18-447
Computer Architecture
Lecture 26: Prefetching & Emerging Memory Technologies

Prof. Onur Mutlu
Carnegie Mellon University

Spring 2015, 4/3/2015

Lab 5 Honors (Critical Path)

Zhipeng Zhao	4.48ns	1%
Raghav Gupta	4.52ns	1%
Amanda Marano	6.37ns	0.8%
Elena Feldman	7.67ns	0.7%
Prin Oungpasuk	8.55ns	0.5%
Derek Tzeng	9.58ns	0.3%
Abhijith Kashyap	9.75ns	0.3%

Lab 5 Honors (4-Way Cache Design)

Jared Choi	1.5%
Kais Kudrolli	1.25%
Junhan Zhou	1.25%
Raghav Gupta	1.25%
Pete Ehrett	0.6%
Xiaofan Li	0.1%

Where We Are in Lecture Schedule

- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory

- Multiprocessors
- Coherence and consistency
- Interconnection networks
- Multi-core issues (e.g., heterogeneous multi-core)

Required Reading

- Onur Mutlu, Justin Meza, and Lavanya Subramanian, **"The Main Memory System: Challenges and Opportunities"**
  *Invited Article in Communications of the Korean Institute of Information Scientists and Engineers (**KIISE**)*, 2015.

http://users.ece.cmu.edu/~omutlu/pub/main-memory-system_kiise15.pdf

Prefetching

Review: Outline of Prefetching Lecture(s)

- Why prefetch? Why could/does it work?
- The four questions
  - What (to prefetch), when, where, how
- Software prefetching
- Hardware prefetching algorithms
- Execution-based prefetching
- Prefetching performance
  - Coverage, accuracy, timeliness
  - Bandwidth consumption, cache pollution
- Prefetcher throttling
- Issues in multi-core (if we get to it)

Review: How to Prefetch More Irregular Access Patterns?

- Regular patterns: Stride, stream prefetchers do well
- More irregular access patterns
  - Indirect array accesses
  - Linked data structures
  - Multiple regular strides (1,2,3,1,2,3,1,2,3,…)
  - Random patterns?
  - Generalized prefetcher for all patterns?

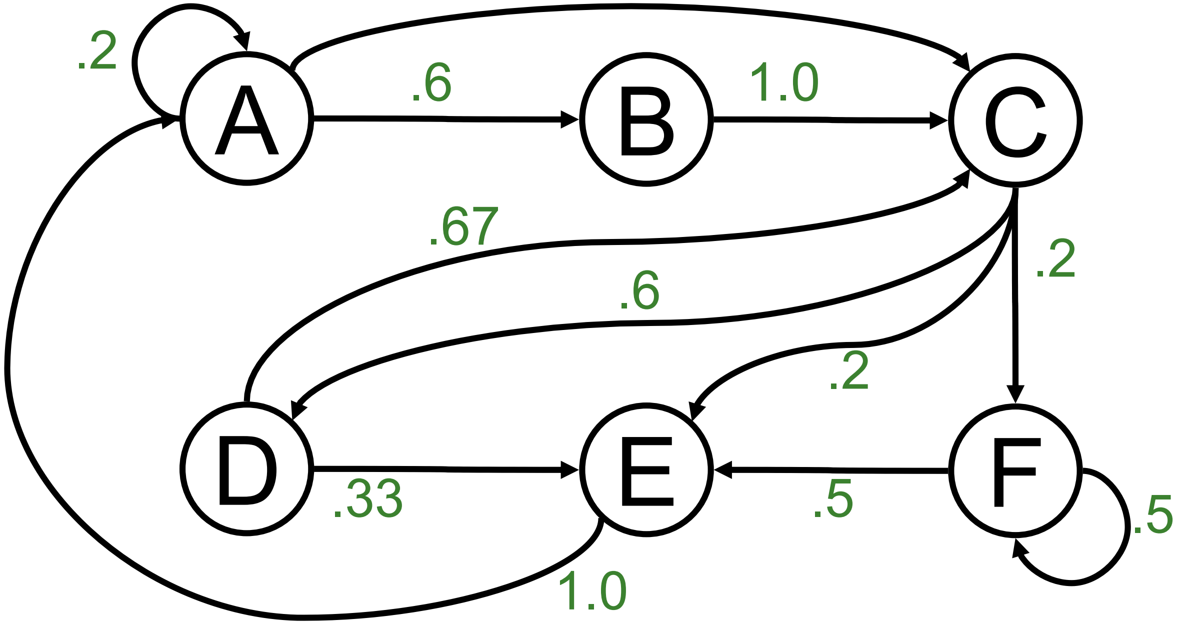
- Correlation based prefetchers
- Content-directed prefetchers
- Precomputation or execution-based prefetchers

Address Correlation Based Prefetching (I)

- Consider the following history of cache block addresses

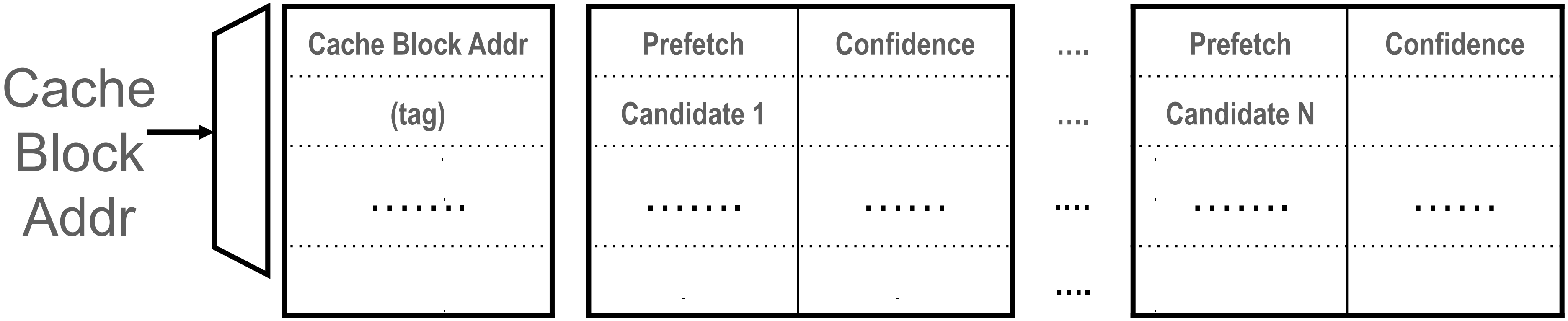
  A, B, C, D, C, E, A, C, F, F, E, A, A, B, C, D, E, A, B, C, D, C

- After referencing a particular address (say A or E), are some addresses more likely to be referenced next



Markov Model

Address Correlation Based Prefetching (II)



- Idea: Record the likely-next addresses (B, C, D) after seeing an address A
  - Next time A is accessed, prefetch B, C, D
  - A is said to be correlated with B, C, D
- Prefetch up to N next addresses to increase *coverage*
- Prefetch accuracy can be improved by using multiple addresses as key for the next address: (A, B) → (C)

  (A,B) correlated with C

- Joseph and Grunwald, "Prefetching using Markov Predictors," ISCA 1997.
  - Also called "Markov prefetchers"

Address Correlation Based Prefetching (III)

- Advantages:
  - Can cover arbitrary access patterns
    - Linked data structures
    - Streaming patterns (though not so efficiently!)

- Disadvantages:
  - Correlation table needs to be very large for high coverage
    - Recording every miss address and its subsequent miss addresses is infeasible
  - Can have low timeliness: Lookahead is limited since a prefetch for the next access/miss is initiated right after previous
  - Can consume a lot of memory bandwidth
    - Especially when Markov model probabilities (correlations) are low
  - Cannot reduce compulsory misses

Content Directed Prefetching (I)

- A specialized prefetcher for pointer values
- Idea: Identify pointers among all values in a fetched cache block and issue prefetch requests for them.
  - Cooksey et al., "A stateless, content-directed data prefetching mechanism," ASPLOS 2002.

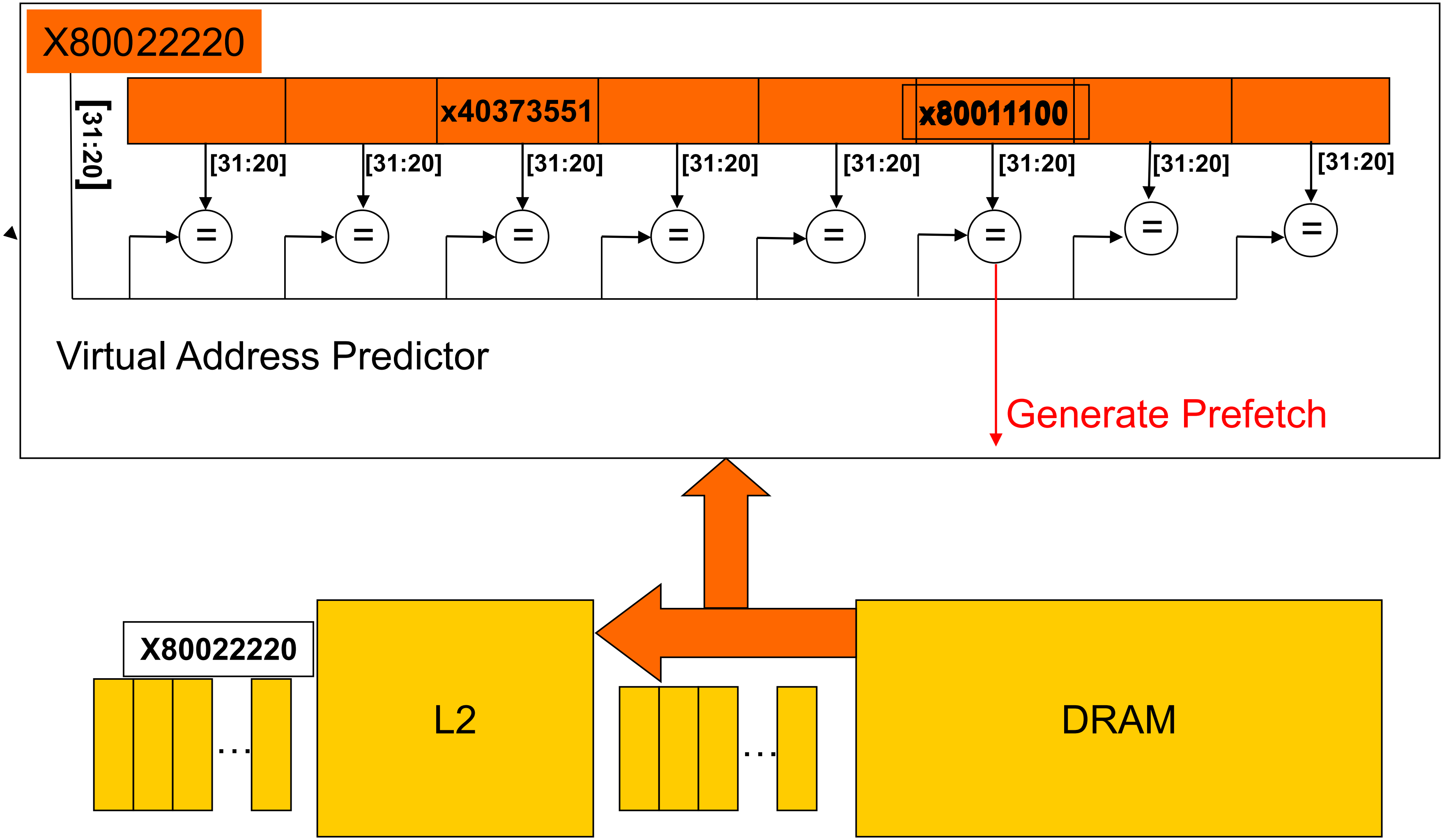
+ No need to memorize/record past addresses!

