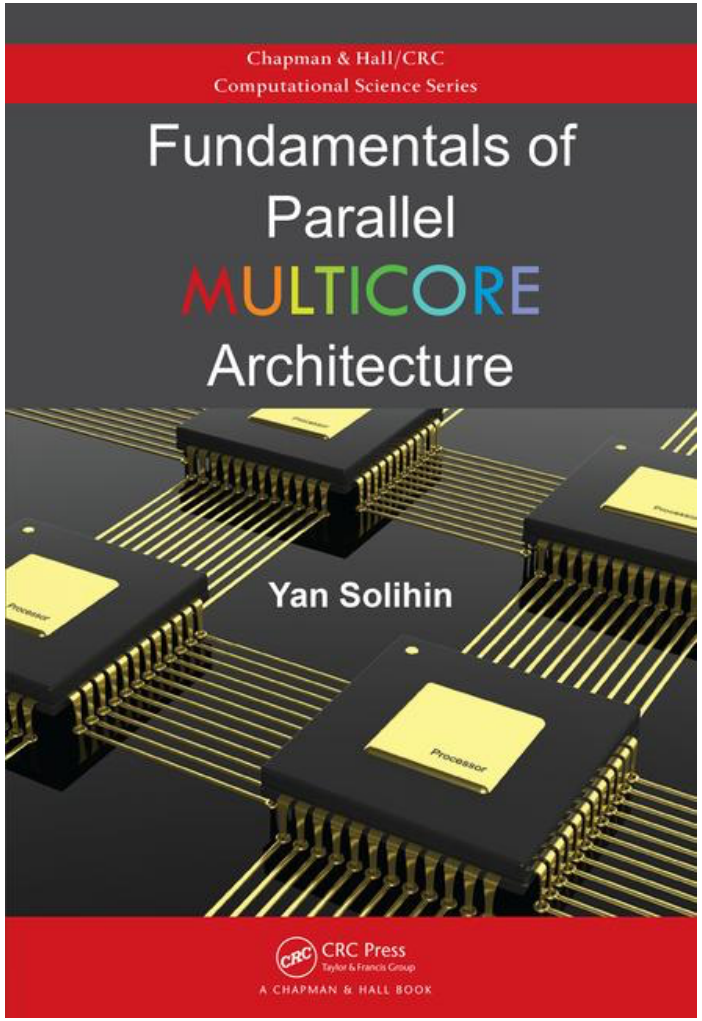
When Moving Data is Expensive: What We can Learn from High Gas Prices

Yan Solihin
Professor, ECE, NCSU

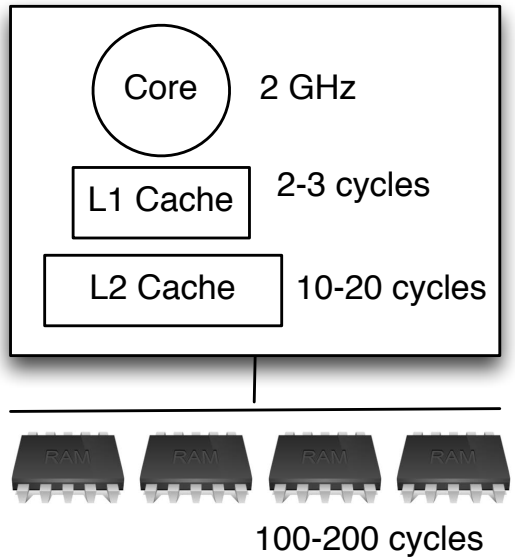
Program Director
Computer Systems Research (CSR)
Secure and Trustworthy Cyberspace (SaTC)
Scalable Parallelism in the eXtreme (SPX)
NSF

Published December 2015



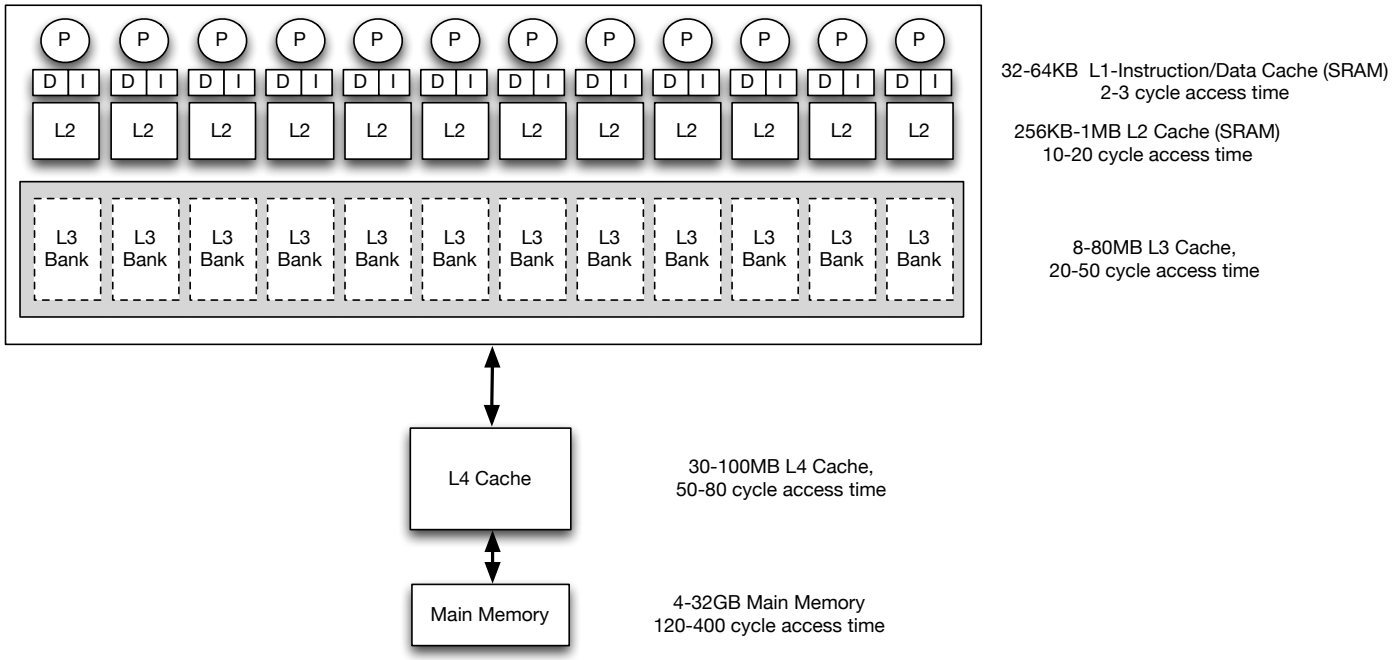
Let's Revisit CPU-Memory Gap

- 1999
  - CPU speed grows at 55%/year
  - Memory speed grows at 7%/year
  - Processing in Memory research born
    - Put simple processor in DRAM chips
    - Berkeley IRAM, Illinois FlexRAM, etc.
  - No industry adoption
    - Caches are good enough
    - DRAM chip has no budget for processing



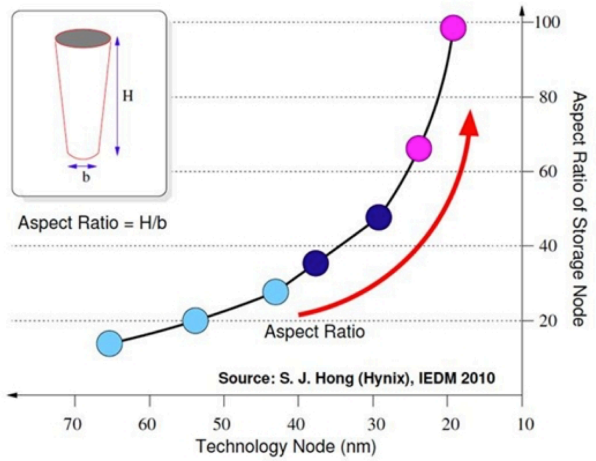
14 Years Later (2013)

- Processor speed growth stalled around 2005-2007
- Multicore design proliferated
- Cache hierarchy got deeper (for scaling purpose)



