Advanced Computer Architecture

Transactional Memory

Fall 2016

Pejman Lotfi-Kamran

Adapted from slides originally developed by Profs. Hill, Hoe, Falsafi and Wenisch of CMU, EPFL, Michigan, Wisconsin

Where Are We?

Fr	Sa	Su	Mo	Tu
	27-Shahrivar		29-Shahrivar	
	3-Mehr		5-Mehr	
	10-Mehr		12-Mehr	
	17-Mehr		19-Mehr	
	24-Mehr		26-Mehr	
	1-Aban		3-Aban	
	8-Aban		10-Aban	
	15-Aban		17-Aban	
	22-Aban		24-Aban	
	29-Aban		1-Azar	
	6-Azar		8-Azar	
	13-Azar		15-Azar	
	20-Azar		22-Azar	
	27-Azar		29-Azar	
	4-Dey		6-Dey	

- This Lecture
  - TM
- Next Lecture:
  - Storage

Transaction Memory

The following slides are from Sean Leeh

Current Parallel Programming Model

- Shared data consistency
- Use “Lock”
- Fine grained lock
  - Error prone
  - Deadlock prone
  - Overhead
- Coarse grained lock
  - Sequentialize threads
  - Prevent parallelism

```
// WITH LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

Thread 0
```
move(a, b, key1);
```

Thread 1
```
move(b, a, key2);
```

DEADLOCK!
(& can’t abort)

Code example source: Mark Hill @Wisconsin

Parallel Software Problems

- Parallel systems are often programmed with
  - Synchronization through **barriers**
  - Shared objects access control through **locks**
- Lock granularity and organization must balance performance and correctness
  - Coarse-grain locking: Lock contention
  - Fine-grain locking: Extra overhead
  - Must be careful to avoid deadlocks or data races
  - Must be careful not to leave anything unprotected for correctness
- Performance tuning is not intuitive
  - Performance bottlenecks are related to low level events
    - ▲E.g. false sharing, coherence misses
  - Feedback is often indirect (cache lines, rather than variables)

Parallel Hardware Complexity (TCC's view)

- Cache coherence protocols are complex
  - Must track ownership of cache lines
  - Difficult to implement and verify all corner cases
- Consistency protocols are complex
  - Must provide rules to correctly order individual loads/stores
  - Difficult for both hardware and software
- Current protocols rely on low latency, not bandwidth
  - Critical short control messages on ownership transfers
  - Latency of short messages unlikely to scale well in the future
  - Bandwidth is likely to scale much better
    - ▲High speed interchip connections
    - ▲Multicore (CMP) = on-chip bandwidth

What do we want?

- A shared memory system with
  - A simple, easy programming model (unlike message passing)
  - A simple, low-complexity hardware implementation (unlike shared memory)
  - Good performance

Lock Freedom

- Why lock is bad?
- Common problems in conventional locking mechanisms in concurrent systems
  - **Priority inversion**: When low-priority process is preempted while holding a lock needed by a high-priority process
  - **Convoying**: When a process holding a lock is de-scheduled (e.g. page fault, no more quantum), no forward progress for other processes capable of running
  - **Deadlock (or Livelock)**: Processes attempt to lock the same set of objects in different orders (could be bugs by programmers)
- Error-prone

Using Transactions

- What is a transaction?
  - A sequence of instructions that is guaranteed to execute and complete only as an **atomic** unit

```
Begin Transaction
    Inst #1
    Inst #2
    Inst #3
    …
End Transaction
```

  - Satisfy the following properties
    - Serializability: Transactions appear to execute serially.
    - Atomicity (or Failure-Atomicity): A transaction either
      - **commits** changes when complete, visible to all; or
      - **aborts,** discarding changes (will retry again)
    - Isolation: concurrently executing threads cannot affect the result of a transaction, so a transaction produces the same result as when no other task was executing

TCC (Stanford) [Hammond et al. ISCA 2004]

- Transactional Coherence and Consistency
- Programmer-defined groups of instructions within a program

