a
- CPU2: RTO, ID=b
- ...
- CPU5: RTO, ID=f

Snoop tags =< I
IQ1 =< mRTOIDa, fRTOIDb>
IQ2 =< mRTOIDb, fRTOIDc>
...
IQ5 =< mRTOIDf>
...
IQ7 =< fRTOIDa>

Micro Benchmark Signature

for (times = 0; times < Max; times++) /* many times*/
for (i=0; i < ArraySize; i = i + Stride/4)
dummy = A[i]; /* touch an item in the array */

Options for building SMPs

1 (4): A single bus

- A single bus for address and data
- Snooping coherence protocol
- ++ Low latency (300ns)
- ++ Trivial memory ordering model
- -- Not very scalable (3 GB/s)
- Ex: Sun E6000, SIG Chalenger, Sequent Symmetry, Intel: Xeon, P6

2 (4): Multiple busses

- Multiple busses striped on some address bit
- Snooping coherence protocol
- ++ More scalable (6 GB/s)
- -- More complicated memory ordering model
- -- Pin-out problem
- Ex: Sun Dragon SC2000 (2 busses), CRAY CS64000 (4 busses)
Options for building SMPs 3 (4): Directory-based
- General interconnect
- Directory-based coherence (point-to-point)
  - ++ More scalable
  - ++ Fewer pins
  - -- Longer Latency (especially cache-to-cache, 800ns)
- Ex: HP V9000, IBM S60

Options for building SMPs 4 (4): Data Crossbar switch
- One or many address busses
- Snooping coherence protocol
- A separate crossbar switch for data
  - ++ More scalable (12 GB/s)
  - ++ Fewer pins (better use of the wide datapaths)
  - -- Longer Latency (500 ns)
- Ex: Sun E10000 StarFire
Implementing Sun’s SunFire 6800

Erik Hagersten
Uppsala University
Sweden

Sun Fire 6800: synchronous & fixed snooping @150MHz; CL=64B

Hierarchical address repeaters

<table>
<thead>
<tr>
<th>Address</th>
<th>Share</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Separate crossbar data switch hierarchy

L2 cache = 8MB, snoop tags on-chip
CPU 1=GHz UltraSPARC III
Mem= 4+GB/CPU

FirePlane, 24 CPUs

Address Rep.

CPU board


ID = <CPU#, Uid>
Here it is!!

ID = \langle CPU#, Uid \rangle
**FirePlane, 24 CPUs**

- Data Repeater
- Data, ID

Latency = 200-240ns
DataBW= 14.4 GB/s
SnoopBW=9.6 GB/s

---

**How to scale snoop BW**

- 1988: 80MB/s → 1997: 10GB/s (w/64B CL)
  - Split transactions
  - Out-of-order reply
  - P2P implementation (rings and hierarchies)
  - On-chip snoop tags
- Future (10-20x improvements is possible)
  - Multiple striped busses
  - Larger cachelines
  - Larger cache lines (may result in false sharing ©)
  - Variable cache line size (UART research)
  - Asynchronous snooping
  - Multi-facet (U. of Wisconsin)

---

**Scalable Shared-Memory Implementations**

Erik Hagersten
Uppsala University
Sweden

---

**What Approach to Shared Memory**

- Bus-based (1980s)
- Interconnection network
- Distributed-memory
- Cache-only

---

**Three options**

- Interconnect
- COMA cache-only (@SICS)
- NUMA non-uniform
- UMA uniform (a.k.a. SMP)
Directory-based snooping: NUMA. Per-cacheline info in the home node

Full mapped directory

Fully mapped directory

"Upgrade" in dir-based

Reducing the Memory Overhead: SCI
Overflow Schemes for Limited Pointers

- **Broadcast (Dir,B)**
  - broadcast bit turned on upon overflow
  - bad for widely-shared invalidated data

- **No-broadcast (Dir,NB)**
  - on overflow, new sharer replaces one of the old ones (invalidated)
  - bad for widely read data

- **Coarse vector (Dir,CV)**
  - change representation to a coarse vector, 1 bit per k nodes
  - on a write, invalidate all nodes that a bit corresponds to

Overflow Schemes (contd.)

- **Software (Dir,SW)**
  - trap to software, use any number of pointers (no precision loss)
  - MIT Alewife: 5 ptrs, plus one bit for local node
  - but extra cost of interrupt processing on software
    - processor overhead and occupancy
    - latency
    - 40 to 425 cycles for remote read in Alewife
    - 84 cycles for 5 inval, 707 for 6.

- **Dynamic pointers (Dir,DP)**
  - use pointers from a hardware free list in portion of memory
  - manipulation done by hw assist, not sw
    - e.g. Stanford FLASH

Protocol Enhancements for Latency

- **Forwarding messages: memory-based protocols**
  - L = Local node (issuing the request)
  - H = Home node (where the dir resides)
  - R = Remote node(s) (where the data resides)
  - L and/or H and/or R may be identical node...