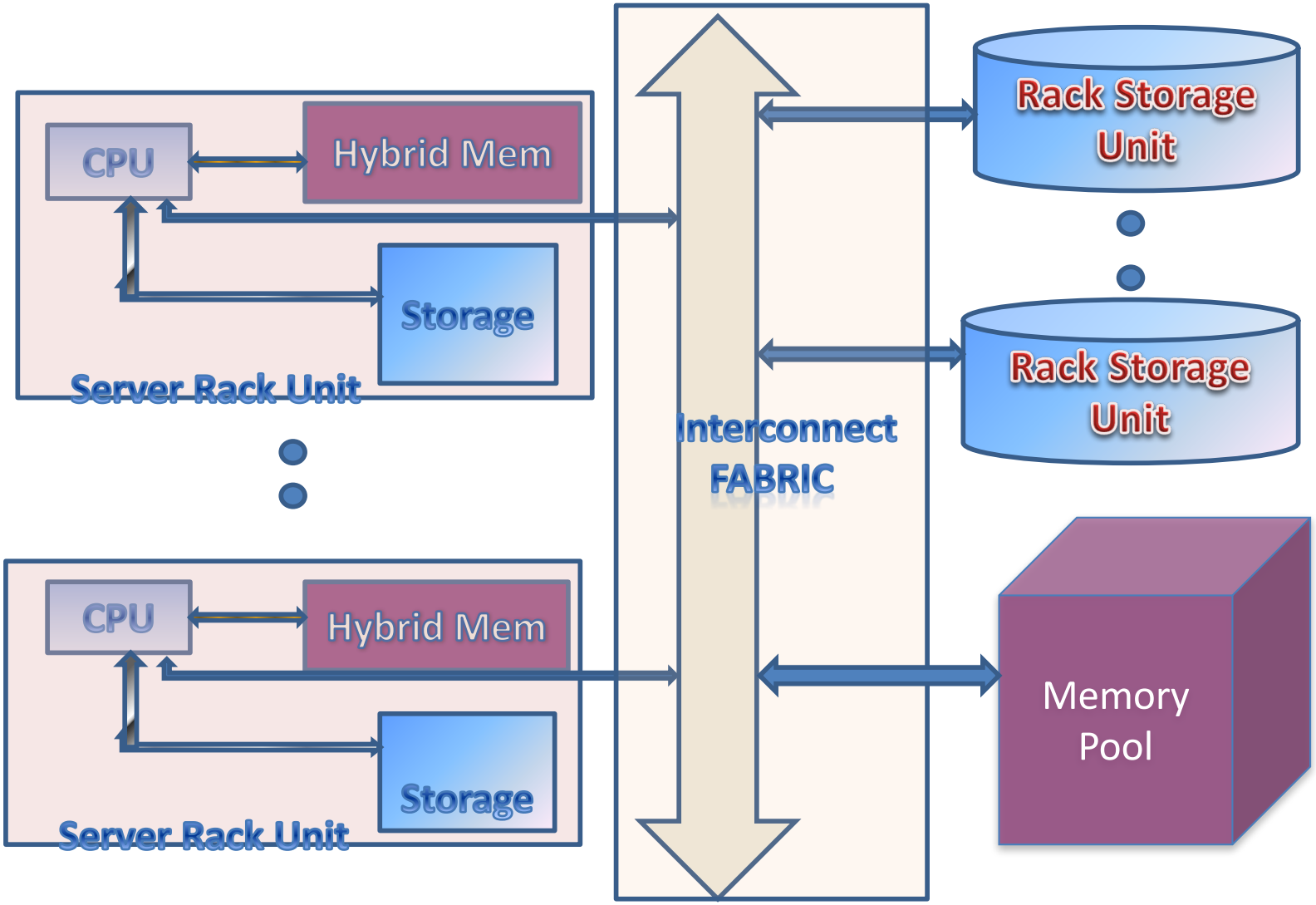
What's in Store in 2027?

- Yan's crystal ball
- Trend 1: Increasing demand for memory
  - In-memory databases, key-value stores, data analytics
  - Servers have limited scope for memory expansion
    - Limited by the number of memory ports on a rack unit
    - Limited by DRAM scaling => must rely on NVM

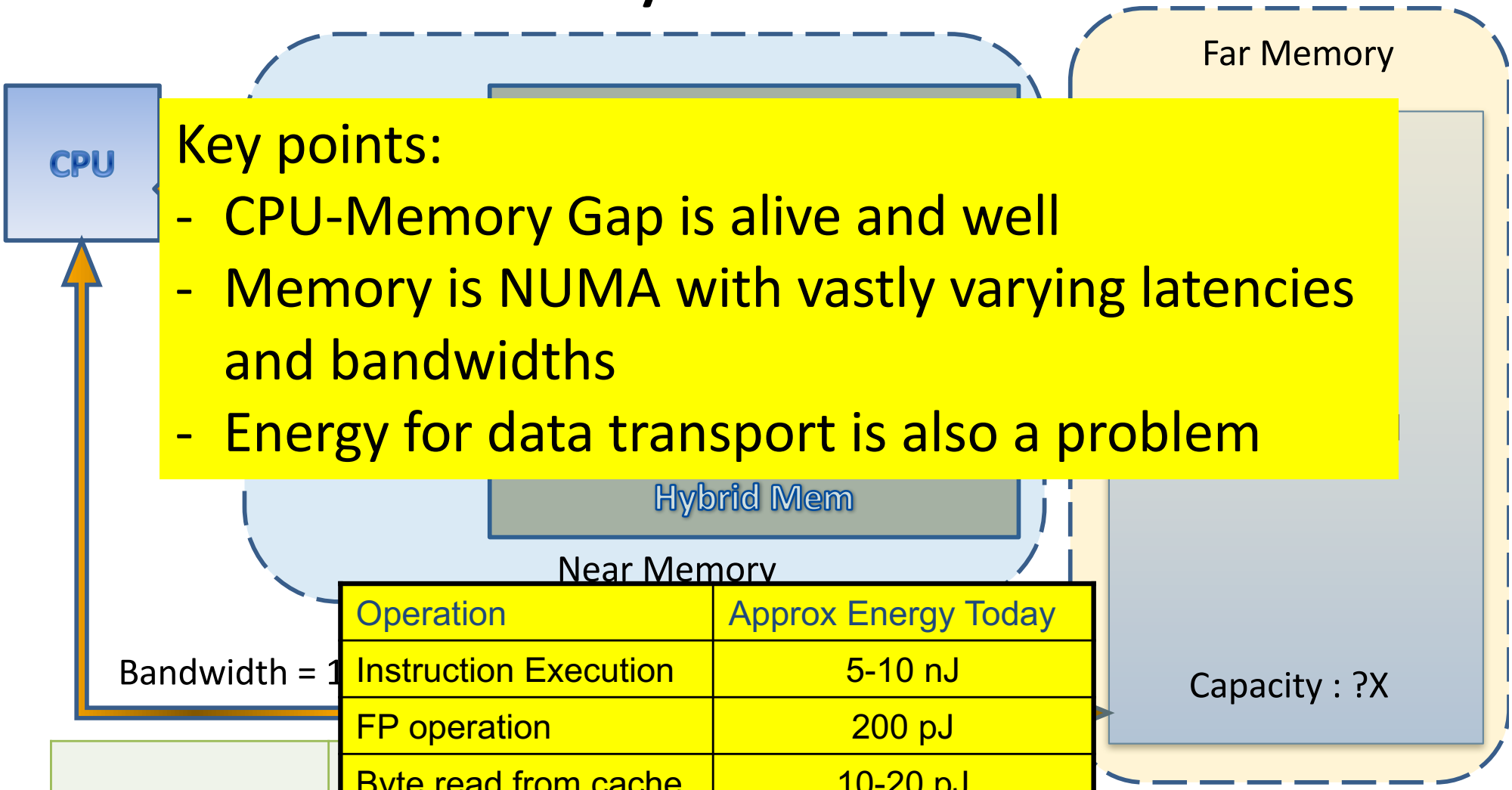


- Trend 2: Resource disaggregation
  - Servers have poor resource utilization
  - Interconnection has become faster
  - Expand memory with external memory pools
    - How connected? Interconnect becomes the bottleneck
      - CCIX , GenZ, PCIe with MMIO
    - Silicon Photonics?
- Trend 1 and 2 point to rack-scale server architectures

Rack-Scale Server



Memory Architecture



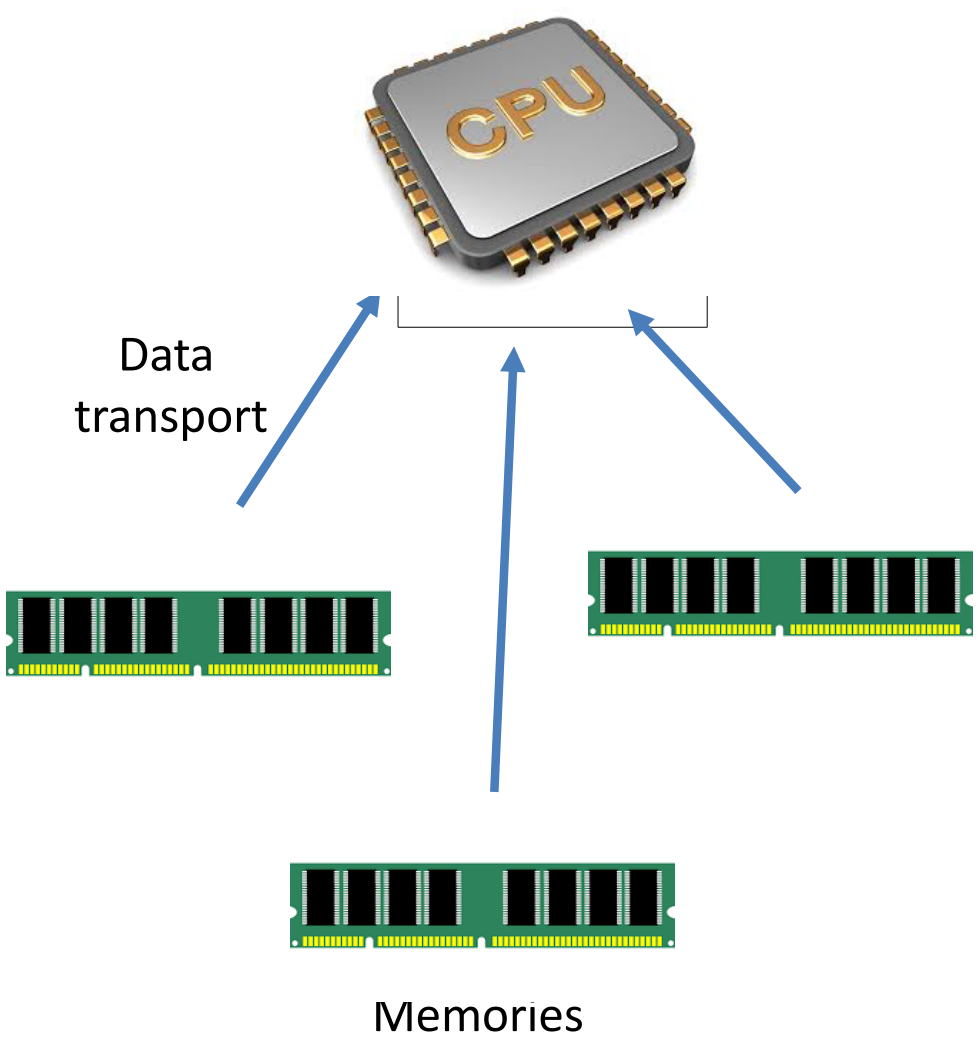
Key points:

