

Overview

- on-chip communication networks
 - systems-on-chip
 - bus-based networks: arbitration, transfer modes (cache coherency)
 - mesh-based networks: hierarchies, message organization & routing
 - issues: signal propagation delay, energy per bit transfers, reliability
- cache coherency reconsidered and a case for the message passing paradigm
- intermission
- the Intel Single-Chip Cloud Computer
 - motivations: high performance & power efficient network
 - architecture: memory hierarchy, power management
 - programming model: the Rock Creek Communication Environment
 - ◆ motivations; overview; transferring data between cores
 - ◆ programming examples; performance
- refs: [1] *On-chip Communication Architectures*, Pasricha & Dutt, 2010, Morgan-Kaufman

[2] *The Future of Microprocessors*, Borkar & Chien, CACM, May 2011

Systems on a Chip

- former systems can now be integrated into a single chip
- usually for special-purpose systems
- high speed per price, power
- often have hierarchical networks

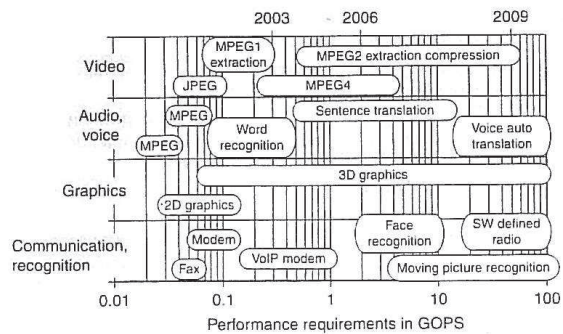


FIGURE 1.5 Increasing performance requirements for emerging applications [3,4]

(courtesy [1])

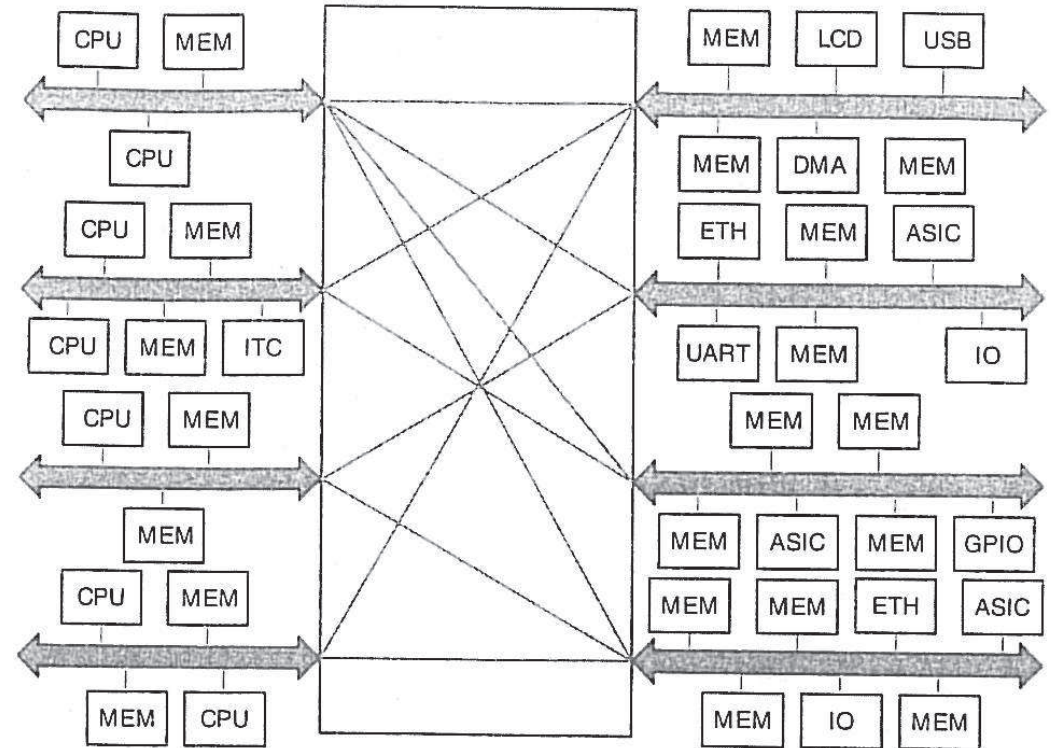


FIGURE 1.4 (d) bus matrix (or crossbar bus)

(courtesy [1])