+ Can eliminate compulsory misses (never-seen pointers)

-- Indiscriminately prefetches *all* pointers in a cache block

- How to identify pointer addresses:
  - Compare address sized values within cache block with cache block's address → if most-significant few bits match, pointer

Content Directed Prefetching (II)



Execution-based Prefetchers (I)

- Idea: Pre-execute a piece of the (pruned) program solely for prefetching data
  - Only need to distill pieces that lead to cache misses

- Speculative thread: Pre-executed program piece can be considered a "thread"

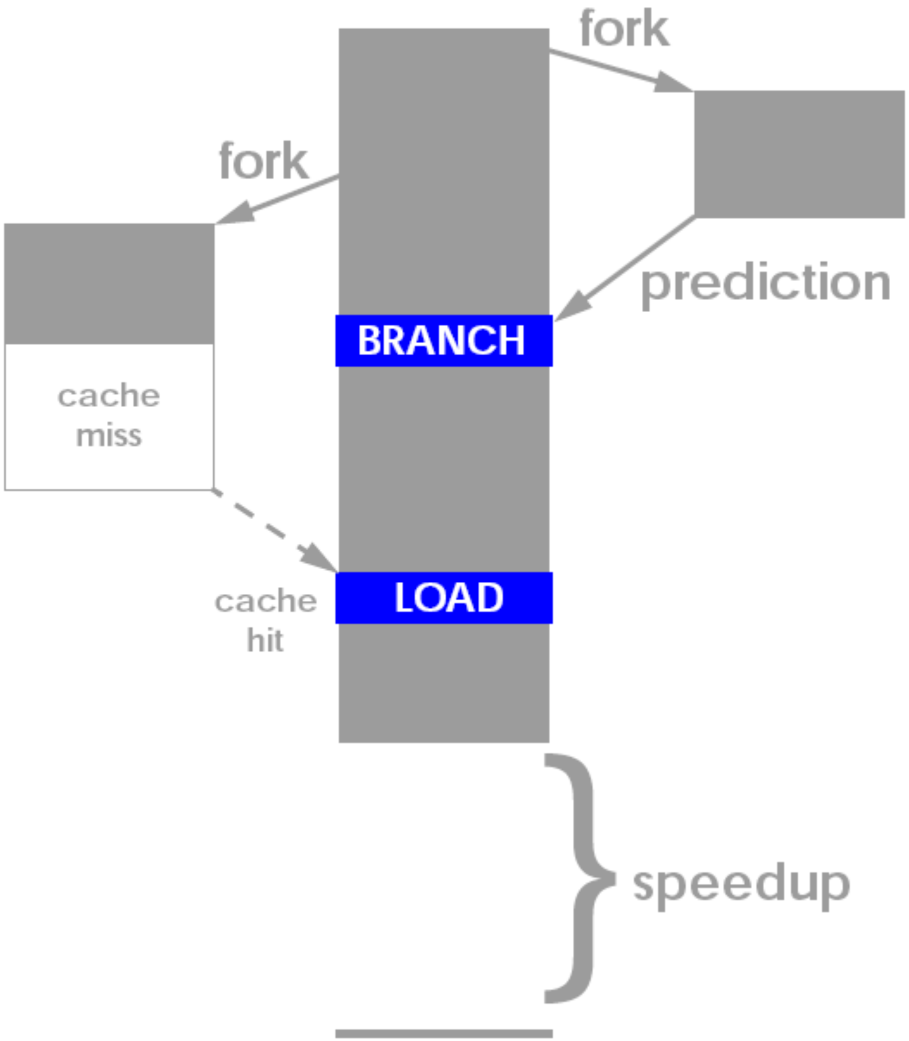
- Speculative thread can be executed
  - On a separate processor/core
  - On a separate hardware thread context (think fine-grained multithreading)
  - On the same thread context in idle cycles (during cache misses)

Execution-based Prefetchers (II)

- How to construct the speculative thread:
  - Software based pruning and "spawn" instructions
  - Hardware based pruning and "spawn" instructions
  - Use the original program (no construction), but
    - Execute it faster without stalling and correctness constraints

- Speculative thread
  - Needs to discover misses before the main program
    - Avoid waiting/stalling and/or compute less
  - To get ahead, uses
    - Perform only address generation computation, branch prediction, value prediction (to predict "unknown" values)
  - Purely speculative so there is no need for recovery of main program if the speculative thread is incorrect

Thread-Based Pre-Execution



- Dubois and Song, "Assisted Execution," USC Tech Report 1998.

- Chappell et al., "Simultaneous Subordinate Microthreading (SSMT)," ISCA 1999.

- Zilles and Sohi, "Execution-based Prediction Using Speculative Slices", ISCA 2001.

Thread-Based Pre-Execution Issues

- Where to execute the precomputation thread?

  1. Separate core (least contention with main thread)
  2. Separate thread context on the same core (more contention)
  3. Same core, same context
     - When the main thread is stalled

- When to spawn the precomputation thread?

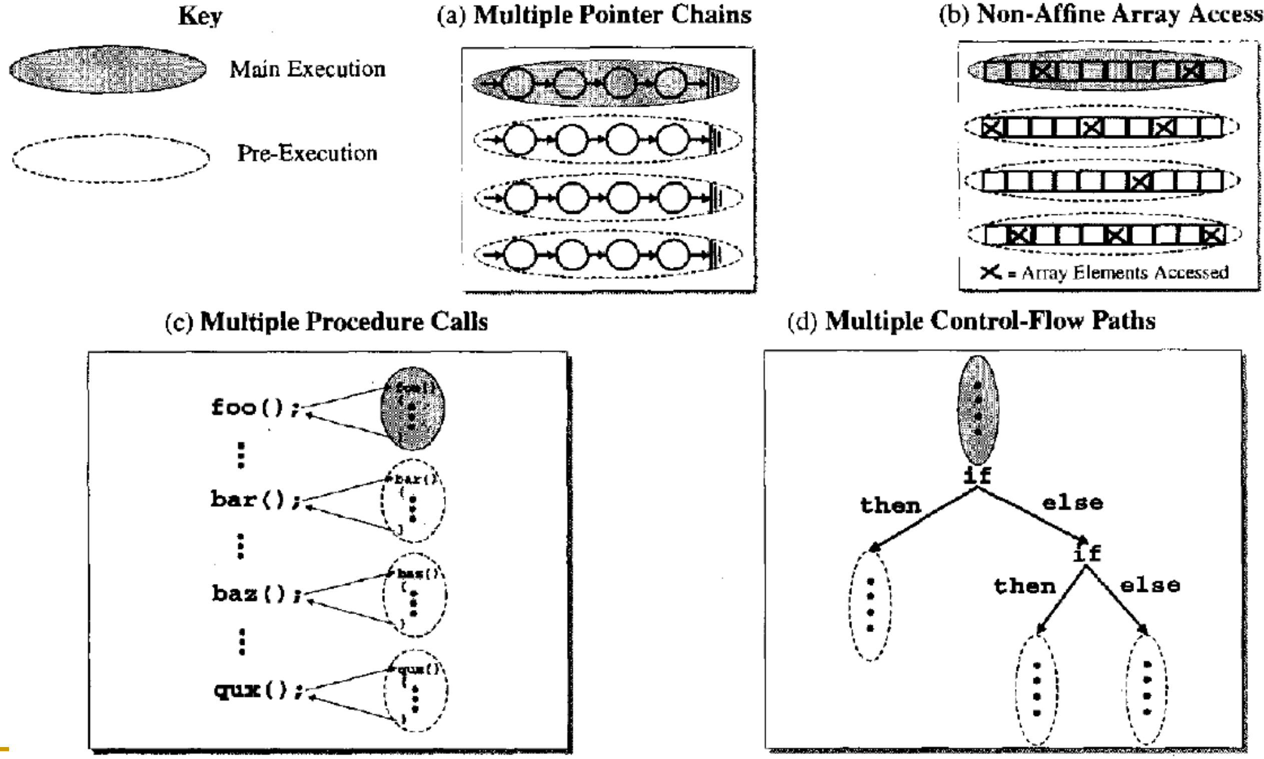
  1. Insert spawn instructions well before the “problem” load
     - How far ahead?
       - Too early: prefetch might not be needed
       - Too late: prefetch might not be timely
  2. When the main thread is stalled

- When to terminate the precomputation thread?

  1. With pre-inserted CANCEL instructions
  2. Based on effectiveness/contention feedback (recall throttling)

Thread-Based Pre-Execution Issues

- What, when, where, how
  - Luk, "Tolerating Memory Latency through Software-Controlled Pre-Execution in Simultaneous Multithreading Processors," ISCA 2001.
  - Many issues in software-based pre-execution discussed