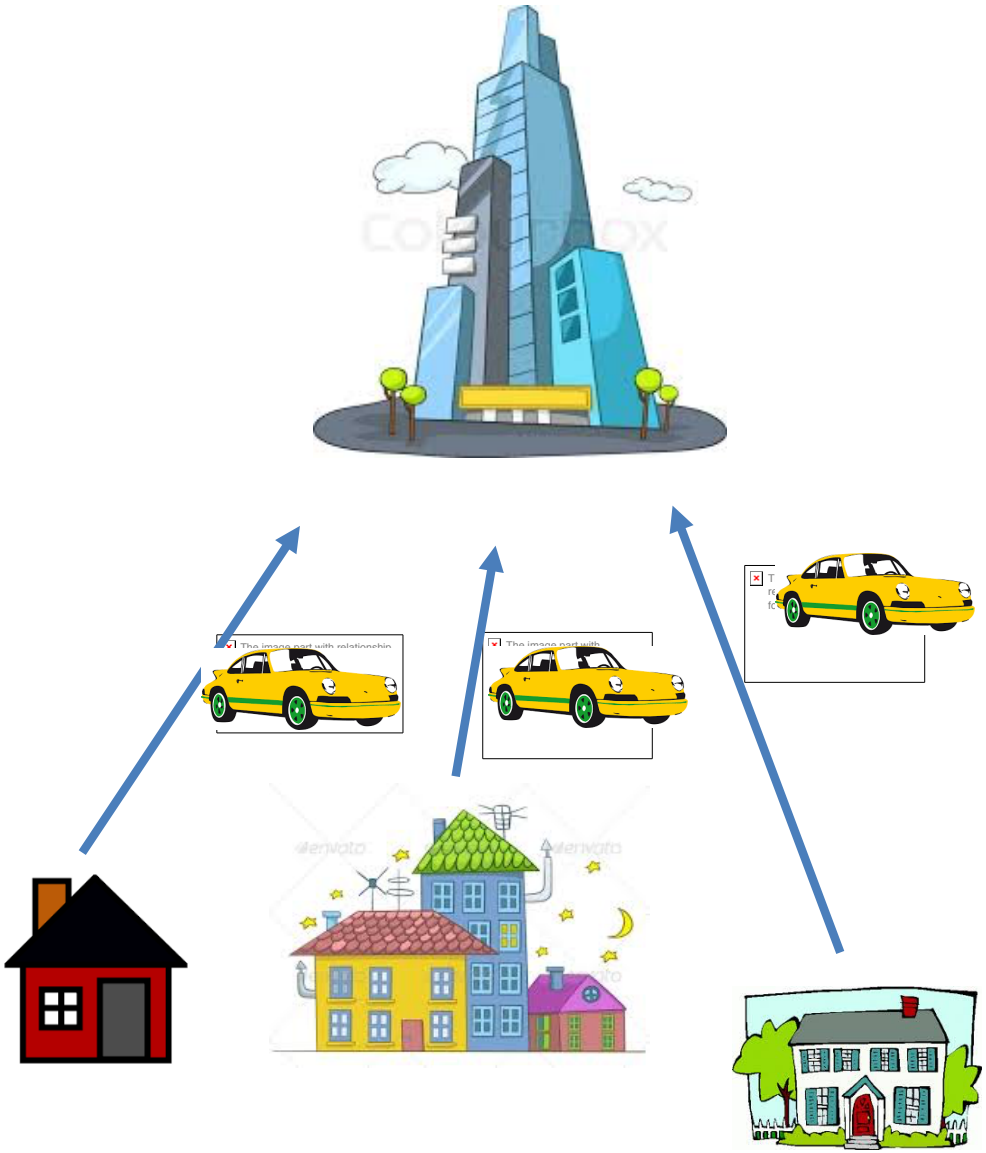
- CPU-Memory Gap is alive and well
- Memory is NUMA with vastly varying latencies and bandwidths
- Energy for data transport is also a problem

Operation	Approx Energy Today
Instruction Execution	5-10 nJ
FP operation	200 pJ
Byte read from cache	10-20 pJ
Byte read from DRAM	1.5 nJ
Byte over IC fabric	5 pJ/hop—250 pJ+

Latency		
Bandwidth		
Capacity	X	Source: Borkar, PACT11 keynote

Data vs. People





Reflecting back to 2006-2008

- Gas price reached > $4/gal => people movement became expensive
- How did people react?
  - Live closer to work
  - Denser city zoning laws
  - Teleworking
  - Carpooling
  - They bought gas hybrid cars
- What can we learn from them?

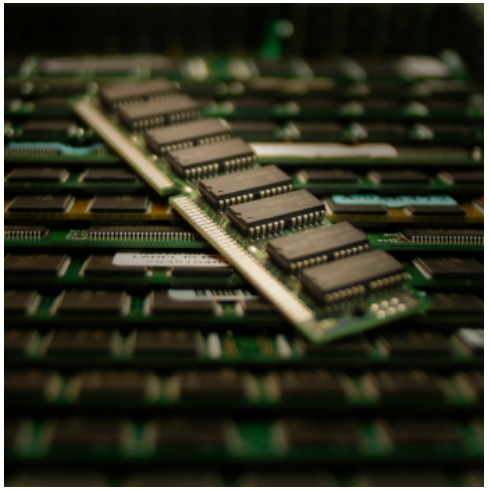
Lessons Learned

Strategy	Computer Architecture Equivalent
Live closer to work	~~Locality optimization~~ Not covered today
Denser city zoning laws	Replace DRAM/SRAM with NVM
Teleworking	Processing in Memory
Carpooling	Bulk data transfer
Hybrid and electric cars	~~Wireless interconnect and silicon~~ photonics Not covered today

Non-Volatile Main Memory

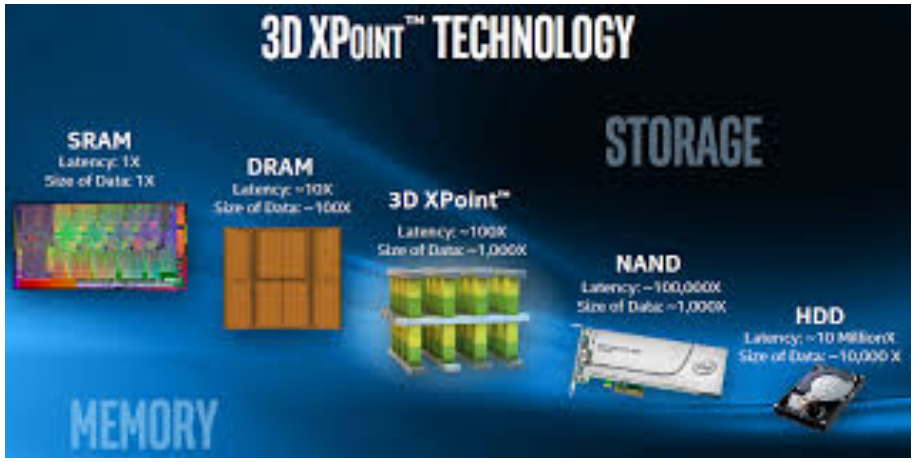
Non-Volatile Memory (NVM)

- NVMs are emerging:
  - Phase-Change Memory (PCM)
  - Memristor
  - ReRAM
  - STT-RAM
  - 3D Xpoint
- Use as storage?
  - Requires fast interface
    - NVMe
    - DDRx
  - Requires reworking of software stack
- Use as memory?
  - Enables persistent memory



Source: http://www.techweekeurope.co.uk/

NVM and Storage Class Memory



Intel 3D Xpoint (Optane):

- 20nm process
- SLC (1 bit/cell)
- 7 microsec latency
- 78,500 (70:30 random) read/write IOPS
- NVMe interface
- 375GB – 1.5TB

Source: Internet

Why Persistent Memory?