On-chip Networks: Bus-based

- traditional model; buses have address and data pathways
- can be several 'masters' (devices that operate the bus, e.g. CPU, DMA engine)
- in a multicore context, there be many! (scalability issue)
- hence arbitration is a complex issue (and takes time!)
- techniques for improving bus utilization:
 - burst transfer mode: multiple requests (in regular pattern) once granted access
 - pipelining: place next address as data from previous request placed on bus

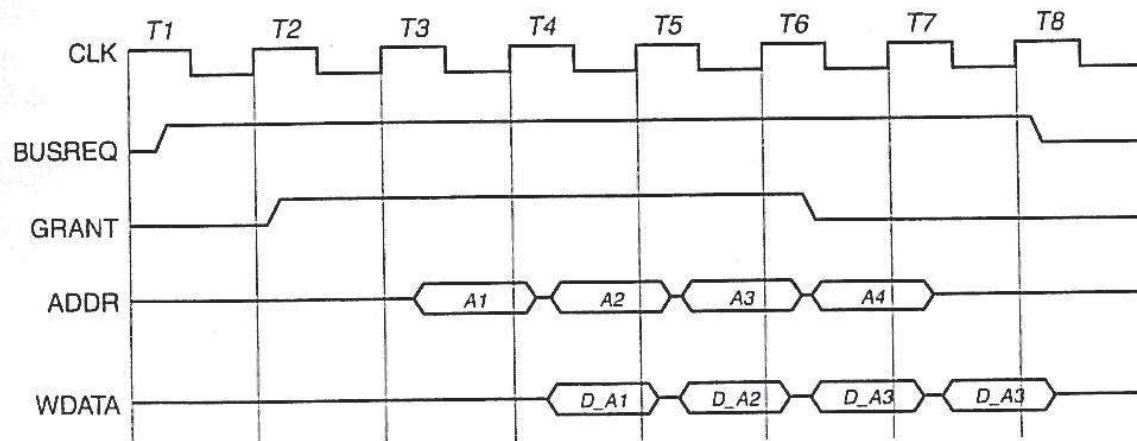
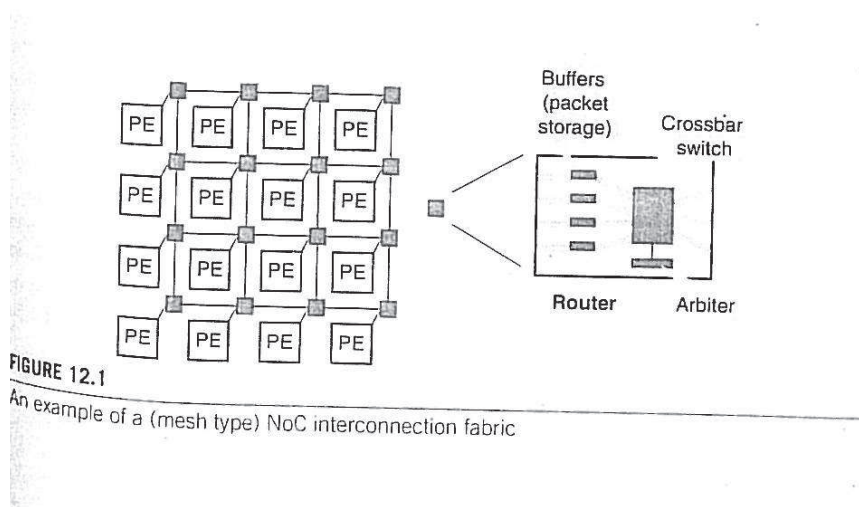


FIGURE 2.8(b)