Key

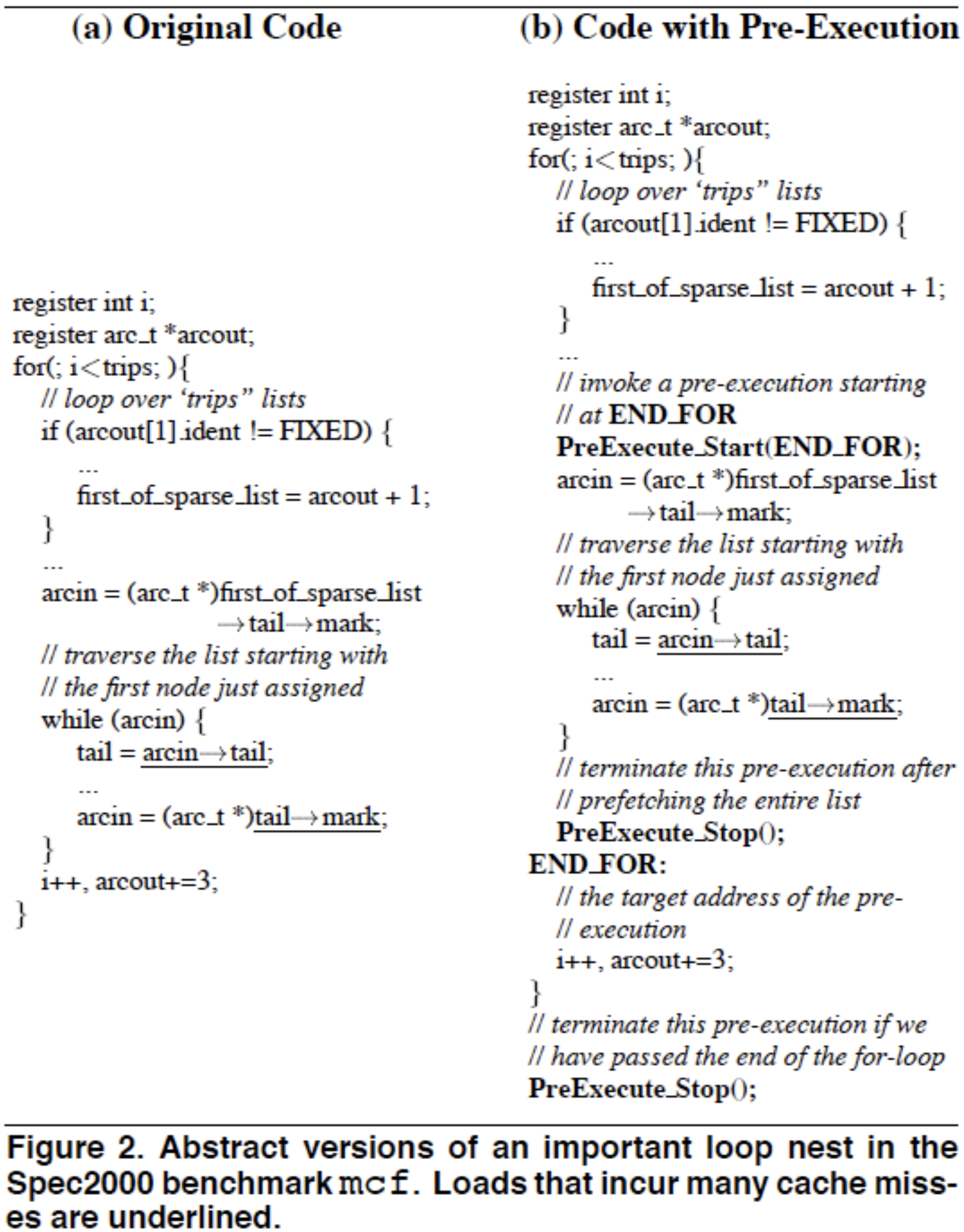
(a) Multiple Pointer Chains

(b) Non-Affine Array Accesses

(c) Multiple Procedure Calls

(d) Multiple Control-Flow Paths

An Example



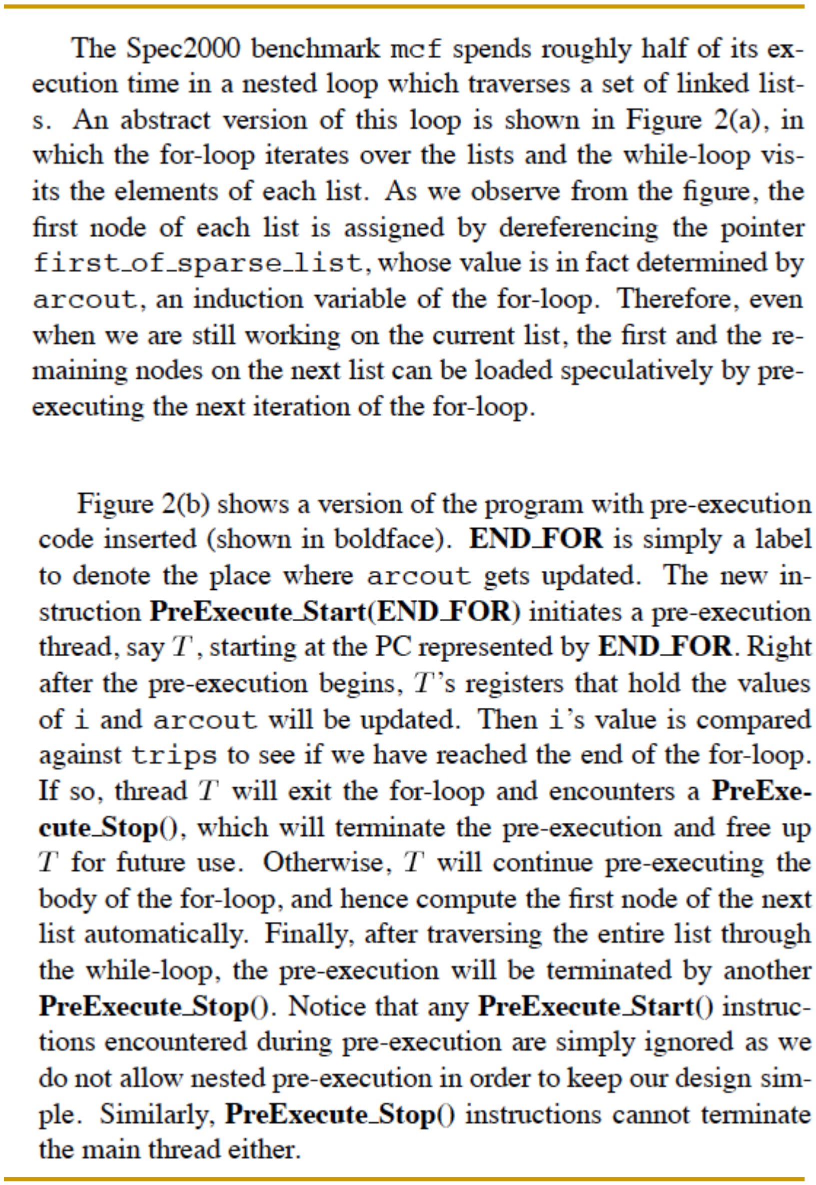
(a) Original Code

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
    // loop over 'trips" lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    arcin = (arc_t *)first_of_sparse_list
                        →tail→mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin→tail;
        ...
        arcin = (arc_t *)tail→mark;
    }
    i++, arcout+=3;
}
```

(b) Code with Pre-Execution

```
register int i;
register arc_t *arcout;
for(; i<trips; ){
    // loop over 'trips" lists
    if (arcout[1].ident != FIXED) {
        ...
        first_of_sparse_list = arcout + 1;
    }
    ...
    // invoke a pre-execution starting
    // at END_FOR
    PreExecute_Start(END_FOR);
    arcin = (arc_t *)first_of_sparse_list
            →tail→mark;
    // traverse the list starting with
    // the first node just assigned
    while (arcin) {
        tail = arcin→tail;
        ...
        arcin = (arc_t *)tail→mark;
    }
    // terminate this pre-execution after
    // prefetching the entire list
    PreExecute_Stop();
END_FOR:
    // the target address of the pre-
    // execution
    i++, arcout+=3;
}
// terminate this pre-execution if we
// have passed the end of the for-loop
PreExecute_Stop();
```

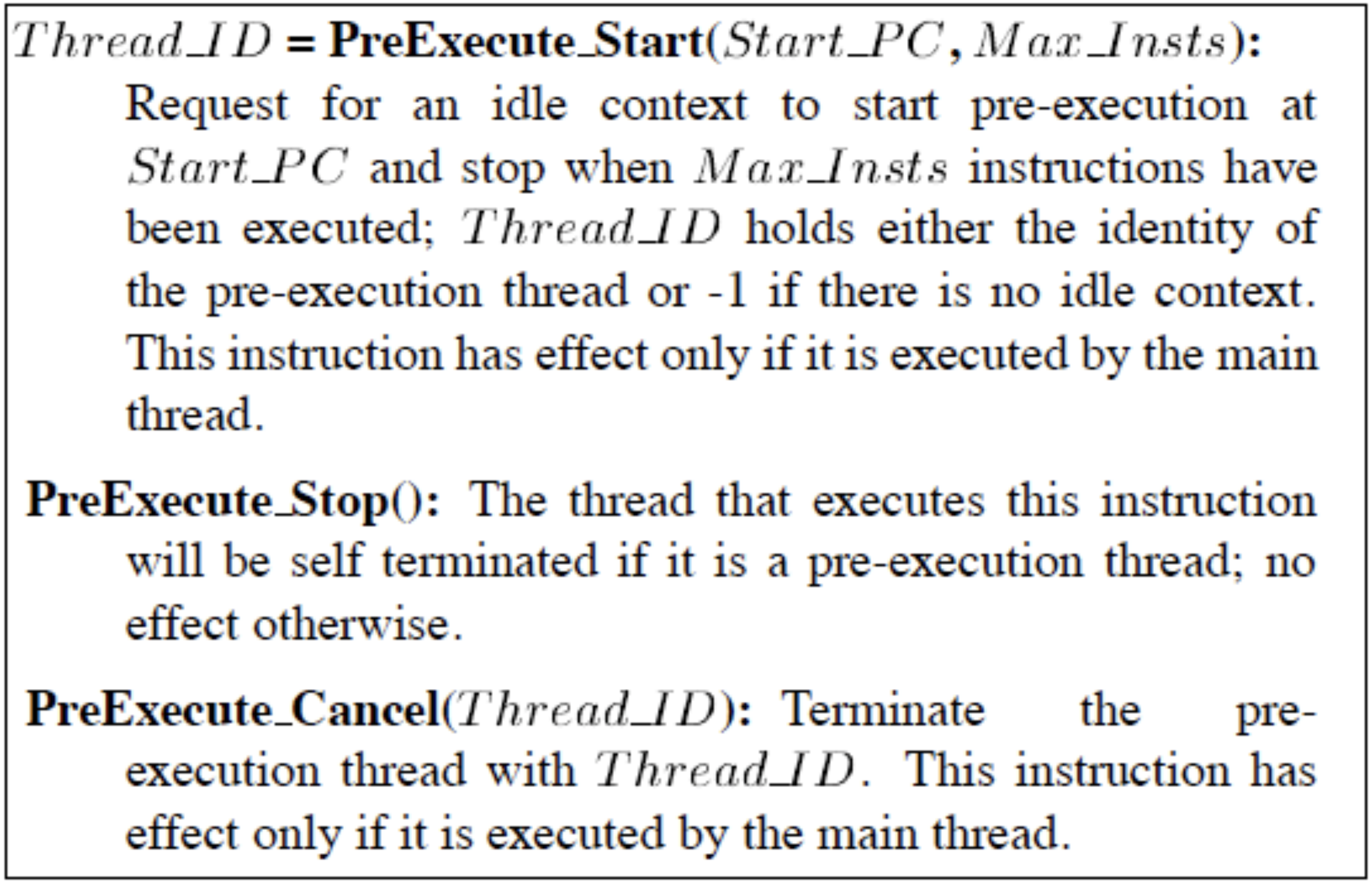
Figure 2. Abstract versions of an important loop nest in the Spec2000 benchmark mcf. Loads that incur many cache misses are underlined.



The Spec2000 benchmark mcf spends roughly half of its execution time in a nested loop which traverses a set of linked lists. An abstract version of this loop is shown in Figure 2(a), in which the for-loop iterates over the lists and the while-loop visits the elements of each list. As we observe from the figure, the first node of each list is assigned by dereferencing the pointer first_of_sparse_list, whose value is in fact determined by arcout, an induction variable of the for-loop. Therefore, even when we are still working on the current list, the first and the remaining nodes on the next list can be loaded speculatively by pre-executing the next iteration of the for-loop.

Figure 2(b) shows a version of the program with pre-execution code inserted (shown in boldface). **END_FOR** is simply a label to denote the place where arcout gets updated. The new instruction **PreExecute_Start(END_FOR)** initiates a pre-execution thread, say T, starting at the PC represented by **END_FOR**. Right after the pre-execution begins, T's registers that hold the values of i and arcout will be updated. Then i's value is compared against trips to see if we have reached the end of the for-loop. If so, thread T will exit the for-loop and encounters a **PreExecute_Stop**(), which will terminate the pre-execution and free up T for future use. Otherwise, T will continue pre-executing the body of the for-loop, and hence compute the first node of the next list automatically. Finally, after traversing the entire list through the while-loop, the pre-execution will be terminated by another **PreExecute_Stop**(). Notice that any **PreExecute_Start**() instructions encountered during pre-execution are simply ignored as we do not allow nested pre-execution in order to keep our design simple. Similarly, **PreExecute_Stop**() instructions cannot terminate the main thread either.

Example ISA Extensions



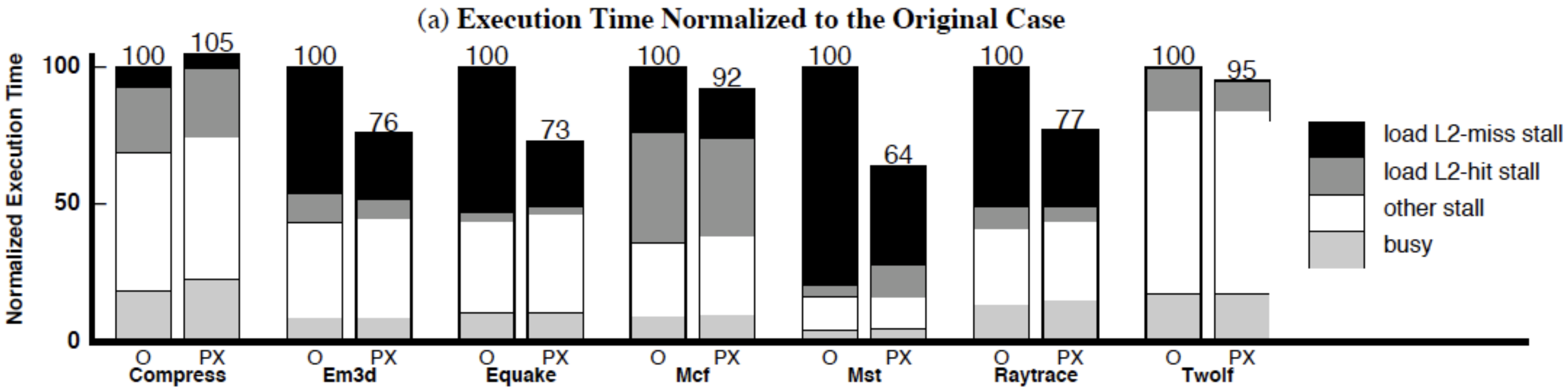
$Thread\_ID$ = **PreExecute_Start**($Start\_PC$, Max_Insts)**:** Request for an idle context to start pre-execution at $Start\_PC$ and stop when Max_Insts instructions have been executed; $Thread\_ID$ holds either the identity of the pre-execution thread or -1 if there is no idle context. This instruction has effect only if it is executed by the main thread.

PreExecute_Stop(): The thread that executes this instruction will be self terminated if it is a pre-execution thread; no effect otherwise.

PreExecute_Cancel($Thread\_ID$)**:** Terminate the pre-execution thread with $Thread\_ID$. This instruction has effect only if it is executed by the main thread.

Figure 4. Proposed instruction set extensions to support pre-execution. (C syntax is used to improve readability.)

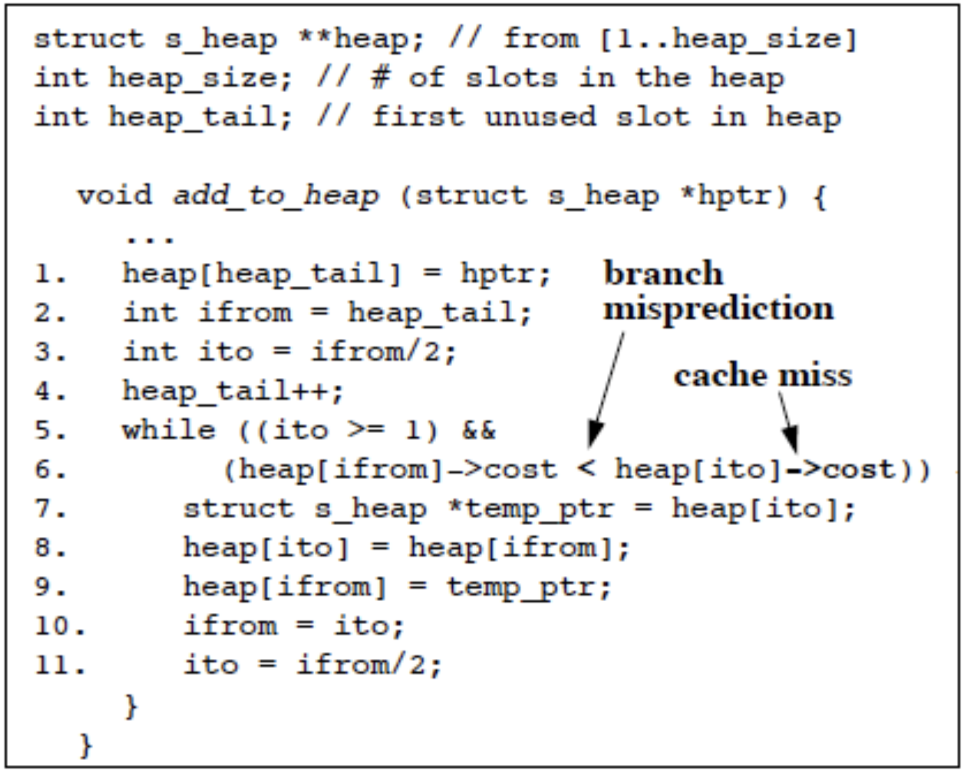
Results on a Multithreaded Processor



Problem Instructions

- Zilles and Sohi, “Execution-based Prediction Using Speculative Slices”, ISCA 2001.
- Zilles and Sohi, ”Understanding the backward slices of performance degrading instructions,” ISCA 2000.

Figure 2. *Example problem instructions from heap insertion routine in* **vpr**.