- Suppose important data is in a linked list
- Every so often, write data in linked list to a file

Without persistent memory

```
f = fopen(...);
p = Head;
while (p != NULL) {
  fprintf(f, "%d\n", p->data);
  p = p->next;
}
fclose(f);
```

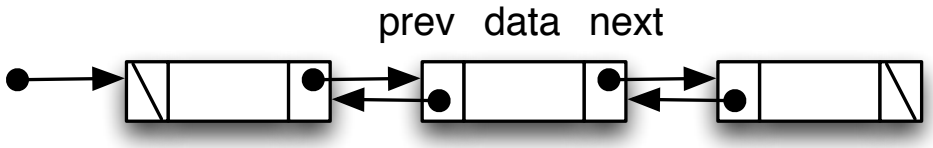
- Expensive, and does not utilize NVMM

With Persistent Memory

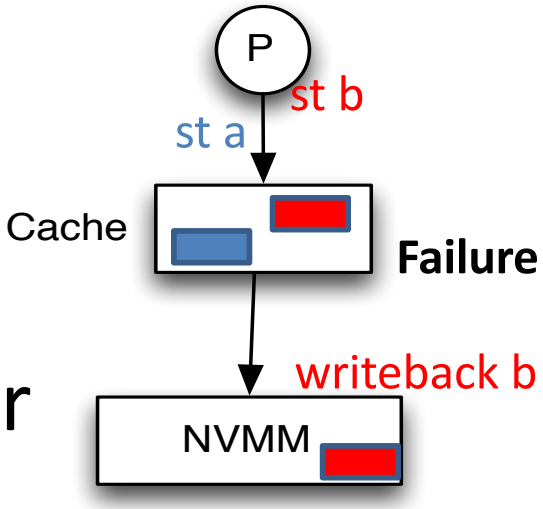
- Skip file operations, keep data in memory
- "Most" data already "durable"
  - Linked list data may be in NVMM (durable) or in caches (not durable yet)
  - On a failure (e.g. power failure, software crash), is the linked list in a consistent state?
- **Persistency** requires reasoning about **failure recovery**, which requires:
  1. Durability ordering
  2. Atomic durability

Durability Ordering

- Example linked list

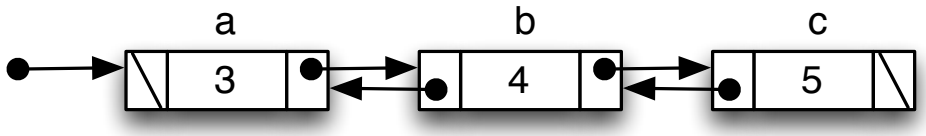


- Suppose we want to perform ops in this order
  - Change a->data to 6
  - Change b->data to 8
- Linked list is inconsistent
- Durability order != program order



Durable Atomicity

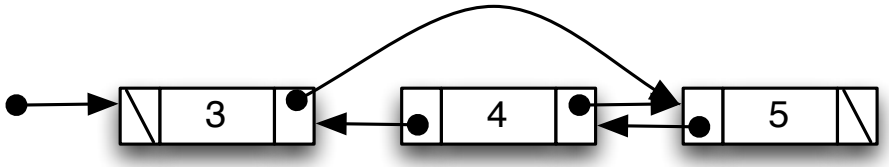
- Suppose we would like to delete Node 4




```
p->prev->next = p->next;
p->next->prev = p->prev;
```

- If failure occurs in the middle, linked list not recoverable!

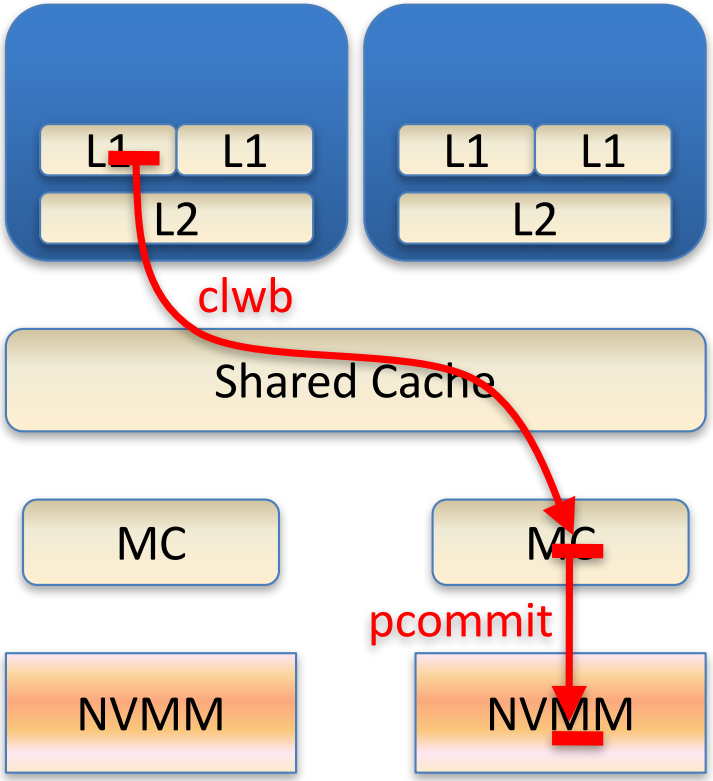
```
p->prev->next = p->next;
```
failure occurs here!

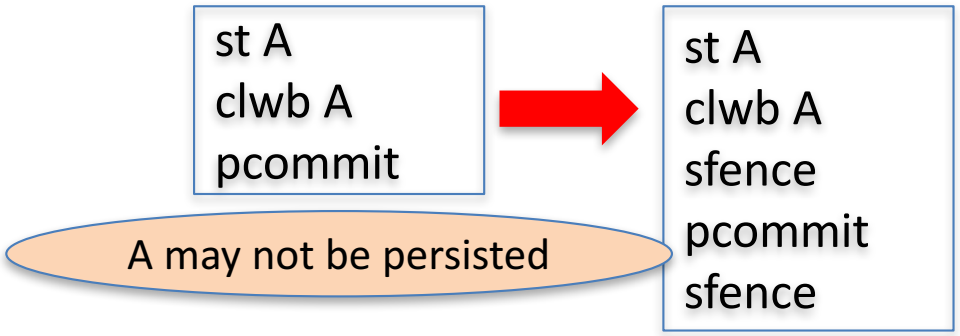


  - Traversing from left to right, Node 4 is missing
  - Traversing from right to left, Node 4 is found

Intel PMEM

- Instructions to implement durable barrier
  - Clflush/clflushopt: clean a dirty block from caches to MC
  - Clwb: write back (i.e. evict) a dirty block
  - Pcommit: commit write from MC to NVMM*





*pcommit has been deprecated

Achieving Failure Safety

- PMEM provides durable barriers, but atomic durability is programmer's responsibility
- Programmers can transactionalize their SW:

Step 1	Perform undo-logging. Make the undo-log updates durable.
Step 2	Logged_bit is set and made durable, indicating a transaction has begun.
Step 3	Commit updates to the memory and make them durable.
Step 4	Logged_bit is unset and made durable, indicating the transaction is complete.

Example

- Create undo log
  - Barrier
- Set logged bit
  - Barrier
- Make changes
  - Barrier
- Unset logged bit
  - Barrier

```
01  log_index =0;
02  log_op = OP_INSERT;
03  memcpy(&LOG[log_index], nn, sizeof(struct node));
04  LOG_ptr[log_index] = nn;
05
06  SFENCE;
07  CLWB(&LOG[log_index]);
08  CLWB(&LOG_ptr[log_index]);
09  log_index++;
10  CLWB((void*)&log_index);
11  CLWB((void*)&log_op);
12  SFENCE;
13  PCOMMIT;
14  SFENCE;
15
16  logged = 1;
17
18  SFENCE;
19  CLWB ((void*)&logged);
20  SFENCE;
21  PCOMMIT;
22  SFENCE;
23
24  temp=(struct node *)malloc(sizeof(struct node));
25  temp->node_key=num;
26  memcpy(temp->data, val, DATASIZE);
27  nn->next=temp;
28  nn=temp;
29  nn->next=right;
30
31  SFENCE;
32  CLWB(nn);
33  CLWB(temp);
34  SFENCE;
35  PCOMMIT;
36  SFENCE;
37
38  logged = 0;
39
40  SFENCE;
41  CLWB((void*)&logged);
42  SFENCE;
43  PCOMMIT;
44  SFENCE;
```