```
Begin Transaction   Start Buffering Results
   Inst #1
   Inst #2
   Inst #3
      …
End Transaction     Commit Results Now
```

- Only commit machine state at **end** of each transaction
  - Each must update machine state **atomically**, all at once
  - To other processors, all instructions within one transaction **appear** to execute only when the transaction commits
  - These commits impose an **order** on how procs modify machine state

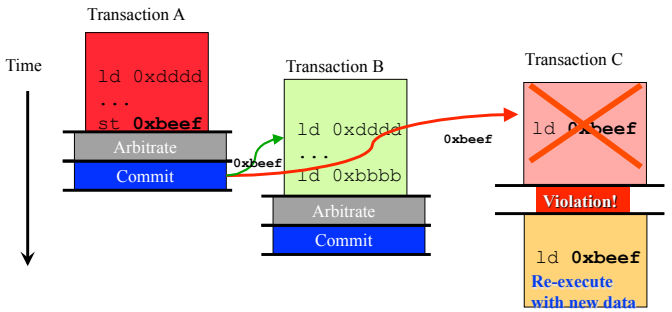
Transaction Code Example

- MIT LTM instruction set

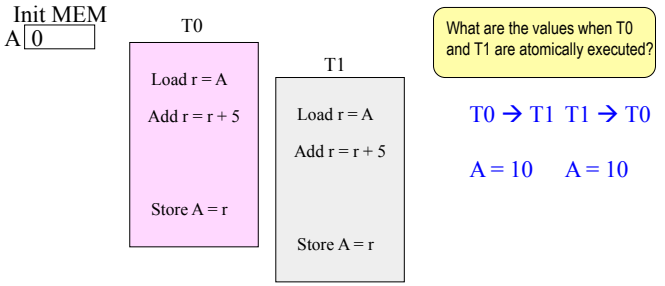
```
xstart:
        XBEGIN on_abort
        lw    r1, 0(r2)
        addi  r1, r1, 1
              . . .
        XEND
. . .
on_abort:
        …                   // back off
        j     xstart        // retry
```


Transactional Memory

- Transactions **appear** to execute in commit order
  - Flow (RAW) dependency cause transaction violation and restart



Transaction Atomicity

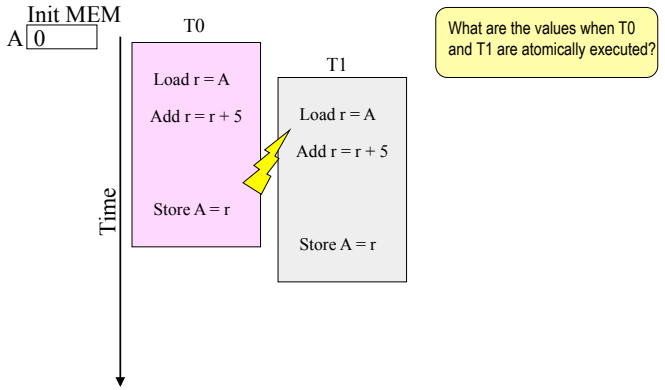


What are the values when T0 and T1 are atomically executed?

T0 → T1 T1 → T0

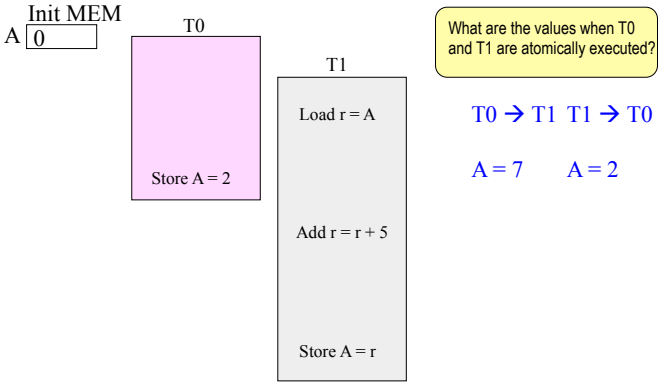
A = 10 A = 10

Transaction Atomicity



What are the values when T0 and T1 are atomically executed?

Transaction Atomicity

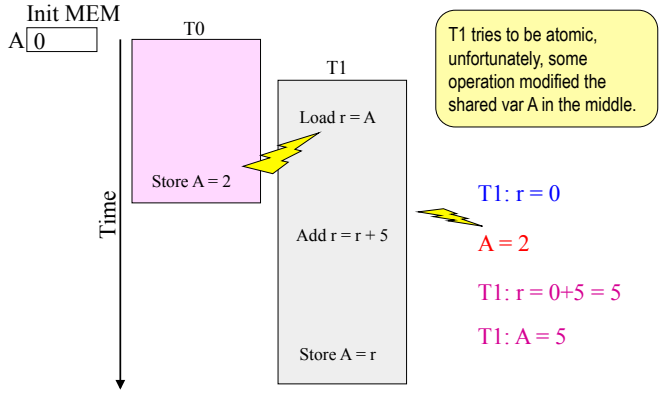


What are the values when T0 and T1 are atomically executed?

T0 → T1 T1 → T0

A = 7 A = 2

Transaction Atomicity



T1 tries to be atomic, unfortunately, some operation modified the shared var A in the middle.

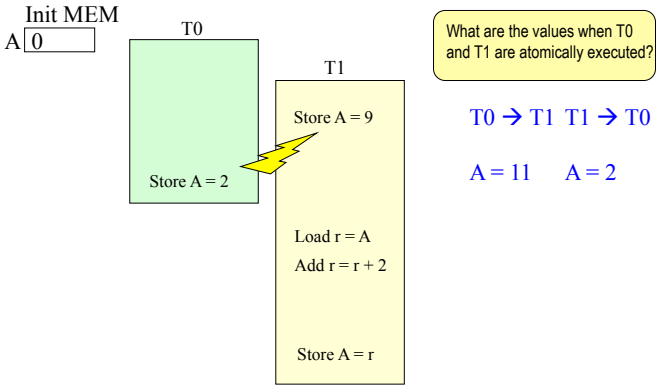
T1: r = 0

A = 2

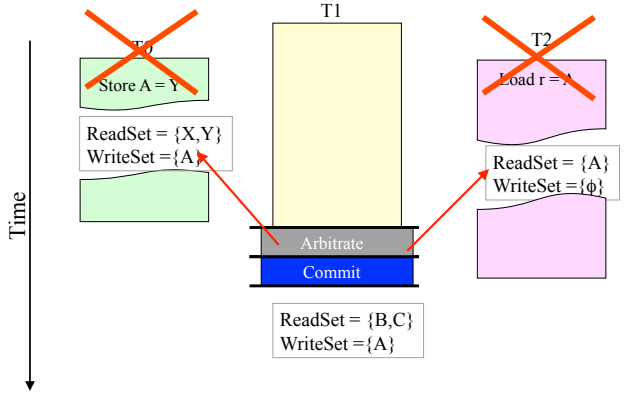
T1: r = 0+5 = 5

T1: A = 5

Transaction Atomicity



Transaction Atomicity



Hardware Transactional Memory Taxonomy

Conflict Detection
- Write set against another thread's read set and write set
  - Lazy
    - ▲Wait till last minute
  - Eager
    - ▲Check on each write
    - ▲Squash during a transaction

Version Management
- Where to put speculative data
  - Lazy
    - ▲Into speculative buffer (assuming transaction will abort)
    - ▲No rollback needed when abort
  - Eager
    - ▲Into cache hierarchy (assuming transaction will commit
    - ▲No data copy needed when go through