```
struct s_heap **heap; // from [1..heap_size]
int heap_size; // # of slots in the heap
int heap_tail; // first unused slot in heap

  void add_to_heap (struct s_heap *hptr) {
    ...
    heap[heap_tail] = hptr;
    int ifrom = heap_tail;
    int ito = ifrom/2;
    heap_tail++;
    while ((ito >= 1) &&
         (heap[ifrom]->cost < heap[ito]->cost))
       struct s_heap *temp_ptr = heap[ito];
       heap[ito] = heap[ifrom];
       heap[ifrom] = temp_ptr;
       ifrom = ito;
       ito = ifrom/2;
    }
  }
```

branch misprediction
cache miss

Fork Point for Prefetching Thread

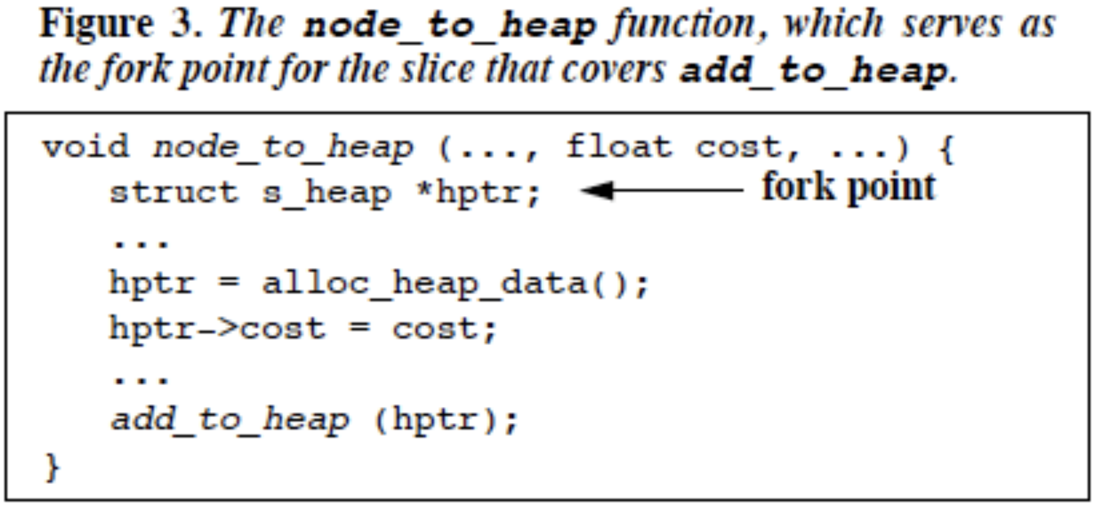
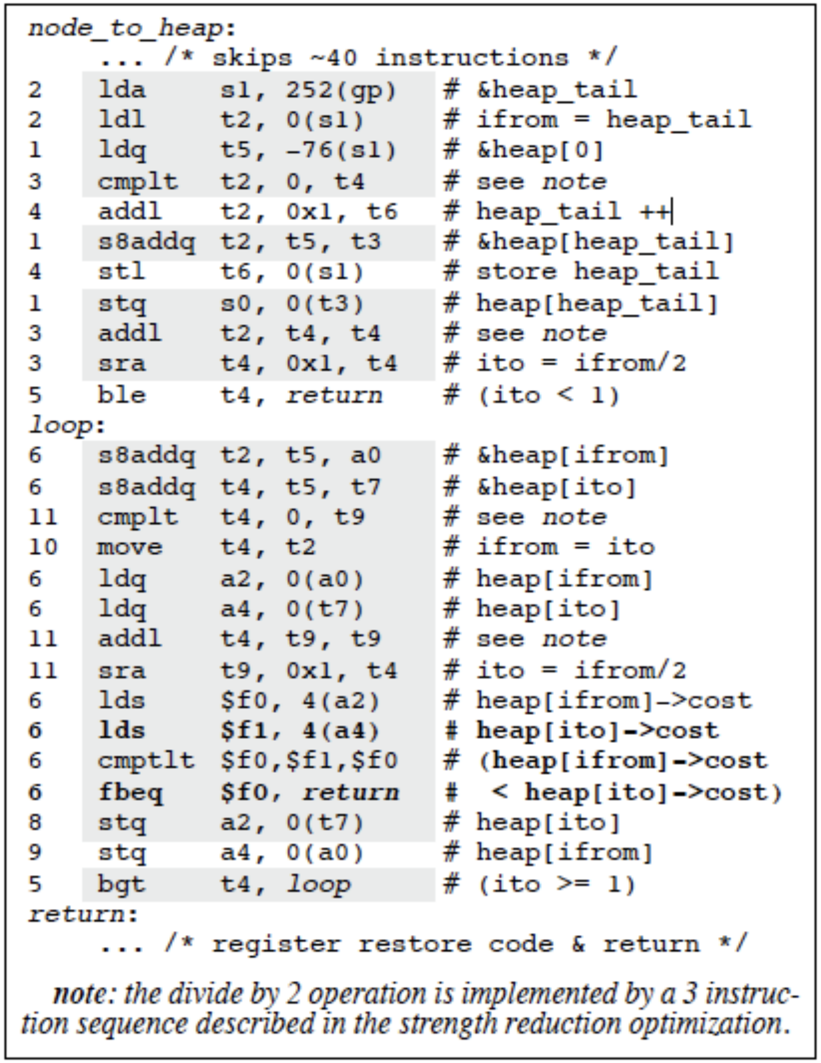


Figure 3. *The* **`node_to_heap`** *function, which serves as the fork point for the slice that covers* **`add_to_heap`**.

```
void node_to_heap (..., float cost, ...) {
    struct s_heap *hptr;  ←——— fork point
    ...
    hptr = alloc_heap_data();
    hptr->cost = cost;
    ...
    add_to_heap (hptr);
}
```

Pre-execution Thread Construction

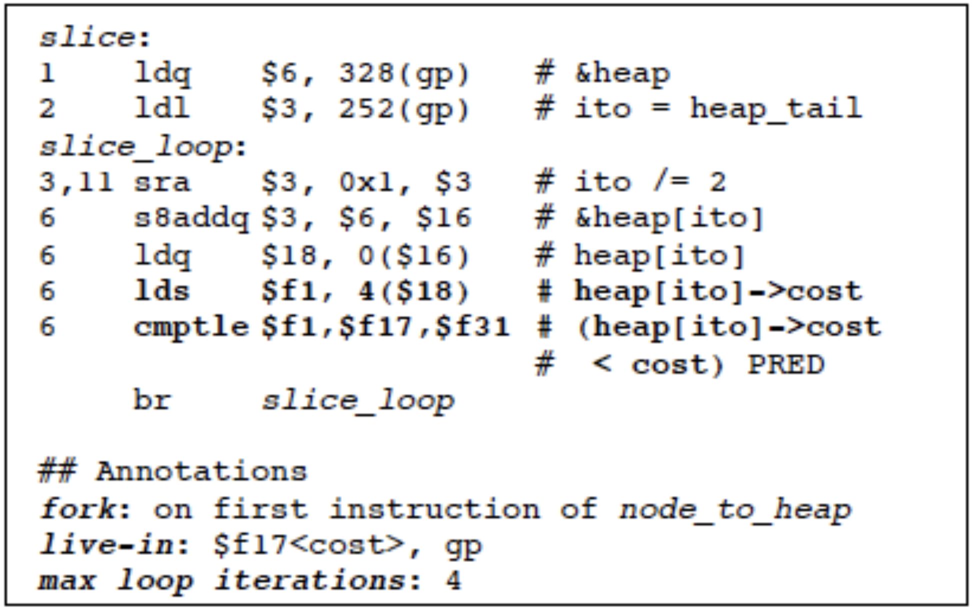
Figure 4. *Alpha assembly for the* **`add_to_heap`** *function. The instructions are annotated with the number of the line in Figure 2 to which they correspond. The problem instructions are in bold and the shaded instructions comprise the un-optimized slice.*



```
node_to_heap:
      ... /* skips ~40 instructions */
2     lda     s1, 252(gp)     # &heap_tail
2     ldl     t2, 0(s1)       # ifrom = heap_tail
1     ldq     t5, -76(s1)     # &heap[0]
3     cmplt   t2, 0, t4       # see note
4     addl    t2, 0x1, t6     # heap_tail ++
1     s8addq  t2, t5, t3      # &heap[heap_tail]
4     stl     t6, 0(s1)       # store heap_tail
1     stq     s0, 0(t3)       # heap[heap_tail]
3     addl    t2, t4, t4      # see note
3     sra     t4, 0x1, t4     # ito = ifrom/2
5     ble     t4, return      # (ito < 1)
loop:
6     s8addq  t2, t5, a0      # &heap[ifrom]
6     s8addq  t4, t5, t7      # &heap[ito]
11    cmplt   t4, 0, t9       # see note
10    move    t4, t2          # ifrom = ito
6     ldq     a2, 0(a0)       # heap[ifrom]
6     ldq     a4, 0(t7)       # heap[ito]
11    addl    t4, t9, t9      # see note
11    sra     t9, 0x1, t4     # ito = ifrom/2
6     lds     $f0, 4(a2)      # heap[ifrom]->cost
6     lds     $f1, 4(a4)      # heap[ito]->cost
6     cmptlt  $f0,$f1,$f0     # (heap[ifrom]->cost
6     fbeq    $f0, return     #  < heap[ito]->cost)
8     stq     a2, 0(t7)       # heap[ito]
9     stq     a4, 0(a0)       # heap[ifrom]
5     bgt     t4, loop        # (ito >= 1)
return:
      ... /* register restore code & return */
```

note: the divide by 2 operation is implemented by a 3 instruction sequence described in the strength reduction optimization.

Figure 5. *Slice constructed for example problem instructions. Much smaller than the original code, the slice contains a loop that mimics the loop in the original code.*



```
slice:
1     ldq     $6, 328(gp)     # &heap
2     ldl     $3, 252(gp)     # ito = heap_tail
slice_loop:
3,11  sra     $3, 0x1, $3     # ito /= 2
6     s8addq  $3, $6, $16     # &heap[ito]
6     ldq     $18, 0($16)     # heap[ito]
6     lds     $f1, 4($18)     # heap[ito]->cost
6     cmptle  $f1,$f17,$f31   # (heap[ito]->cost
                              #  < cost) PRED
      br      slice_loop