Undo logging

Set logged_bit

Transaction

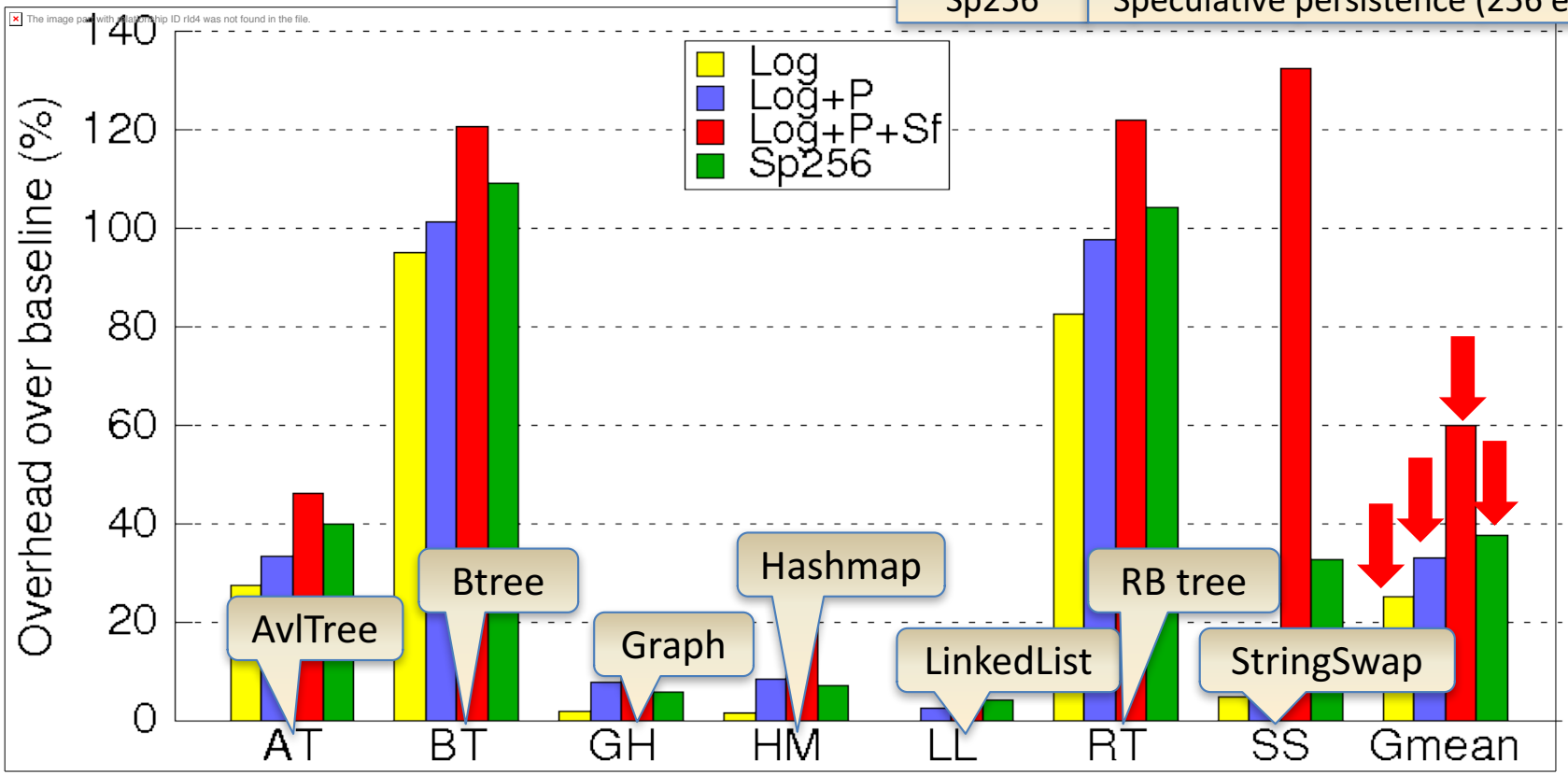
Unset logged_bit

Speculative Persistence [ISCA'17]

- Observation: processor frequently stalls at pcommit
- Solution: execute speculatively past pcommit
  - Mis-speculation is rare: only when failure occurs
  - Applies to sequential program also
  - Principles
    - Apply speculation aggressively
    - Make correct speculation fast
    - Recovery time is not very important

Evaluation - O

Log	Benchmark + undo logging
Log+P	Log + PMEM instructions
Log+P+Sf	Log + P + sfence (fail-safe version)
Sp256	Speculative persistence (256 entries)



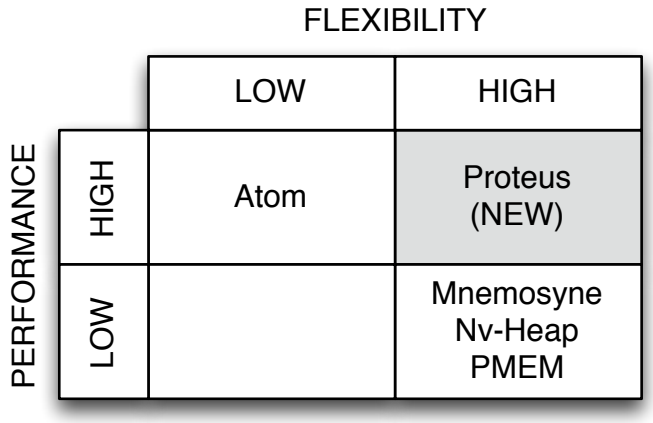
- Baseline = original benchmark without logging or persistence
- Adding sfence nearly doubles the overheads (33% vs. 60%)
- SP reduces the overheads from 60% to 38% (Log+P+Sf)

Observations

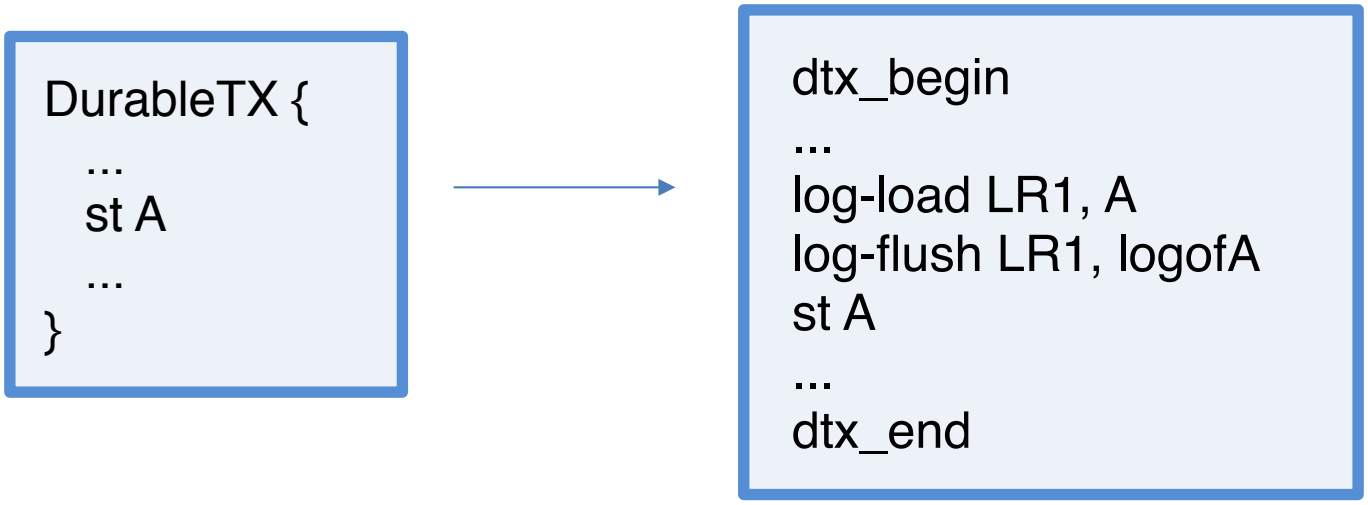
- SP only removes pipeline stall overheads
- Logging code overheads remain
- Can we perform logging in hardware?
  - No extra code
  - Low execution time overhead
  - But, not flexible
    - It limits transaction count
    - It limits transaction size

Alternatives?

- ATOM [HPCA'17] introduced hardware logging
  - Automatically logs stores within a transaction
  - Hardware treats log updates and stores differently
- We propose Proteus [MICRO'17]
  - Software supported hardware logging