HTM Taxonomy [LogTM 2006]

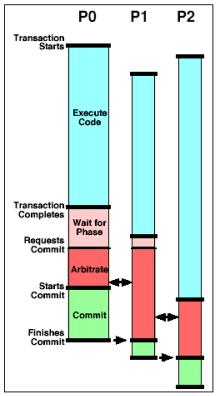
		Version Management	
		Lazy	**Eager**
Conflict Detection	**Lazy**	Optimistic C. Ctrl. DBMS	None
	Eager	MIT LTM Intel/Brown VTM	Conservative C. Ctrl DBMS MIT UTM LogTM

TCC System

- Similar to prior thread-level speculation (TLS) techniques
  - CMU Stampede
  - Stanford Hydra
  - Wisconsin Multiscalar
  - UIUC speculative multithreading CMP
- Loosely coupled TLS system
- Completely eliminates conventional cache coherence and consistency models
  - No MESI-style cache coherence protocol
- But require new hardware support

The TCC Cycle

- Transactions run in a cycle
- Speculatively execute code and buffer
- Wait for commit permission
  - **Phase** provides synchronization, if necessary (assigned phase number, oldest phase commit first)
  - Arbitrate with other processors
- Commit stores together (as a packet)
  - Provides a well-defined write ordering
  - Can invalidate or update other caches
  - Large packet utilizes bandwidth effectively
- And repeat



Advantages of TCC

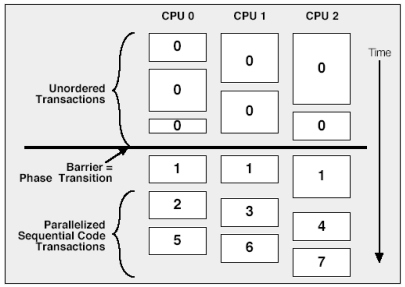
- Trades bandwidth for simplicity and latency tolerance
  - Easier to build
  - Not dependent on timing/latency of loads and stores
- Transactions eliminate locks
  - Transactions are inherently atomic
  - Catches most common parallel programming errors
- Shared memory **consistency** is simplified
  - Conventional model sequences individual loads and stores
  - Now only have hardware sequence *transaction commits*
- Shared memory **coherence** is simplified
  - Processors may have copies of cache lines in any state (no MESI !)
  - Commit order implies an *ownership* sequence

How to Use TCC

- Divide code into *potentially* parallel tasks
  - Usually loop iterations
  - For initial division, tasks = transactions
    - ▲But can be subdivided up or grouped to match HW limits (buffering)
  - Similar to threading in conventional parallel programming, but:
    - ▲We do not have to verify parallelism in advance
    - ▲Locking is handled automatically
    - ▲Easier to get parallel programs running correctly
- Programmer then *orders* transactions as necessary