## Annotations
fork: on first instruction of node_to_heap
live-in: $f17<cost>, gp
max loop iterations: 4
```

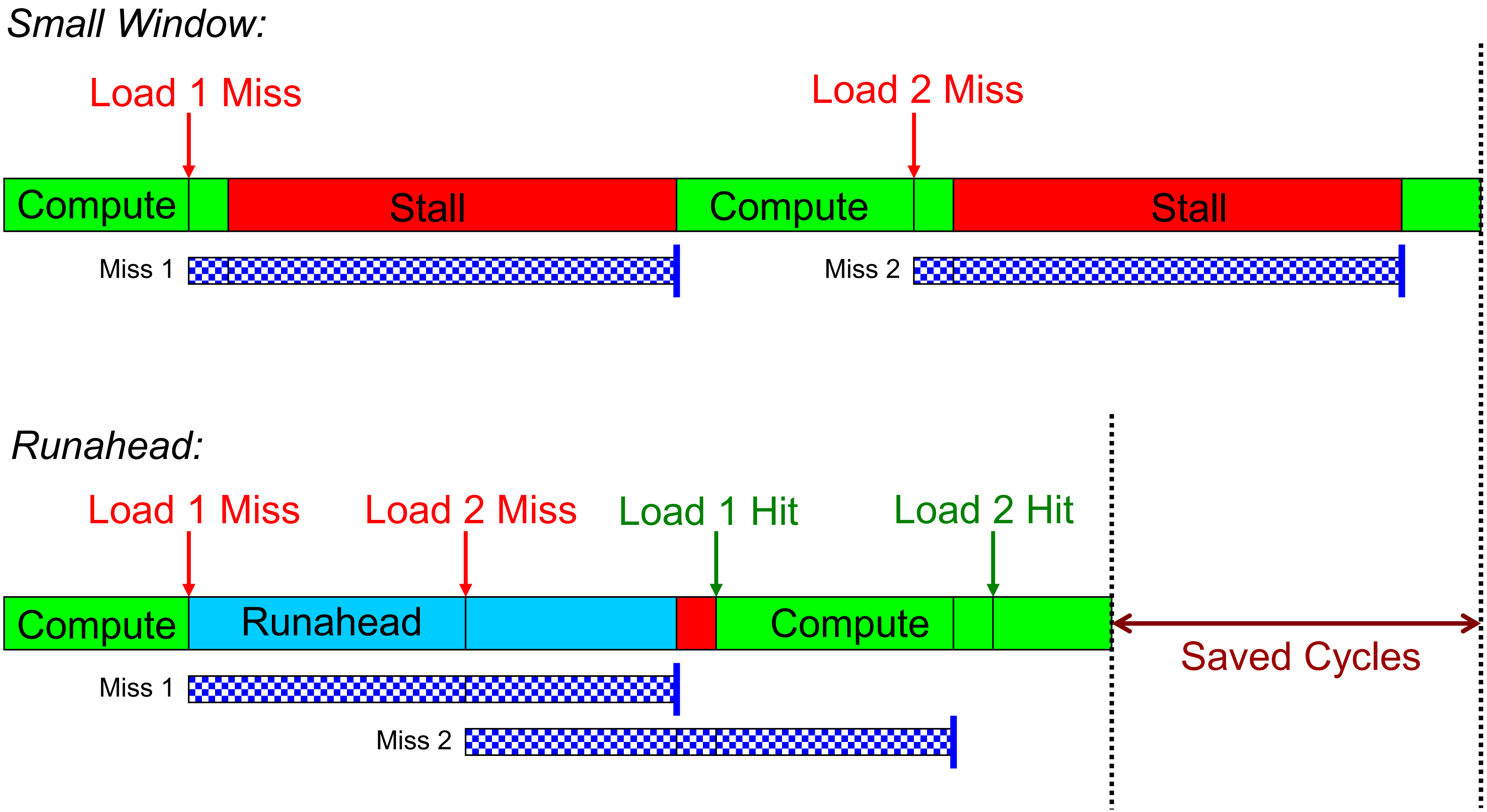
Review: Runahead Execution

- A simple pre-execution method for prefetching purposes

- When the oldest instruction is a long-latency cache miss:
  - Checkpoint architectural state and enter runahead mode
- In runahead mode:
  - Speculatively pre-execute instructions
  - The purpose of pre-execution is to generate prefetches
  - L2-miss dependent instructions are marked INV and dropped
- Runahead mode ends when the original miss returns
  - Checkpoint is restored and normal execution resumes

- Mutlu et al., “Runahead Execution: An Alternative to Very Large Instruction Windows for Out-of-order Processors,” HPCA 2003.

Review: Runahead Execution (Mutlu et al., HPCA 2003)



Runahead as an Execution-based Prefetcher

- Idea of an Execution-Based Prefetcher: Pre-execute a piece of the (pruned) program solely for prefetching data

- Idea of Runahead: Pre-execute the main program solely for prefetching data

- Advantages and disadvantages of runahead vs. other execution-based prefetchers?

- Can you make runahead even better by pruning the program portion executed in runahead mode?

Taking Advantage of Pure Speculation

- Runahead mode is purely speculative

- The goal is to find and generate cache misses that would otherwise stall execution later on

- How do we achieve this goal most efficiently and with the highest benefit?

- Idea: Find and execute only those instructions that will lead to cache misses (that cannot already be captured by the instruction window)

- How?

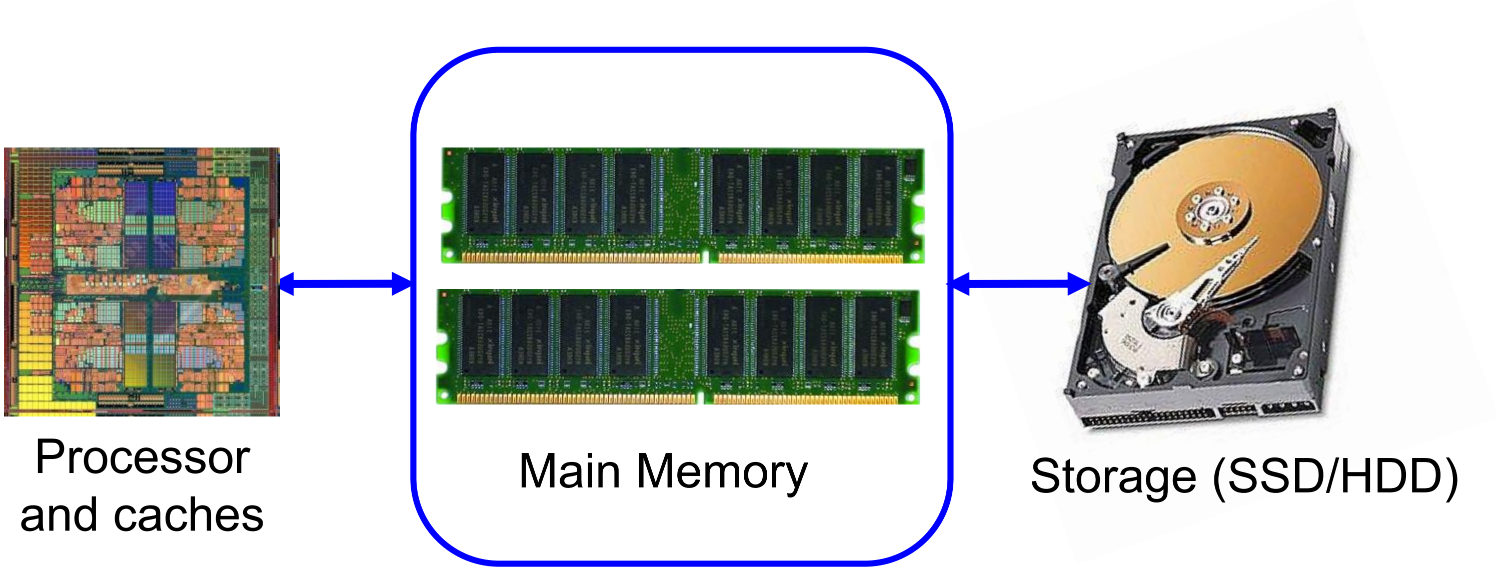
Where We Are in Lecture Schedule

- The memory hierarchy
- Caches, caches, more caches
- Virtualizing the memory hierarchy: Virtual Memory
- Main memory: DRAM
- Main memory control, scheduling
- Memory latency tolerance techniques
- Non-volatile memory

- Multiprocessors
- Coherence and consistency
- Interconnection networks
- Multi-core issues (e.g., heterogeneous multi-core)

Emerging Memory Technologies

The Main Memory System



Processor and caches

Main Memory

Storage (SSD/HDD)

- Main memory is a critical component of all computing systems: server, mobile, embedded, desktop, sensor

- Main memory system must scale (in *size*, *technology*, *efficiency*, *cost*, and *management algorithms*) to maintain performance growth and technology scaling benefits

Major Trends Affecting Main Memory (I)

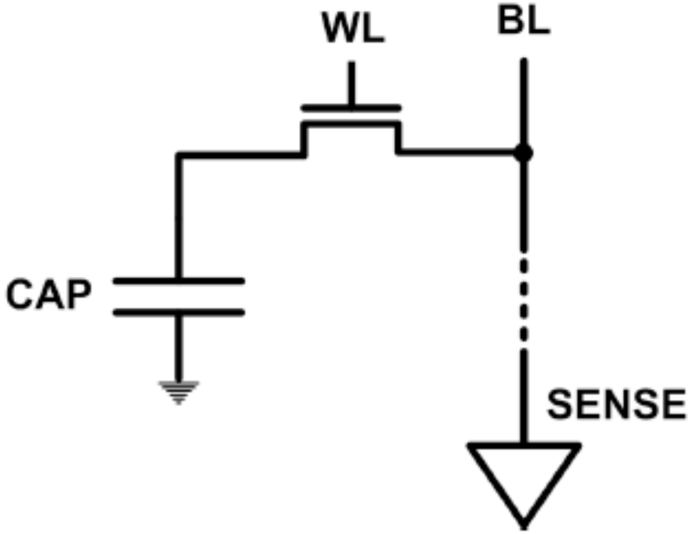
- Need for main memory capacity, bandwidth, QoS increasing

- Main memory energy/power is a key system design concern

- DRAM technology scaling is ending

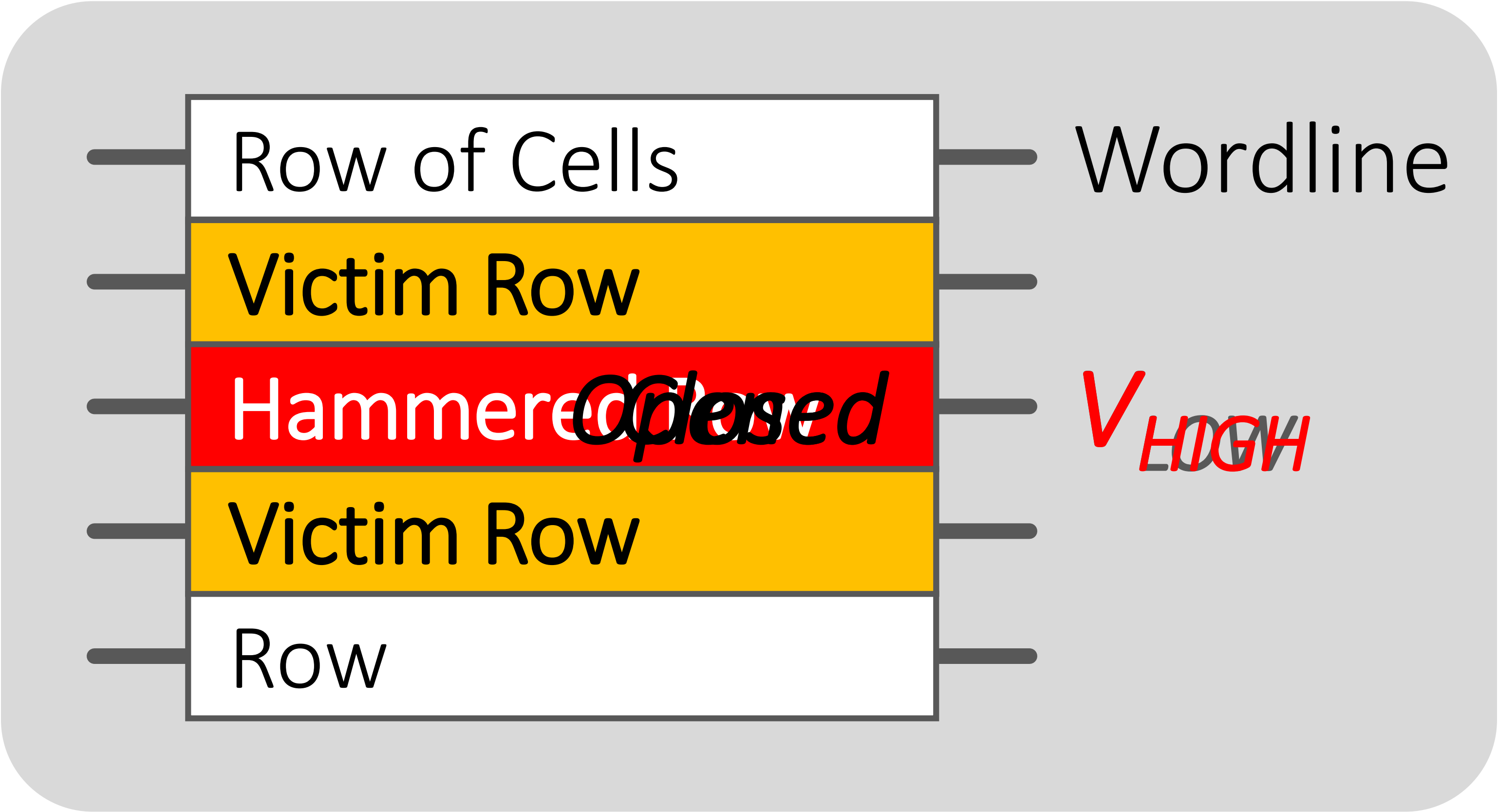
The DRAM Scaling Problem

- DRAM stores charge in a capacitor (charge-based memory)
  - Capacitor must be large enough for reliable sensing
  - Access transistor should be large enough for low leakage and high retention time
  - Scaling beyond 40-35nm (2013) is challenging [ITRS, 2009]



- DRAM capacity, cost, and energy/power hard to scale

An Example of The Scaling Problem



Row of Cells

Wordline

Victim Row

Hammered Row

Opened

Closed

V_{HIGH}

V_{LOW}

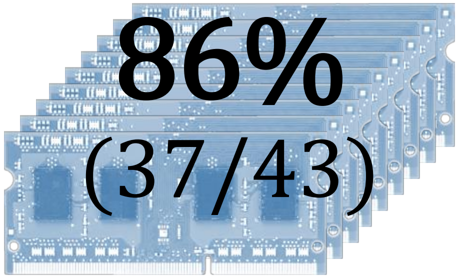
Victim Row

Row

Repeatedly opening and closing a row induces ***disturbance errors*** *in adjacent rows in* ***most real DRAM chips*** *[Kim+ ISCA 2014]*