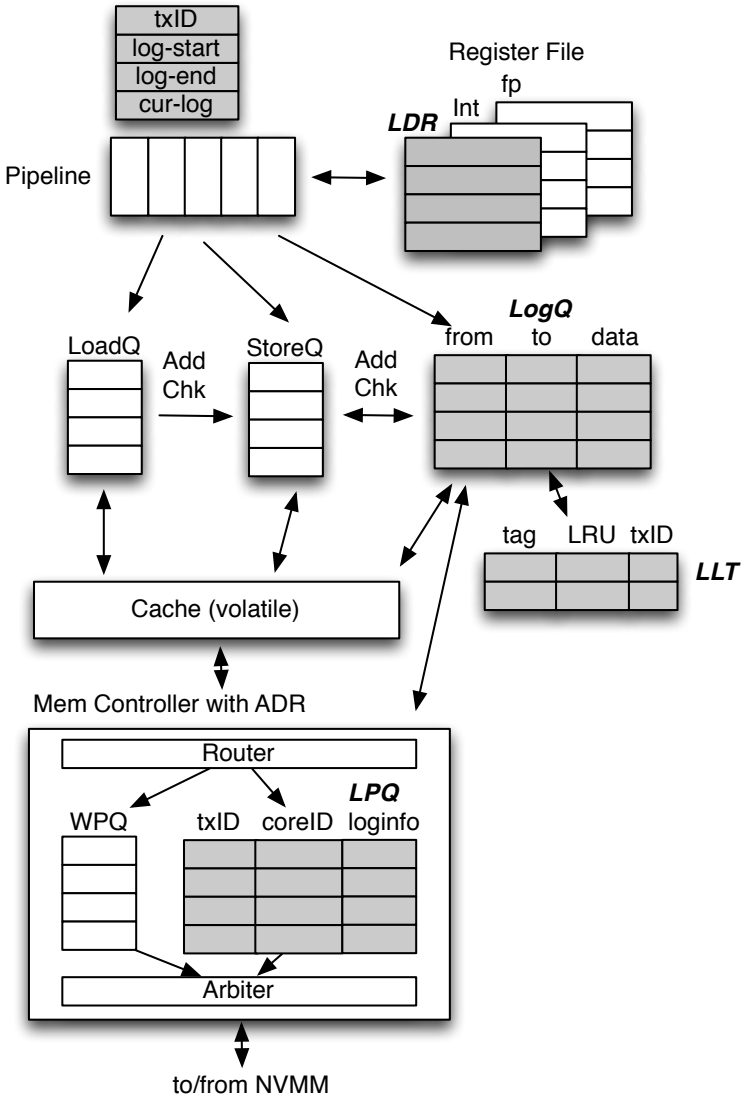


Our Solution



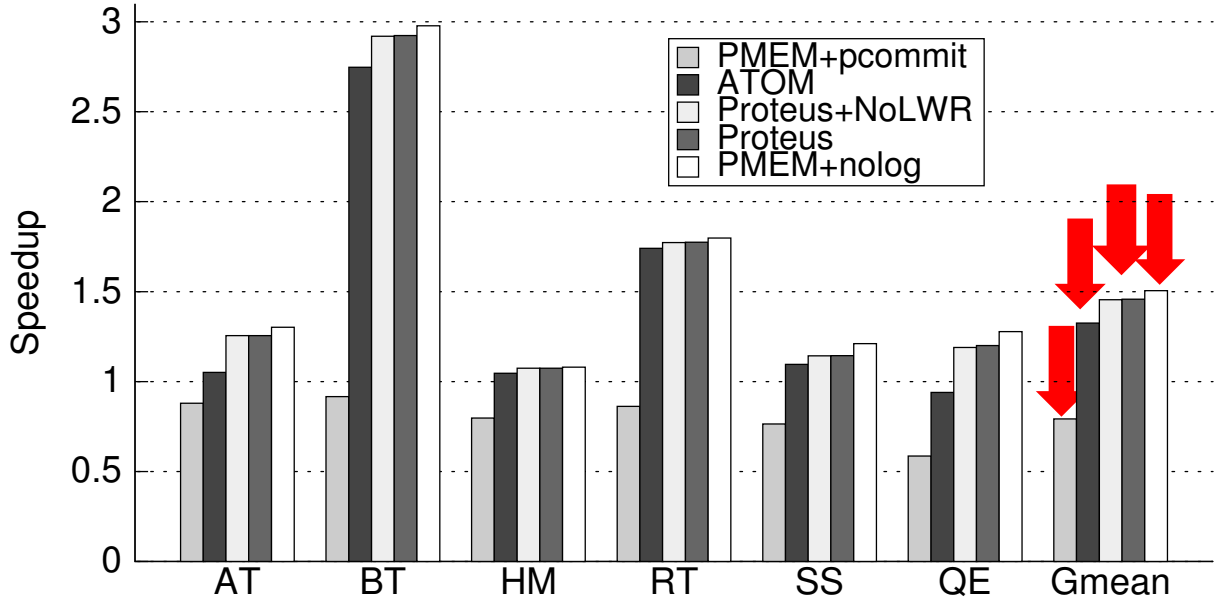
- Compiler replaces durable store with a pair of log-load and log-flush
  - Flexibility of software logging is maintained
  - Lower instruction overheads vs. software logging

Architecture



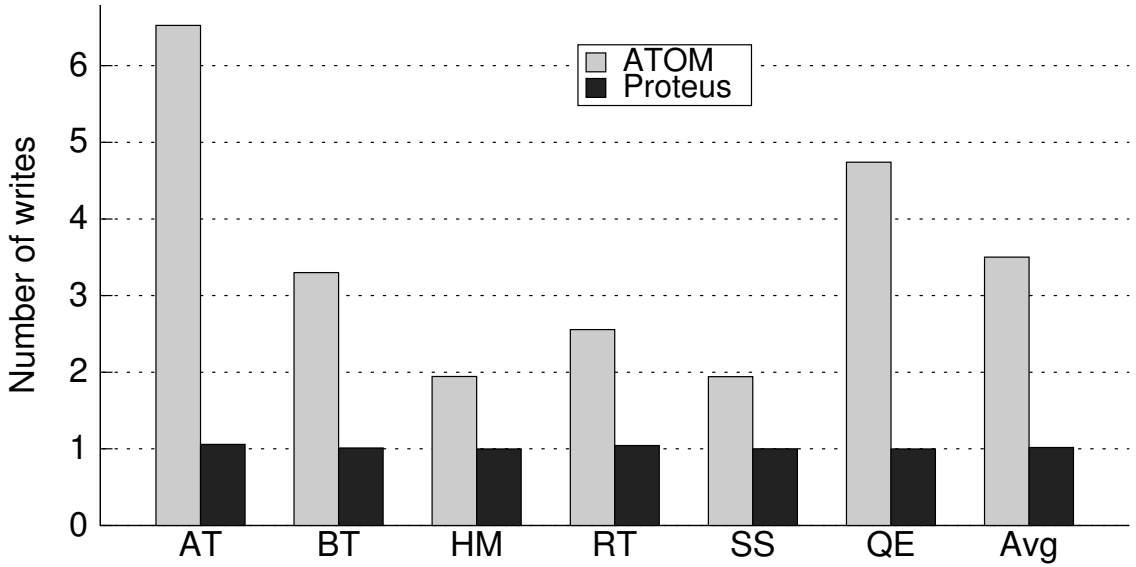
- 4 special registers keep track of log area in mem
- Log data registers (LDR) keep data being logged
- LogQ keeps non-persisted log-flush requests
- LLT is used for coherence lookup
- LPQ temporarily holds log writes, is in the non-volatile domain

Performance Results



- Deprecating pcommit helps (vs. sw logging)
- Proteus slightly faster than ATOM and is not far from an ideal case of no logging

Number of NVMM Writes vs. ATOM



- Much fewer writes => improved write endurance

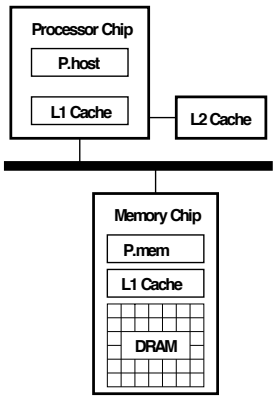
Processing in Memory (PIM)

Processing in Memory

- HPCA 2001

Automatically Mapping Code on an Intelligent Memory Architecture*

Jaejin Lee‡, **Yan Solihin**†§, and **Josep Torrellas**†
†University of Illinois at Urbana-Champaign
‡Michigan State University
§Los Alamos National Laboratory
http://iacoma.cs.uiuc.edu/flexram



Case	Original Loop	Partitioned Loop	
		P.host Code	P.mem Code
Fully Parallel	`DO I = 1, 100` `  B(I) = A(I)`	`DO I = 1, 70` `  B(I) = A(I)`	`DO I = 71, 100` `  B(I) = A(I)`

Distributable Without Synchronization	`DO I = 1, 100`
	`  A(I) = A(I-1)`
	`  C(I) = C(I+1)`
		`DO I = 1, 100`
		`  A(I) = A(I-1)`
			`DO I = 1, 100`
			`  C(I) = C(I+1)`

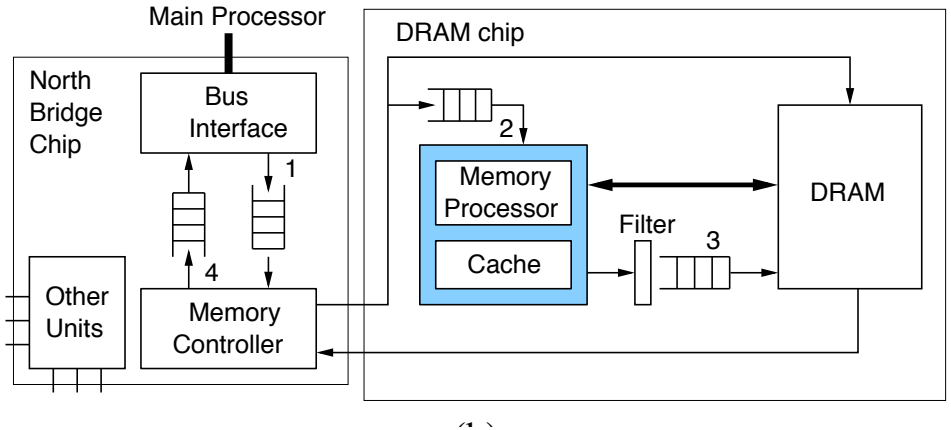
Distributable With Dopipe	`DO I = 1, 100`
	`  A(I) = A(I-1)+B(I)`
	`  C(I) = A(I)`
		`DO I = 1, 100`
		`  A(I) = A(I-1)+B(I)`
		`  IF (MOD(I,4).EQ.0) THEN`
		`  WRITEBACK(A(I-3) to A(I))`
		`  SIGNAL`
		`  ENDIF`
			`DO I = 1, 100`
			`  IF (MOD(I+3,4).EQ.0) THEN`
			`  WAIT`
			`  ENDIF`
			`  C(I) = A(I)`