  - Ordering techniques implemented using **phase number**
  - Deadlock-free (At least one transaction is the oldest one)
  - Livelock-free (watchdog HW can easily insert barriers anywhere)

How to Use TCC

- Three common ordering scenarios
  - Unordered for purely parallel tasks
  - Fully ordered to specify **sequential** task (algorithm level)
  - Partially ordered to insert synchronization like barriers



Basic TCC Transaction Control Bits

- In each local cache
  - **Read bits** (per cache line, or per word to eliminate false sharing)
    - ▲Set on ***speculative loads***
    - ▲Snooped by a committing transaction (writes by other CPU)
  - **Modified bits** (per cache line)
    - ▲Set on ***speculative stores***
    - ▲Indicate what to ***rollback*** if a violation is detected
    - ▲Different from dirty bit
  - **Renamed bits** (optional)
    - ▲At word or byte granularity
    - ▲To indicate local updates (RAW) that do not cause a violation
    - ▲Subsequent reads that read lines with these bits set, they do NOT set read bits because local RAW is not considered a violation

During A Transaction Commit

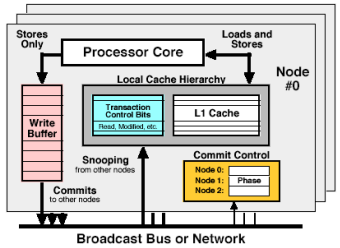
- Need to collect all of the modified caches together into a commit packet
- Potential solutions
  - A separate write buffer, or
  - An address buffer maintaining a list of the line tags to be committed
  - Size?
- Broadcast all writes out as one single (large) packet to the rest of the system

Re-execute A Transaction

- Rollback is needed when a transaction cannot commit
- Checkpoints needed prior to a transaction
- Checkpoint memory
  - Use **local** cache
  - Overflow issue
    - ▲Conflict or capacity misses require all the victim lines to be kept somewhere (e.g. victim cache)
- Checkpoint register state
  - Hardware approach: Flash-copying rename table / arch register file
  - Software approach: extra instruction overheads

Sample TCC Hardware

- Write buffers and L1 Transaction Control Bits
  - Write buffer in processor, before broadcast
- A broadcast bus or network to distribute commit packets
  - All processors see the commits in a single order
  - Snooping on broadcasts triggers violations, if necessary