Most DRAM Modules Are At Risk

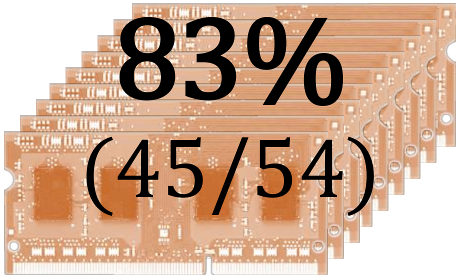
A company



86%
(37/43)

Up to
1.0×10^{7}
errors

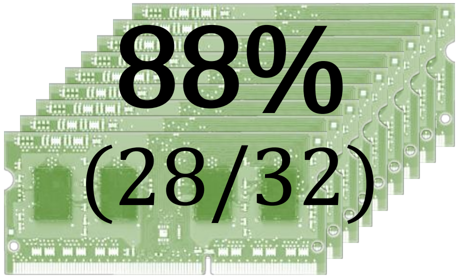
B company



83%
(45/54)

Up to
2.7×10^{6}
errors

C company

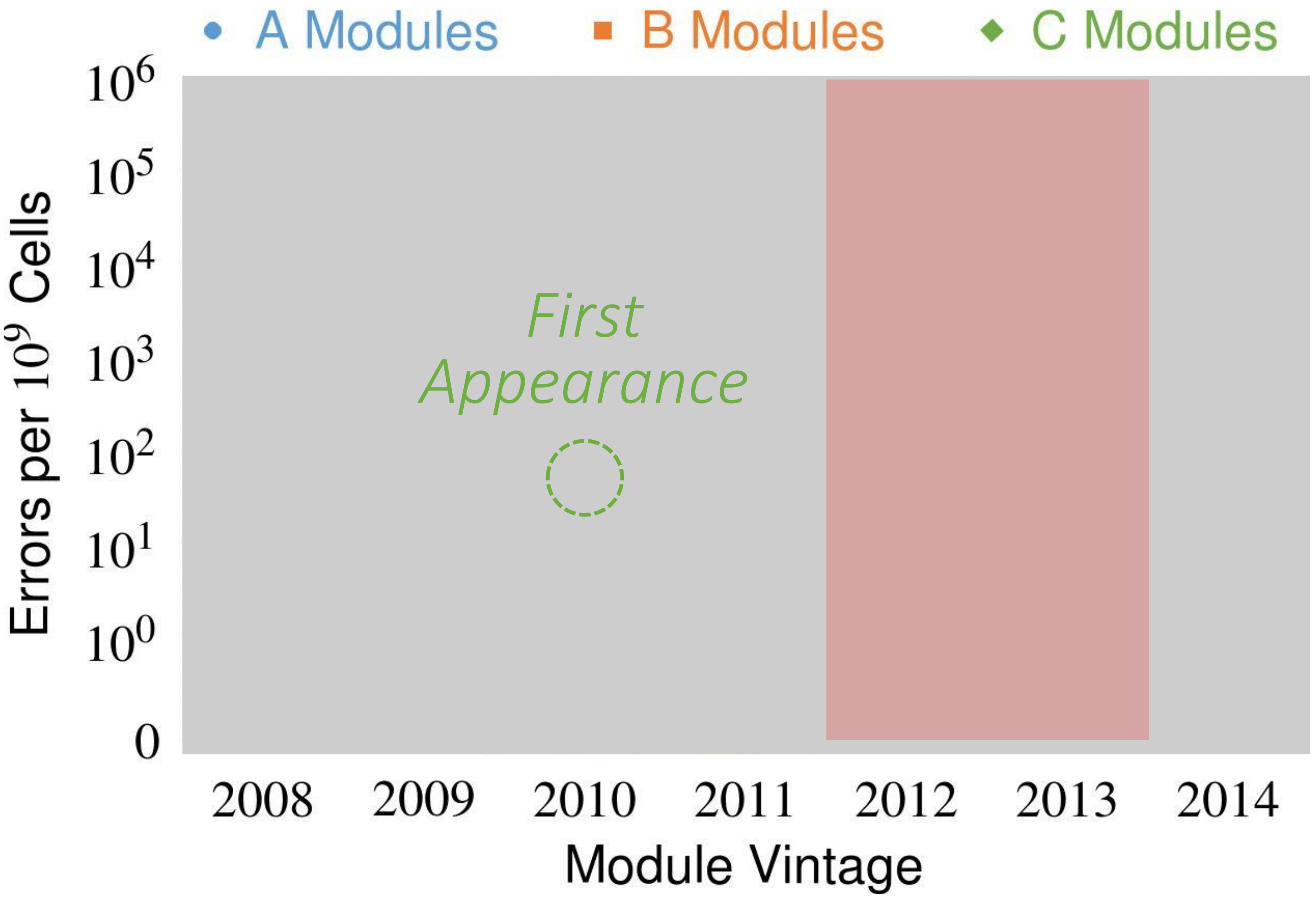


88%
(28/32)

Up to
3.3×10^{5}
errors

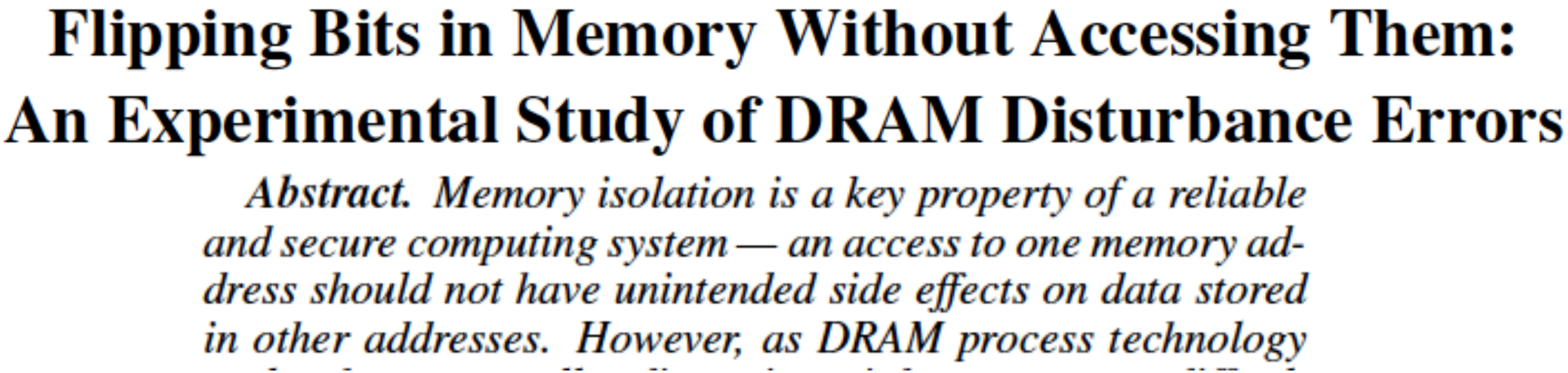
Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.

Errors *vs.* Vintage



All modules from 2012–2013 are vulnerable

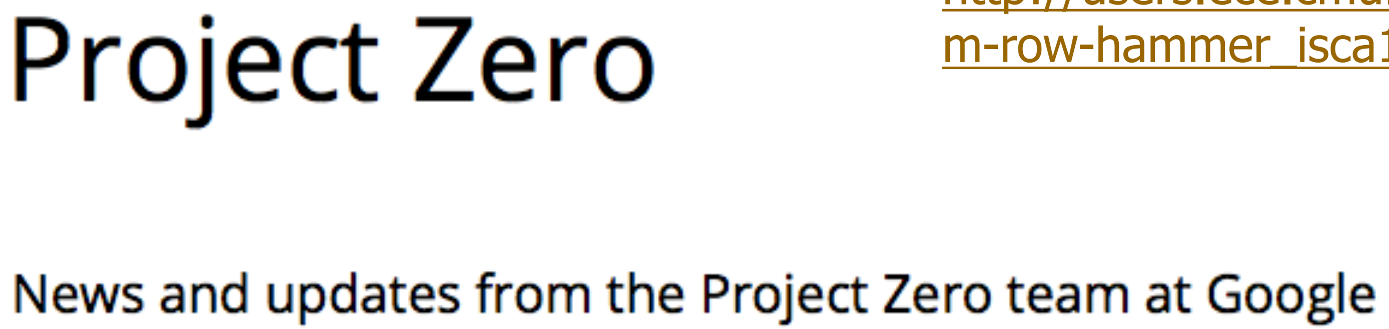
Security Implications



Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors

Abstract. Memory isolation is a key property of a reliable and secure computing system — an access to one memory address should not have unintended side effects on data stored in other addresses. However, as DRAM process technology

http://users.ece.cmu.edu/~omutlu/pub/dram-row-hammer_isca14.pdf

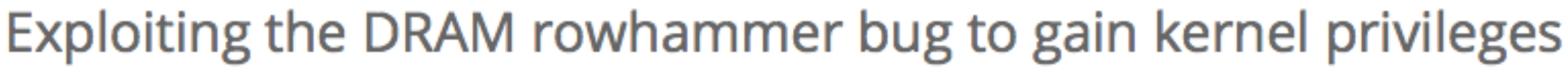


Project Zero

News and updates from the Project Zero team at Google

http://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html

Monday, March 9, 2015



Exploiting the DRAM rowhammer bug to gain kernel privileges

Security Implications

How Can We Fix the Memory Problem & Design (Memory) Systems of the Future?

How Do We Solve The Problem?

- Tolerate it: Make DRAM and controllers more intelligent
  - New interfaces, functions, architectures: system-DRAM codesign

- Eliminate or minimize it: Replace or (more likely) augment DRAM with a different technology
  - New technologies and system-wide rethinking of memory & storage

- Embrace it: Design heterogeneous-reliability memories that map error-tolerant data to less reliable portions
  - New usage and execution models

Solutions (to memory scaling) require software/hardware/device cooperation

Trends: Problems with DRAM as Main Memory

- Need for main memory capacity increasing
  - DRAM capacity hard to scale

- Main memory energy/power is a key system design concern
  - DRAM consumes high power due to leakage and refresh

- DRAM technology scaling is ending
  - DRAM capacity, cost, and energy/power hard to scale

Solutions to the DRAM Scaling Problem

- Two potential solutions
  - Tolerate DRAM (by taking a fresh look at it)
  - Enable emerging memory technologies to eliminate/minimize DRAM

- Do both
  - Hybrid memory systems

Solution 1: Tolerate DRAM

- Overcome DRAM shortcomings with
  - System-DRAM co-design
  - Novel DRAM architectures, interface, functions
  - Better waste management (efficient utilization)

- Key issues to tackle
  - Reduce energy
  - Enable reliability at low cost
  - Improve bandwidth and latency
  - Reduce waste
  - Enable computation close to data

Solution 1: Tolerate DRAM

- Liu+, "RAIDR: Retention-Aware Intelligent DRAM Refresh," ISCA 2012.
- Kim+, "A Case for Exploiting Subarray-Level Parallelism in DRAM," ISCA 2012.
- Lee+, "Tiered-Latency DRAM: A Low Latency and Low Cost DRAM Architecture," HPCA 2013.
- Liu+, "An Experimental Study of Data Retention Behavior in Modern DRAM Devices," ISCA 2013.
- Seshadri+, "RowClone: Fast and Efficient In-DRAM Copy and Initialization of Bulk Data," MICRO 2013.
- Pekhimenko+, "Linearly Compressed Pages: A Main Memory Compression Framework," MICRO 2013.
- Chang+, "Improving DRAM Performance by Parallelizing Refreshes with Accesses," HPCA 2014.