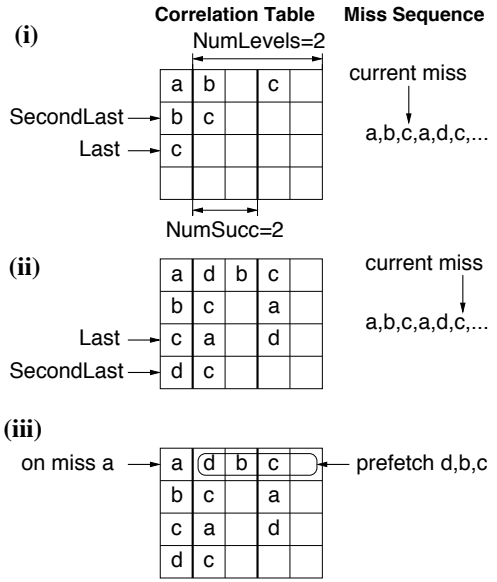
- ISCA 2002

Using a User-Level Memory Thread for Correlation Prefetching*

Yan Solihin † **Jaejin Lee** ‡ **Josep Torrellas** †

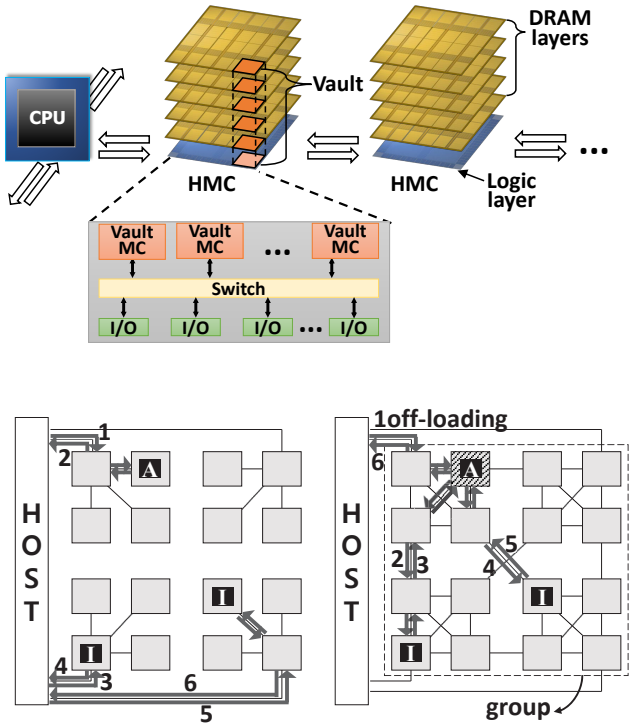
† University of Illinois at Urbana-Champaign
‡ Michigan State University
http://iacoma.cs.uiuc.edu
http://www.cse.msu.edu/∼jlee





More Recent PIM

- Hong et al. [PACT 2016]
  - Accelerating linked list traversal in memory
  - Memories are treated as distributed multiprocessors
  - Performance gain from fewer hops and utilization of high intra-memory bandwdith



(a) host-processing (b) NDP ($M = 1$)

Same Challenges as Before

- Gain from lower data access time > loss from slower computation
  - Data must be mostly local (not in other memory modules)
  - Data must be mostly uncached by host processor
  - NUMA computation model
- Coherence with other memory processors and with host processor
- Memory access using virtual address or physical address?
  - Additional complexity from MMU

Bulk Data Operations

Bulk Data Operations

- Bulk data copying and initialization (BCI)
  - Memory-to-memory bulk data transfers
    - Kernel libs: copy_from_user, memcpy, etc.
    - User libs: memcpy, memset, etc.
  - Cache-to-cache bulk transfers
  - Processor-to-accelerator bulk transfers
- Examples of memory-to-memory BCI
  - TCP/IP processing
    - Apache web server spends 20% time on BCI
  - File operations
  - Page initialization

Current M2M BCI Implementation

Two flavors

Explicit loop of loads & stores

PowerPC

```
loop:
    lwz r1, 0(r2)
    addi r2, r2, 32
    stw r1, 0(r3)
    addi r3, r3, 32
    bdnz loop
```

Copy instruction (expanded into **implicit loop** of loads & stores)

X86

```
mov esi, src
mov edi, dst
mov ecx, len
rep movsd
```

S/390

```
la r2, source_addr
la r3, source_len
la r4, dst_addr
la r5, dst_len
mvcl r2, r4
```

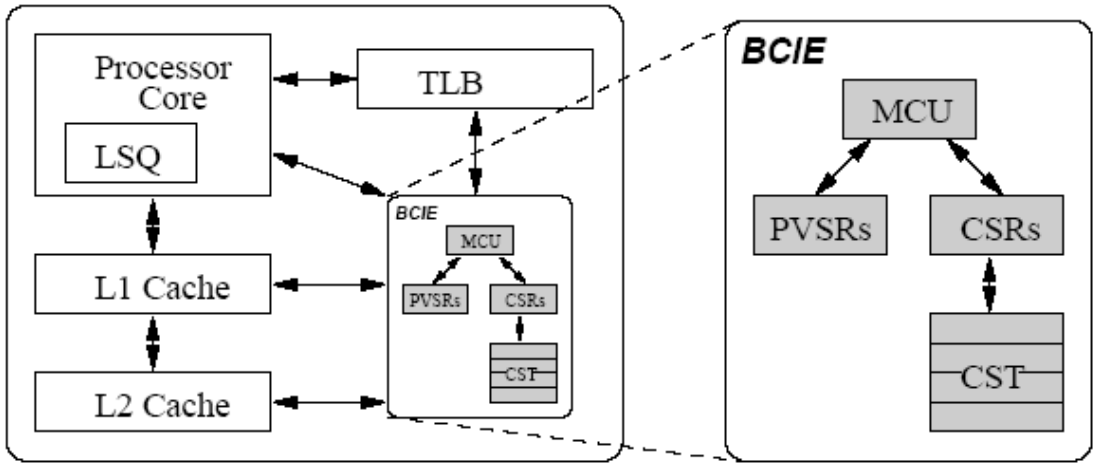
Current BCI Performs Poorly

- TCP/IP processing has become the major performance bottleneck of networking.

- Why so poorly?
  - Granularity inefficiency
  - Pipeline inefficiency
  - Cache affinity inflexibility

FastBCI [PACT09]

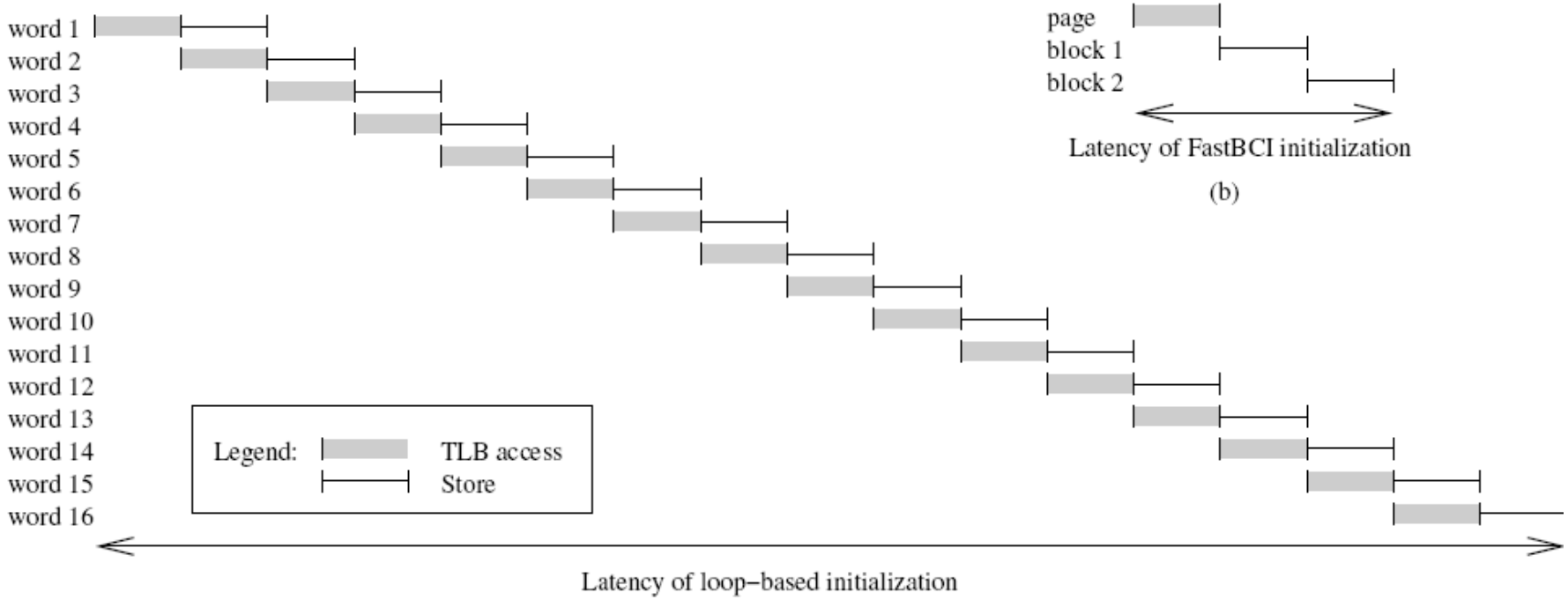
- An efficient architectural support for BCI
- New instruction: `BLKCPY Reg_SRC, Reg_DEST, Reg_PARAM`
  - Reg_SRC/Reg_DEST: specify src/dst base addresses
  - Reg_PARAM: specify size (4KB max) and cache affinity options
- On chip engine instead of implicit loop