- Commit arbitration/sequence logic



So far…

Replication & placement in CMP caches

But, CMP caches experience much pressure

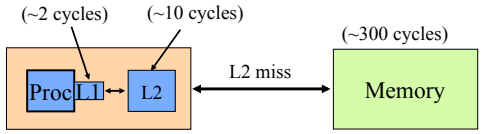
- Contention internal to the workload
- Contention across programs

For large last-level caches:

- need to think about adaptive/optimal
  - Insertion
    - ▲ Which "way" does one bring a block into
  - Promotion
    - ▲ Which position does one move a block into after a hit

Background

Fast processor + Slow memory ➔ Cache hierarchy



L1 misses ➔ Short latency, can be hidden

L2 misses ➔ Long-latency, hurts performance

Important to reduce Last Level (L2) cache misses

The following slides are from Moin Qureshi et al.

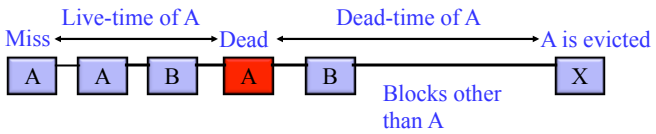
Motivation

- L1 for latency, L2 for capacity
- Traditionally L2 managed similar to L1 (typically LRU)
- L1 filters temporal locality ➔ Poor locality in L2
- LRU causes thrashing when working set > cache size

Most lines remain unused between insertion and eviction

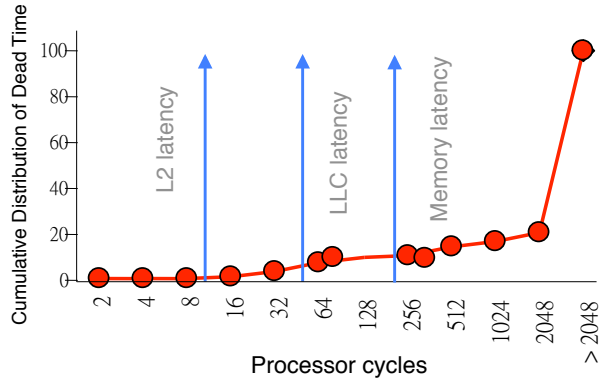
Life Cycle of Cache Blocks

- Two classes of blocks in the cache [Mendelson et al.]
  - Live blocks: referenced again before eviction
  - Dead blocks: **not** referenced before eviction



- Dead-time >> Live-time
  - Over 80% of the cache blocks are "dead"

L1 dead time distribution (one core) [Lai'01]



Dead Cache Space

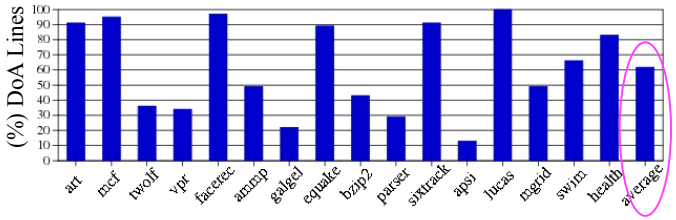
Opportunity for Optimization:
Prefetching, Victim Caching, Careful Sharing

Much work in this area

- Dead-block prediction
  - Timing based (wakeup timers on blocks) [Hu' 02]
  - Reference count based (timers) [Kharbutli' 08]
  - Trace based (rep. instruction traces) [Lai' 01]
  - Practical trace based [Ferdman' 06]
  - Cache bursts (better accuracy) [Liu' 08]
- Prefetching into dead space
  - Predict dead block, prefetch next block [Lai' 01]
- Victim caching in dead space
  - Much dead space in L2
  - Use as victim cache [Khan' 10]

Dead on Arrival (DoA) Lines

DoA Lines: Lines unused between insertion and eviction



For the 1MB 16-way L2, 60% of lines are DoA

➔ Ineffective use of cache space

Why DoA lines? mcf (left) art (right) snippets