- Khan+, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study," SIGMETRICS 2014.
- Luo+, "Characterizing Application Memory Error Vulnerability to Optimize Data Center Cost," DSN 2014.
- Kim+, "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors," ISCA 2014.
- Lee+, "Adaptive-Latency DRAM: Optimizing DRAM Timing for the Common-Case," HPCA 2015.
- Qureshi+, "AVATAR: A Variable-Retention-Time (VRT) Aware Refresh for DRAM Systems," DSN 2015.
- Meza+, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," DSN 2015.
- Kim+, "Ramulator: A Fast and Extensible DRAM Simulator," IEEE CAL 2015.

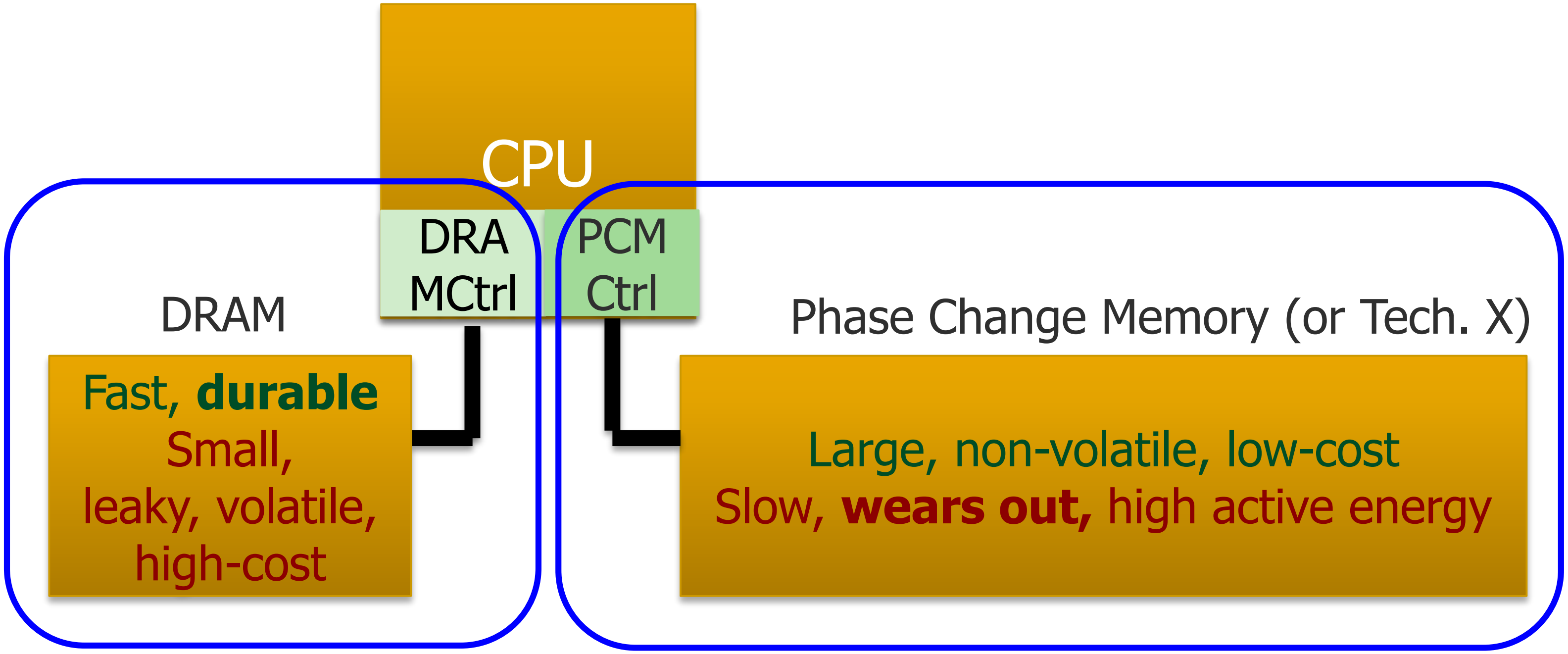
- Avoid DRAM:
  - Seshadri+, "The Evicted-Address Filter: A Unified Mechanism to Address Both Cache Pollution and Thrashing," PACT 2012.
  - Pekhimenko+, "Base-Delta-Immediate Compression: Practical Data Compression for On-Chip Caches," PACT 2012.
  - Seshadri+, "The Dirty-Block Index," ISCA 2014.
  - Pekhimenko+, "Exploiting Compressed Block Size as an Indicator of Future Reuse," HPCA 2015.

Solution 2: Emerging Memory Technologies

- Some emerging resistive memory technologies seem more scalable than DRAM (and they are non-volatile)
- Example: Phase Change Memory
  - Expected to scale to 9nm (2022 [ITRS])
  - Expected to be denser than DRAM: can store multiple bits/cell
- But, emerging technologies have shortcomings as well
  - Can they be enabled to replace/augment/surpass DRAM?

- Lee+, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA'09, CACM'10, Micro'10.
- Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters 2012.
- Yoon, Meza+, "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012.
- Kultursay+, "Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative," ISPASS 2013.
- Meza+, "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory," WEED 2013.
- Lu+, "Loose Ordering Consistency for Persistent Memory," ICCD 2014.
- Zhao+, "FIRM: Fair and High-Performance Memory Control for Persistent Memory Systems," MICRO 2014.
- Yoon, Meza+, "Efficient Data Mapping and Buffering Techniques for Multi-Level Cell Phase-Change Memories," ACM TACO 2014.

Hybrid Memory Systems


CPU
DRA MCtrl
PCM Ctrl
DRAM
Fast, **durable**
Small,
leaky, volatile,
high-cost
Phase Change Memory (or Tech. X)
Large, non-volatile, low-cost
Slow, **wears out,** high active energy

Hardware/software manage data allocation and movement
to achieve the best of multiple technologies

Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Yoon, Meza et al., "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012 Best Paper Award.

Requirements from an Ideal Memory System

- Traditional
  - Higher capacity
  - Continuous low cost
  - High system performance (higher bandwidth, low latency)

- New
  - Technology scalability: lower cost, higher capacity, lower energy
  - Energy (and power) efficiency
  - QoS support and configurability (for consolidation)

Emerging, resistive memory technologies (NVM) can help

The Promise of Emerging Technologies

- Likely need to replace/augment DRAM with a technology that is
  - Technology scalable
  - And at least similarly efficient, high performance, and fault-tolerant
    - or can be architected to be so

- Some emerging resistive memory technologies appear promising
  - Phase Change Memory (PCM)?
  - Spin Torque Transfer Magnetic Memory (STT-MRAM)?
  - Memristors?
  - And, maybe there are other ones