FastBCI Benefits

- Granularity efficiency
  - Reduce 99.8%/87.5% TLB/cache accesses
- Pipeline efficiency
  - Early Commit and Non-blocking
- Cache affinity flexibility
  - Options: Cacheable, Non-cacheable and Cache Neutral
  - Cache neutral: No ***new*** copying data brought into cache
    - Tends to produce robust performance
- Roughly equal performance gains from all three. Total 2-3x faster

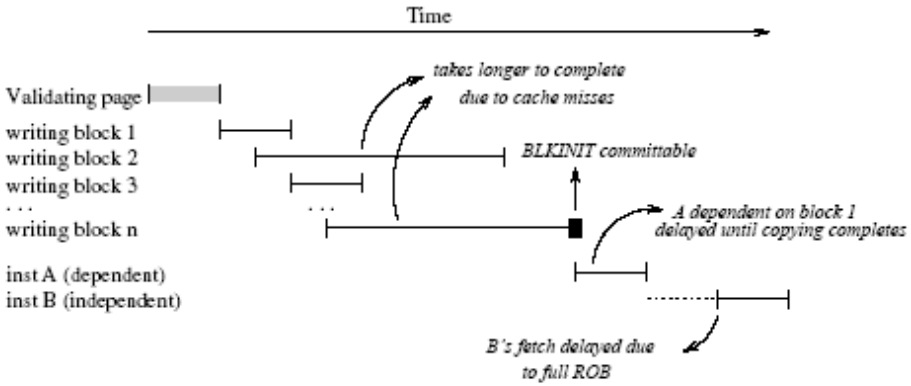
Granularity Efficiency



Increase TLB/Cache access granularity

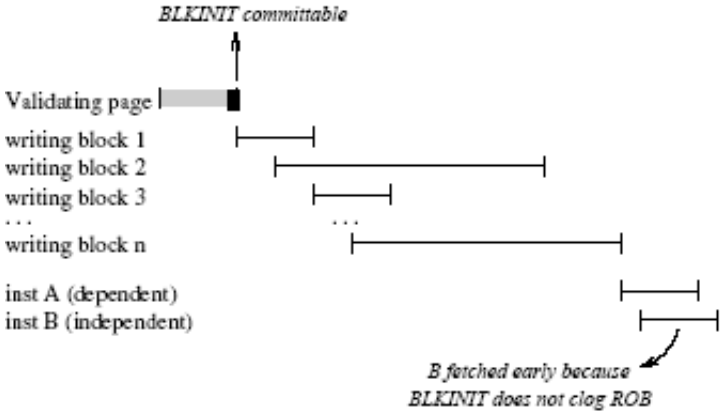
- TLB access at page granularity
  - Reduce 99.8% TLB accesses
- Cache access at cache-block granularity
  - Reduce 87.5% cache accesses

Pipeline Efficiency



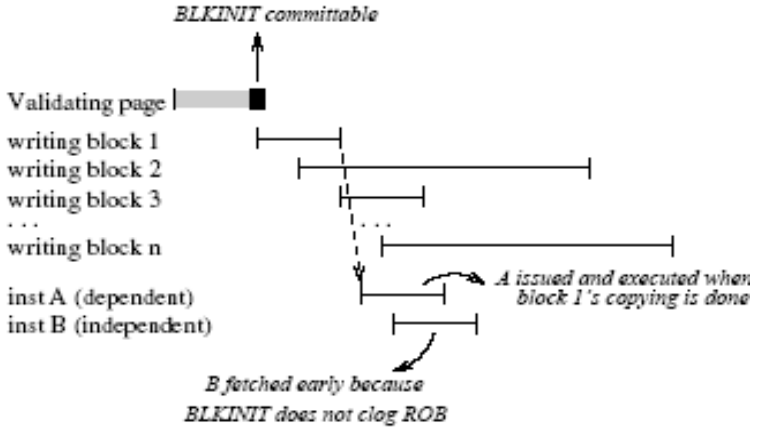
(a) FastBCI with blocking and no early commit

BCI takes long time, FastBCI instruction blocks ROB based instruction commit



(b) FastBCI with Early Commit

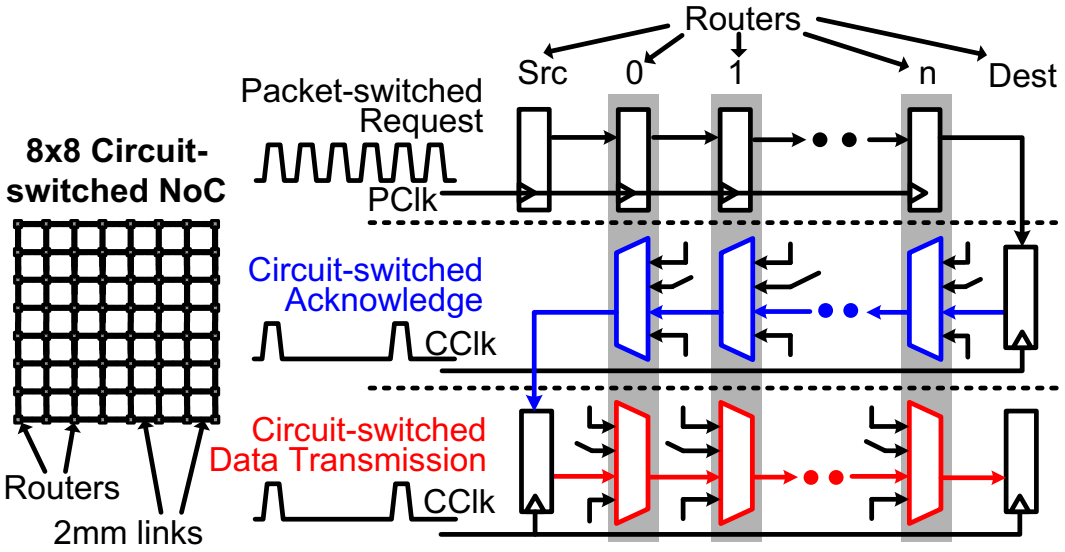
Non-blocking: copying progress is tracked per block, instructions dependent on completed blocks are committable.



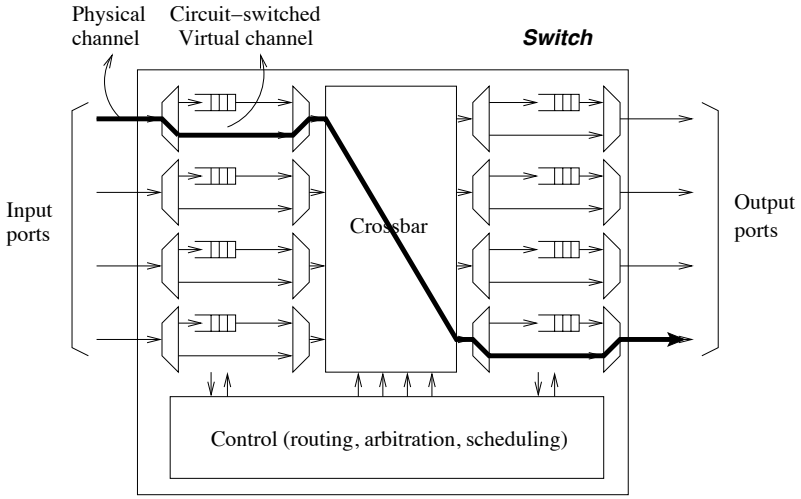
(c) FastBCI with Early Commit + Non-blocking

Early commit: FastBCI instruction commits once copying regions exception validation is completed

Another Avenue: Circuit Switching NoC



Source: Borkar, PACT 2011 keynote



Packet vs. Circuit Switching

- CS latency within 5% of PS latency with high hop count and/or large messages

Message size, distance (16B-link)	Uncontended Latency (number of clock cycles)		How much slower?
	Packet Switching (PS)	Circuit Switching (CS)	
64B, 2 hops	13	27	108%
1KB, 2 hops	73	87	19%
64KB, 2 hops	265	279	5%
64B, 6 hops	33	75	127%
1KB, 6 hops	93	135	45%
64KB, 6 hops	285	327	15%

Message size, distance (2B-link)	Uncontended Latency (number of clock cycles)		How much slower?
	Packet Switching (PS)	Circuit Switching (CS)	
64B, 2 hops	41	55	34%
1KB, 2 hops	514	535	4%
64KB, 2 hops	2050	2071	1%
64B, 6 hops	54	103	91%
1KB, 6 hops	534	583	9%
64KB, 6 hops	2070	2119	2%

Conclusions

- Future architecture has both deep cache hierarchy as well as heterogeneous memories
- Increasingly important
  - Locality optimization
  - Persistent memory (with NVMM)
  - Processing in Memory
  - Bulk data transfer/operation

Thank you

I’d be happy to answer questions