```
while (arcin)
{
   tail = arcin->tail;

   if(tail->time + arcin->org_cost  > latest)
   {
      arcin= (arc_t *) tail->mark;
      continue;
   }
   ...
   arcin= (arc_t *) tail->mark
}
```

Causes 84% of all L2 misses (28% by each instruction)

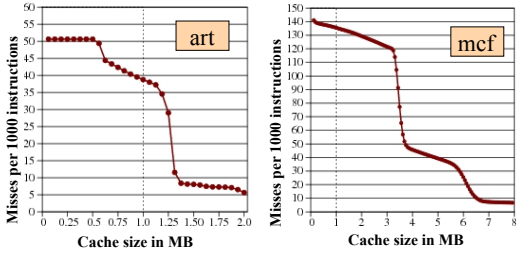
```
numf1s = lwidth*lheight;  // = 100*100 for ref input set
numf2s = numObjects+1;    // = 10+1 for ref input set
 •••
for (tj=spot;tj<numf2s;tj++)
{
   Y[tj].y = 0;

   if( !Y[tj].reset )
   for (ti=0;ti<numf1s;ti++)
   Y[tj].y += f1_layer[ti].P  * bus[ti][tj] ;
}
```

Causes 41% of all L2 misses

Causes 39% of all L2 misses

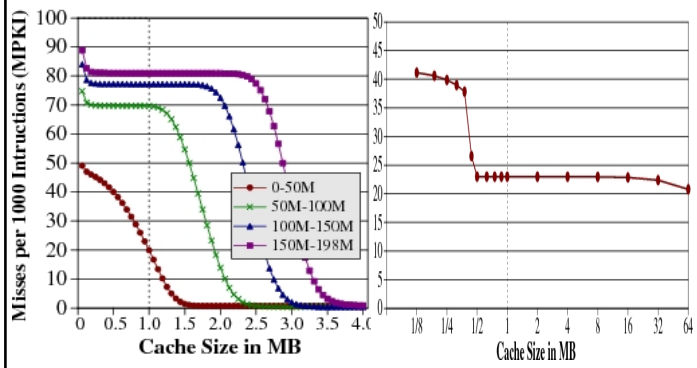
Why DoA Lines ?

- Streaming data ➔ Never reused. L2 caches don't help.
- Working set of application greater than cache size



Soln: if working set > cache size, retain some working set

health (left) and swim (right) mpki



Overview

Problem: LRU replacement inefficient for L2 caches

Goal: A replacement policy that has:
1. Low hardware overhead
2. Low complexity
3. High performance
4. Robust across workloads

Proposal: A mechanism that reduces misses by 21% and has total storage overhead < two bytes

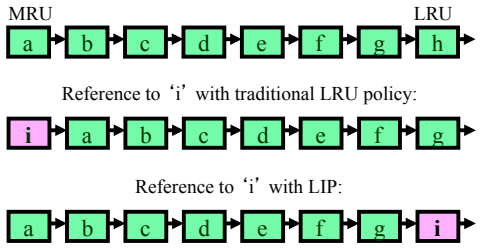
Cache Insertion Policy

Two components of cache replacement:

1. Victim Selection:
   Which line to replace for incoming line?
   (E.g. LRU, Random, FIFO, LFU)
2. Insertion Policy:
   Where is incoming line placed in replacement list? (E.g. insert incoming line at MRU position)

Simple changes to insertion policy can greatly improve cache performance for memory-intensive workloads

LRU-Insertion Policy (LIP)



Choose victim. Do NOT promote to MRU

Lines do not enter non-LRU positions unless reused

Bimodal-Insertion Policy (BIP)

LIP does not age older lines

Infrequently insert lines in MRU position

Let ε = Bimodal throttle parameter