Potential issues

- **Memory model requirements**
  - Race condition:
    - Replacement/read race
    - Unordered network ➞ read/read race
  - Solution: NACK

- **Deadlock**: circular dependence ➞ HALT
- **Livelock**: lots of activity, no progress

cc-NUMA issues

- Memory placement is key!
- Gotta’ migrate data to where it’s being used
- Gotta’ have cache affinity
  - Long time between process switches in the OS
  - Reschedule processor on the CPU it run last
- Origin 2000’s migration always turned off

Sun’s WildFire System

Erik Hagersten
Uppsala University
Sweden
**Sun’s WildFire System**

- Runs unmodified SMP apps in a more scalable way than E6000
- Modifications to E6000 snooping to be extended
- CPUs generate local address OR global address
- Global address --> no replication (NUMA)
- Coherent Memory Replication (~Simple COMA@ SICS)
- Hardware support for detecting migration/replication pages
- Directory cache + address translation cache backed by memory
- Deterministic directory implementation (easy to verify)

**WildFire:** One Solaris spanning four nodes

- Sun’s WildFire System
  - Runs unmodified SMP apps in a more scalable way than E6000
  - Modifications to E6000 snooping to be extended
  - CPUs generate local address OR global address
  - Global address --> no replication (NUMA)
  - Coherent Memory Replication (~Simple COMA@ SICS)
  - Hardware support for detecting migration/replication pages
  - Directory cache + address translation cache backed by memory
  - Deterministic directory implementation (easy to verify)

**COMA: self-optimizing DSM**

- ccNUMA
  - Self-optimizing architecture
  - Problem at high memory pressure
  - Complex hardware and coherence protocol

**Adaptive S-COMA of Large SMPs**

- A page may have space allocated in many nodes
- HW maintains memory coherence per cache line
- Replication under SW control --> simple HW (S-COMA)
- Adaptive replication algorithm in OS (R-NUMA)
- Coherent Memory Replication (CMR)
- Hierarchical affinity scheduler (HAS)
- Few large nodes --> simple interconnect and coherence protocol

**A WildFire Node**

- 16 slots with either CPUs, IO or...
- WildFire extension board
  - Up to 28 UltraSPARC processors
  - Gigaplane™ bus has peak bw 2.67 GB/s
  - Local access time of 330ns (lmbench)

**Sun WildFire Interface Board**

- SRAM
- ADDR Controller
- Data Buffers
- Link Link Link
- This space for rent
Sun WildFire Interface Board

WildFire as a vanilla "NUMA"

NUMA -- local memory access

NUMA -- remote memory access

Global Cache Coherence Prot.

NUMA -- local memory access
**Gigaplane Bus Timing**

- Data w/ E-miss-bit
- Associative Counters

**WildFire Bus Extensions**

- Asserted by WildFire
- Resent by WildFire
- Ignore transaction squashes an ongoing transaction => not put in IQ
- WildFire eventually reissues the same transaction
- RTSF -- a new transaction sends data to CPU and memory

**WildFire Directory -- only 4 nodes!!**

- k nodes (with one or more procs).
- With each cache-block in memory: k presence-bits, 1 dirty-bit
- With each cache-block in cache: 1 valid bit, and 1 dirty (owner) bit
- ReadRequest from main memory by processor i:
  - If dirty-bit OFF then { read from main memory; turn p[i] ON}
  - if dirty-bit ON then { recall line from dirty proc (cache state to shared); update memory; turn dirty-bit OFF; turn p[i] ON; supply recalled data to i;}

**NUMA "detecting excess misses"**

- OS Initializes a CMR page
- Address Translation
  - Grey. <--> Yel...
- Init acc right to INV
An access to a CMR page

Address Translation (AT) overhead = 8B/8kB = 0.1%
No extra latency added

An access to a CMR page (hit)

Access right OK? YES!

Deterministic Directory
- MOSI protocol, fully mapped directory (one bit/node)
- Directory blocking: one outstanding trans/cache line
- Directory blocks new requests until completion received
- The directory state and cache state always in agreement (except for silent replacement...)

Replication Issues Revisited
- "Physical" memory
- Only "promising" pages are replicated
- OS dynamically limits the amount of replication
- Solaris CMR changes in the hat_layer (=port)
Advantages of Multiprocessor Nodes

Pros:
- amortization of fixed node costs over multiple processors
- can use commodity SMPs
- fewer nodes to keep track of in the directory
- much communication may stay within node (NUCA)
- can share "node caches" (WildFire: Coherent Memory Replication)

Cons:
- bandwidth shared among processors and interface
- bus may increases latency to local memory
- snoopy bus at remote node increases delays there too

Memory cost of replication

Example: Replicate 10% of data in all nodes
- 50 nodes, each with 2 CPUs
  \[ \Rightarrow 490\% \text{ overhead} \]
- 4 nodes, each with 25 CPUs
  \[ \Rightarrow 30\% \text{ overhead} \]

Does migration/replication help?
NAS parallel Benchmark Study (Execution time in seconds) [M. Bull, EPCC 2002]

WildFire’s Technology Limits

SRAM size = DRAM size / 256
Snoop frequency

Does migration/replication help?
NAS parallel Benchmark Study (Execution time in seconds) [M. Bull, EPCC 2002]

WildFire’s Technology Limits

SRAM size = DRAM size / 256
Snoop frequency

Sun’s SunFire 15k
Erik Hagersten
Uppsala University
Sweden

StarCat
Sun Fire 15k
StarCat, 72 CPUs

Active Backplane

18x18 addr X-bar

18x18 addr X-bar

StarCat Coherence Mechanism

Active Backplane

18x18 addr X-bar

18x18 addr X-bar

Allocate Dir$ entry only for write requests. Speculate on clean data on Dir$ miss

Allocate Dir$ entry only for write requests. Speculate on clean data on Dir$ miss

StarCat Performance Data

Active Backplane

18x18 addr X-bar

18x18 addr X-bar

WildCat coherence w/o CMR & w/ faster interconnect

Up to 104 CPU (trading for I/O)