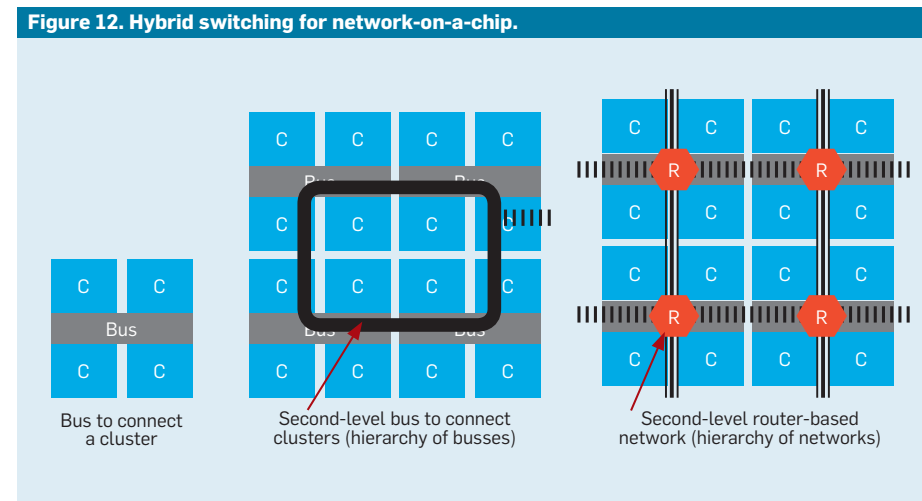
- broadcast: one master sends to others
 - e.g. cache coherency - snoop, invalidation

On-Chip Networks: Current and Next Generations

- buses: only one device can access (make a 'transaction') at one time
- crossbars: devices split in 2 groups of size p
 - can have p transactions at once, provided at most 1 per device
 - e.g. T2: $p = 9$ cores and L2\$ banks (OS Slide Cast Ch 5, p86); Fermi GPU: $p = 14$?
 - for largish p , need to be internally organized as a series of switches
- may be also organized as a ring (e.g. Cell Broadband Engine)
- for larger p , may need a more scalable topology such as a 2-D mesh



(courtesy [1])



(hierarchical networks, courtesy [2])

On-chip Network Mechanisms: Protocols and Routing

- unit of communication is a message, of limited length
 - e.g. T2 L1\$ cache line invalidation (writeback) has a 'payload' of 8 (8+16) bytes
- generally, routers break messages into fixed-size packets
- each packet is routed using circuit-based switching (connection established between source and destination – enables 'pipelined' transmission)
- unit of transmission is a flit, with a phit being transferred each cycle
 - routing: the destination's id in the header flit is used to determine the path

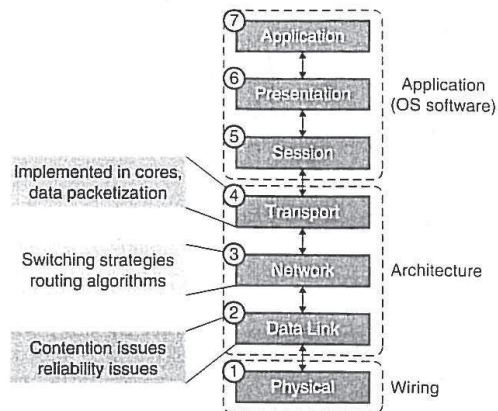


FIGURE 12.2

NoCs in the context of the 7 layer ISO/OSI protocol stack model

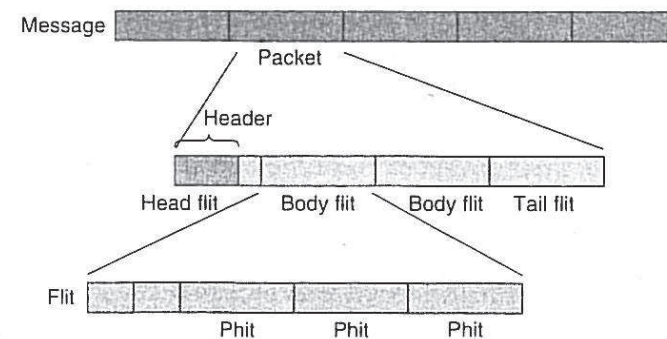
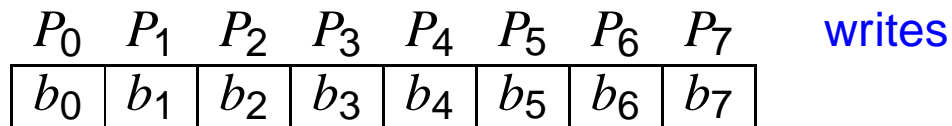


FIGURE 12.7

Structure of messages, packets, flits, and phits

Cache Coherency Considered Harmful (the 'Coherency Wall')

- a core writes data at address x in its L1\$; must invalidate x in other L1\$s
- standard protocols requires a broadcast message for every cache line invalidation
 - maintaining (MOESI) protocol also requires a broadcast on every miss
 - energy cost of each is $O(p)$; overall cost is $O(p^2)$!
 - also causes contention (hence delay) in the network (worse than $O(p^2)$?)
- directory-based protocols can direct invalidation messages to only the caches holding the same data
 - far more scalable (e.g. SGI Altix large-scale SMP), for lightly-shared data
 - worse otherwise; also introduces overhead through indirection
 - also for each cached line, need a bit vector of length p : $O(p^2)$ storage cost
- false sharing in any case results wasted traffic



● hey, what about GPUs?