```
if ( rand() < ε )
        Insert at MRU position;
else
        Insert at LRU position;
```

For small ε , BIP retains thrashing protection of LIP while responding to changes in working set

Circular Reference Model

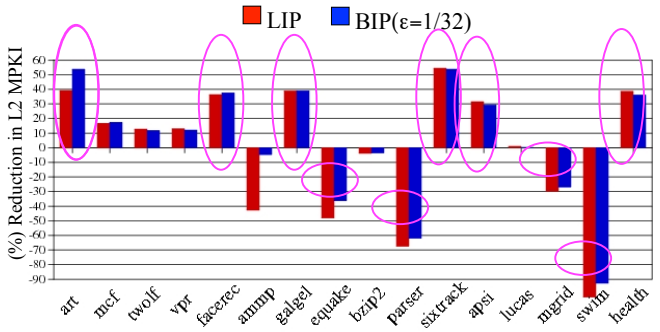
[Smith & Goodman, ISCA' 84]

Reference stream has T blocks and repeats N times.
Cache has K blocks (K<T and N>>T)

Policy	$(a_1\ a_2\ a_3\ \ldots\ a_T)^N$	$(b_1\ b_2\ b_3\ \ldots\ b_T)^N$
LRU		
OPT		
LIP		
BIP (small ε)		

For small ε , BIP retains thrashing protection of LIP while adapting to changes in working set

Results for LIP and BIP



Changes to insertion policy increases misses for LRU-friendly workloads

Dynamic-Insertion Policy (DIP)

Two types of workloads: LRU-friendly or BIP-friendly

DIP can be implemented by:

1. Monitor both policies (LRU and BIP)
2. Choose the best-performing policy
3. Apply the best policy to the cache

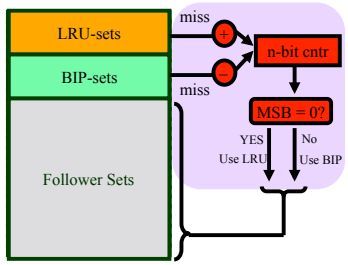
Need a cost-effective implementation ➔ "Set Dueling"

DIP via "Set Dueling"

Divide the cache in three:
- Dedicated LRU sets
- Dedicated BIP sets
- Follower sets (winner of LRU,BIP)

n-bit saturating counter
misses to LRU-sets: counter++
misses to BIP-set: counter--

Counter decides policy for Follower sets:
- MSB = 0, Use LRU
- MSB = 1, Use BIP



monitor ➔ choose ➔ apply
(using a single counter)

Bounds on Dedicated Sets

How many dedicated sets required for "Set Dueling"?

$\mu_{LRU}, \sigma_{LRU}, \mu_{BIP}, \sigma_{BIP}$ = Avg. misses and stdev. for LRU and BIP

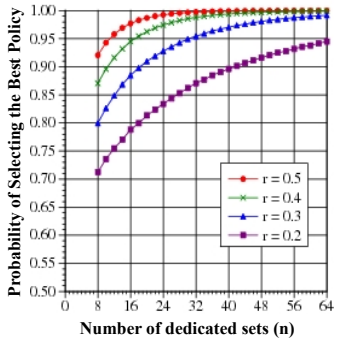
P(Best) = probability of selecting best policy

$P(Best) = P(Z < r\sqrt{n})$

n = number of dedicated sets
Z = standard Gaussian variable

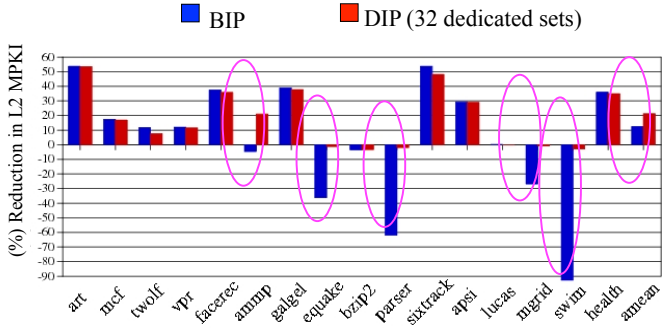
$r = |\mu_{LRU} - \mu_{BIP}| / \sqrt{(\sigma_{LRU}^2 + \sigma_{BIP}^2)}$

(For majority workloads r > 0.2)



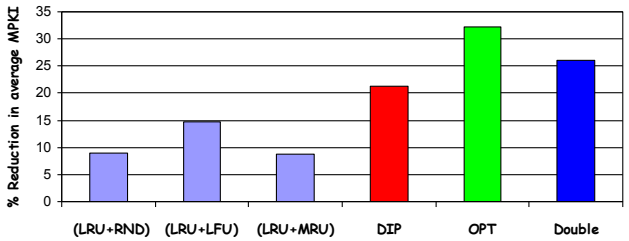
32-64 dedicated sets sufficient

Results for DIP



DIP reduces average MPKI by 21% and requires < two bytes storage overhead

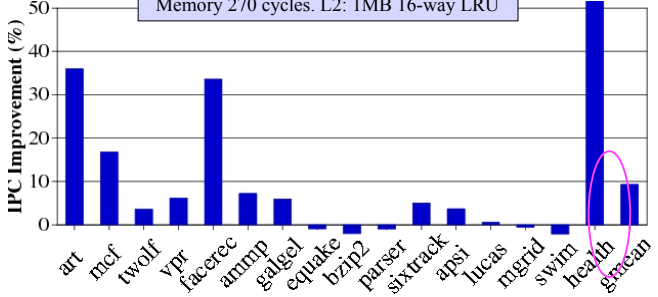
DIP vs. Other Policies



DIP bridges two-thirds of gap between LRU and OPT

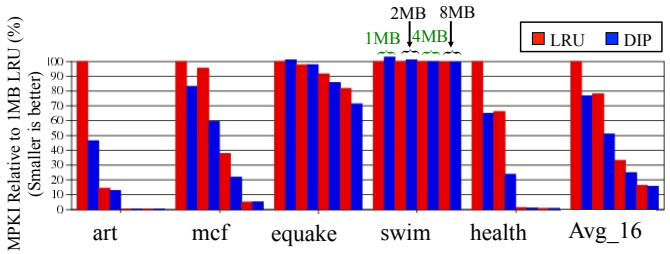
IPC Improvement

Processor: 4 wide, 32-entry window
Memory 270 cycles. L2: 1MB 16-way LRU



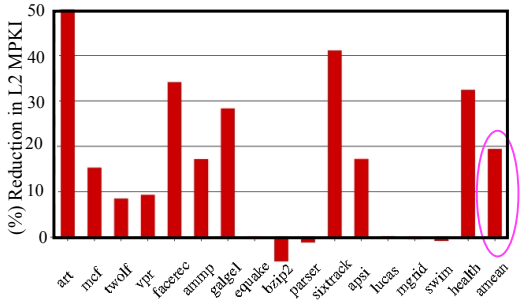
DIP Improves IPC by 9.3% on average

DIP vs. LRU Across Cache Sizes



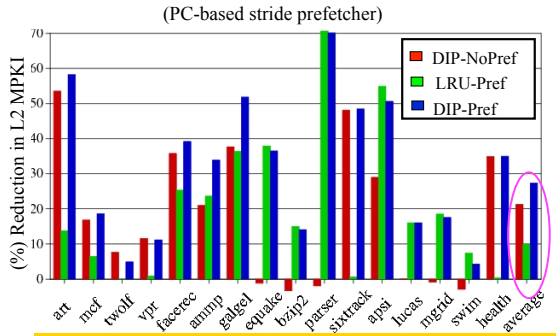
MPKI reduces till workload fits in the cache

DIP with 1MB 8-way L2 Cache



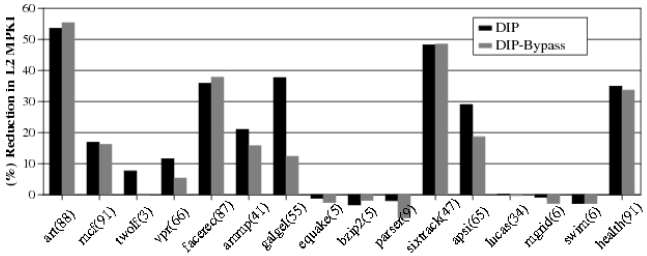
MPKI reduction with 8-way (19%) similar to 16-way (21%)

Interaction with Prefetching

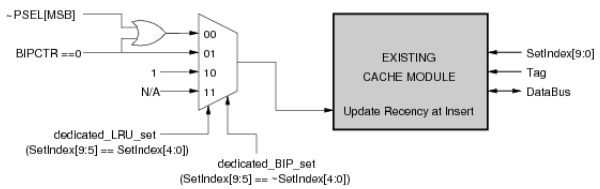


DIP also works well in presence of prefetching

DIP Bypass



DIP (design and implementation)



Summary

LRU inefficient for L2 caches. Most lines remain unused between insertion and eviction

Proposed changes to cache insertion policy (DIP) has:

✓ 1. Low hardware overhead
Requires < two bytes storage overhead

✓ 2. Low complexity
Trivial to implement. No changes to cache structure

✓ 3. High performance
Reduces misses by 21%. Two-thirds as good as OPT

✓ 4. Robust across workloads
Almost as good as LRU for LRU-friendly workloads