- atomic instructions, synchronizing the memory system down to the L2\$, have (large) $O(p)$ energy cost each (hence $O(p^2)$ total!)
- cache line size is sub-optimal for messages on on-chip networks

Shared Memory Programming Paradigm Challenged

- ref: [3] Kumar et al, *The Case For Message Passing On Many-Core Chips*
- traditionally: paradigm to use when hardware shared memory is available
 - faster and easier!
- challenged in late 90's: message passing code faster for large p on large-scale SGI NUMA shared memory machines (Blackford & Whaley, 98)
 - message passing: separate processes run on each CPU, communicating via some message passing library
 - ◆ note: default paradigm for Xe cluster, 2×4 cores per node
 - reason: better separation of the hardware-shared memory in different processes
 - confirmed by numerous similar other studies since especially if fast on-board/on-chip communication network communication channels available
- is it easier? (note: message passing can also be used for synchronization)
 - SM is a “difficult programming model” [3] (relaxed consistency memory models)
 - data races, non-determinism; more effort to optimize in the end [5, p6]
 - no safety / composability / modularity

SCC Architecture

- chip layout [5, p24], tile layout [5, p25]
- overall organization [5, p27]
- router architecture [5, p28] (VC = virtual channel)
 - uses wormhole routing (Dally 86)
- what the SCC looks like: test board [5, p30]; system overview [5, p31]
 - the 4 memory controllers (and memory banks; c.f. T2, Fermi) are part of the network
- memory hierarchy: programmer's view [5, p35]
 - globally accessible per-core test-and-set registers provides time and power efficient synchronization
- power management:
 - voltage / frequency islands (variable V and f) [5, p47]
 - slightly super-linear relationship: chip voltage & power [5, p33]
 - power breakdown: in the cores and routers [5, p34]; note: cores 87W → 5W

Programming Model: the Rock Creek Communication Environment

- research goals [5, p36]; high level view [5, p38], SPMD execution model
- RCCE software architecture for 3 platforms [5, p37]
- message passing in RCCE [5, p38-41], using the message passing buffer (fast shared memory)
 - shared cache memory has MPBT flag, in page table (propagated down to the L1\$ on a miss)
 - note limited size of shared memory area!
- issues in lack of coherency, when performing [5, p39]:
 - a `put()`: must invalidate in L1\$ before write
 - a `get()`, must ensure stale data is not in the L1\$ (why is MPB cached at all?)
- special `A = RCCE_malloc(sizeA)` for (MP) shared memory [5, p40] (c.f. CUDA)
 - must be called by all participating processes, so all will be able to access the same address offset (**A**)
- send and receive implemented in terms of `put()` & `get()` (copy to/from MPB (shared) memory) [5, p42]

Example RCCE Program: ringsend.c on the SCC

- *comm. is synchronous*: senders must wait till the corresp. receiver does the `recv()`

```
#include "RCCE.h"
int RCCE_APP(int argc, char *argv[]) {
    int id, p, reps = 2, leftId, rightId, token, tmpToken;
    RCC_Init(&argc, &argv); p = RCCE_num_ues(); id = RCCE_ue();
    printf("%d: I am process %d of %d\n", id, id, p);
    token = id, leftId = (p+id-1)%p, rightId = (id+1)%p;
    for (int i=0; i < reps; i++) {
        if (id % 2 == 0) { // pairs of processes must sync
            RCCE_send(&token, sizeof(int), rightId); // even to odd
            RCCE_recv(&tmpToken, sizeof(int), leftId); // odd to even
        } else {
            RCCE_recv(&tmpToken, sizeof(int), leftId); // even to odd
            RCCE_send(&token, sizeof(int), rightId); // odd to even
        }
        token = tmpToken;
    } //for(i...)
    printf("%d: after %d reps, I have token from process %d\n",
           id, reps, token);
} // RCC_APP()
```

- run by `rccerun -nue 4 ringsend 1`; (similar program [5, p68-9])