  - Can they be enabled to replace/augment/surpass DRAM?

Charge vs. Resistive Memories

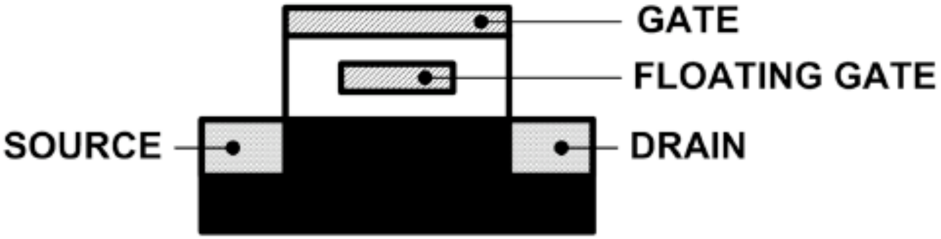
- Charge Memory (e.g., DRAM, Flash)
  - Write data by capturing charge Q
  - Read data by detecting voltage V

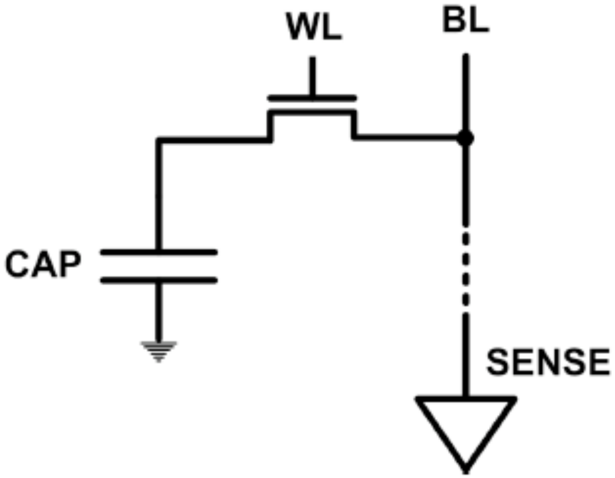
- Resistive Memory (e.g., PCM, STT-MRAM, memristors)
  - Write data by pulsing current dQ/dt
  - Read data by detecting resistance R

Limits of Charge Memory

- Difficult charge placement and control
  - Flash: floating gate charge
  - DRAM: capacitor charge, transistor leakage

- Reliable sensing becomes difficult as charge storage unit size reduces





Emerging Resistive Memory Technologies

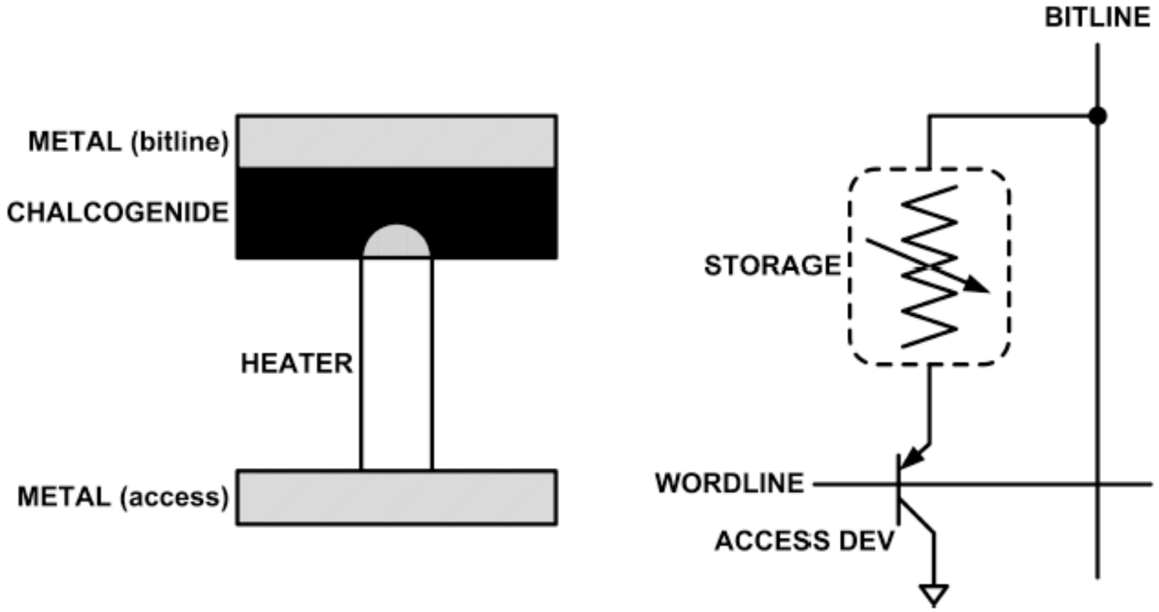
- PCM
  - Inject current to change material phase
  - Resistance determined by phase

- STT-MRAM
  - Inject current to change magnet polarity
  - Resistance determined by polarity

- Memristors/RRAM/ReRAM
  - Inject current to change atomic structure
  - Resistance determined by atom distance

What is Phase Change Memory?

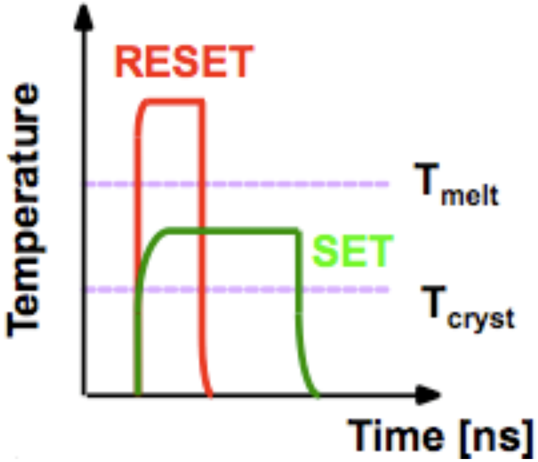
- Phase change material (chalcogenide glass) exists in two states:
  - Amorphous: Low optical reflexivity and high electrical resistivity
  - Crystalline: High optical reflexivity and low electrical resistivity

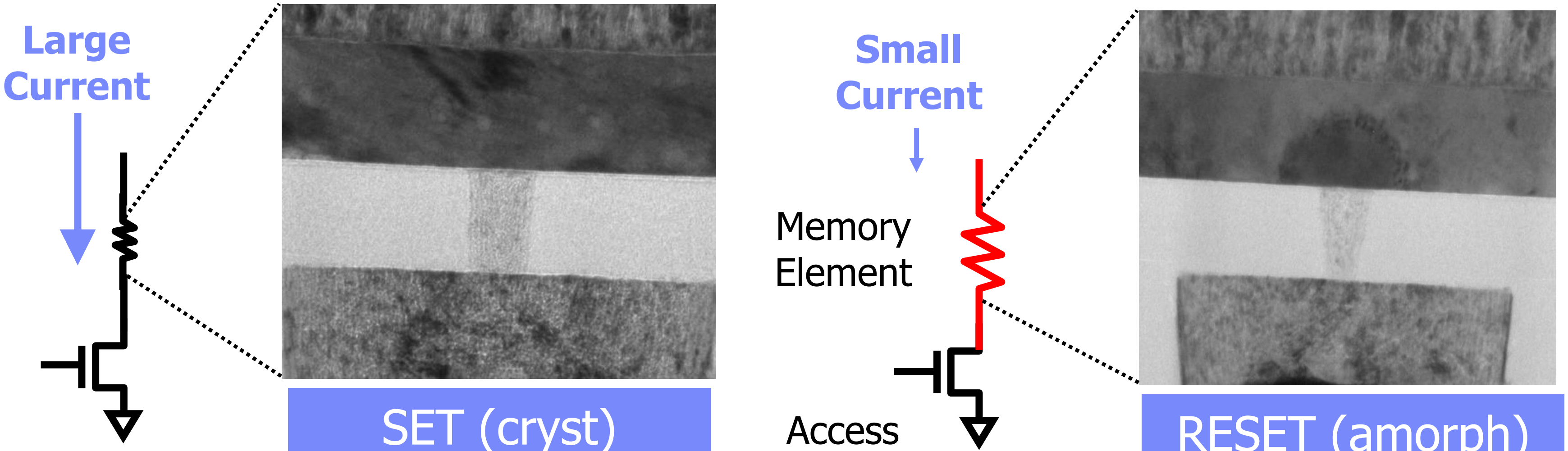


PCM is resistive memory: High resistance (0), Low resistance (1)
PCM cell can be switched between states reliably and quickly

How Does PCM Work?

- Write: change phase via current injection
  - SET: sustained current to heat cell above T*cryst*
  - RESET: cell heated above T*melt* and quenched
- Read: detect phase via material resistance
  - Amorphous vs. crystalline





SET (cryst)
Low resistance

10^3-10^4 Ω

RESET (amorph)
High resistance

10^6-10^7 Ω

Photo Courtesy: Bipin Rajendran, IBM   Slide Courtesy: Moinuddin Qureshi, IBM

Opportunity: PCM Advantages

- Scales better than DRAM, Flash
  - Requires current pulses, which scale linearly with feature size
  - Expected to scale to 9nm (2022 [ITRS])
  - Prototyped at 20nm (Raoux+, IBM JRD 2008)
- Can be denser than DRAM
  - Can store multiple bits per cell due to large resistance range
  - Prototypes with 2 bits/cell in ISSCC' 08, 4 bits/cell by 2012
- Non-volatile
  - Retain data for >10 years at 85C
- No refresh needed, low idle power

Phase Change Memory Properties

- Surveyed prototypes from 2003-2008 (ITRS, IEDM, VLSI, ISSCC)
- Derived PCM parameters for F=90nm

- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

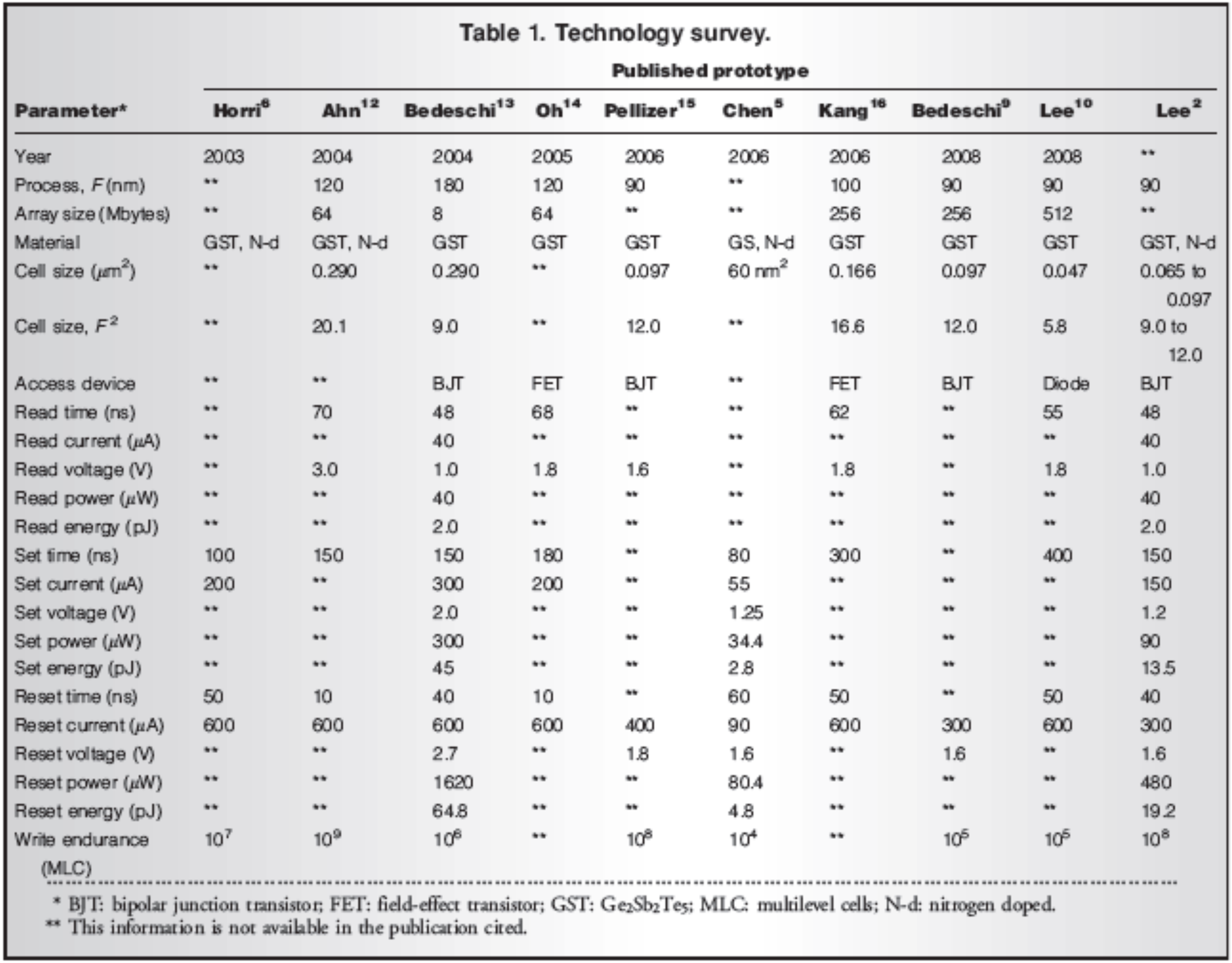


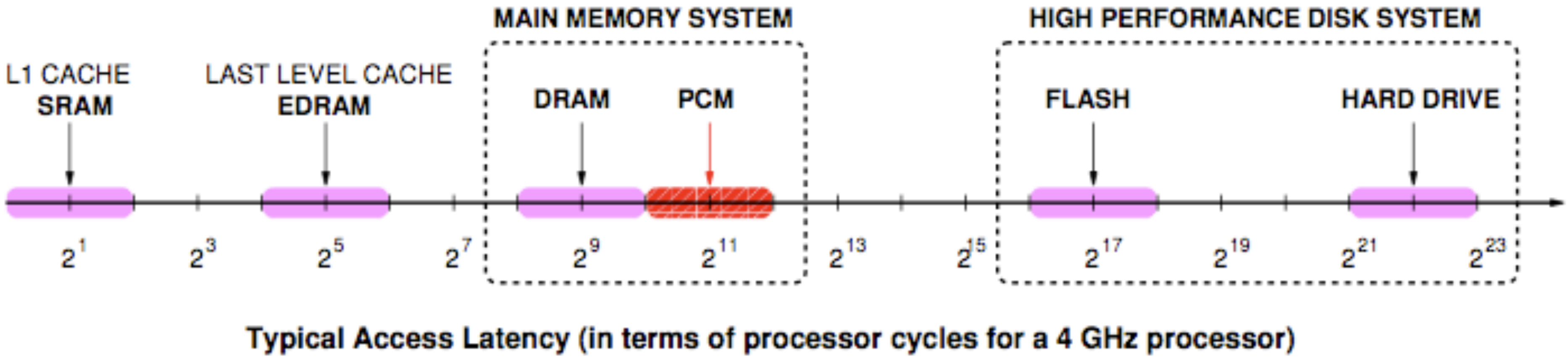
Table 1. Technology survey.

	Published prototype									
Parameter*	Horri[6]	Ahn[12]	Bedeschi[13]	Oh[14]	Pellizer[15]	Chen[5]	Kang[16]	Bedeschi[9]	Lee[10]	Lee[2]
Year	2003	2004	2004	2005	2006	2006	2006	2008	2008	**
Process, F (nm)	**	120	180	120	90	**	100	90	90	90
Array size (Mbytes)	**	64	8	64	**	**	256	256	512	**
Material	GST, N-d	GST, N-d	GST	GST	GST	GS, N-d	GST	GST	GST	GST, N-d
Cell size (μm^2)	**	0.290	0.290	**	0.097	60 nm^2	0.166	0.097	0.047	0.065 to 0.097
Cell size, F^2	**	20.1	9.0	**	12.0	**	16.6	12.0	5.8	9.0 to 12.0
Access device	**	**	BJT	FET	BJT	**	FET	BJT	Diode	BJT
Read time (ns)	**	70	48	68	**	**	62	**	55	48
Read current (μA)	**	**	40	**	**	**	**	**	**	40
Read voltage (V)	**	3.0	1.0	1.8	1.6	**	1.8	**	1.8	1.0
Read power (μW)	**	**	40	**	**	**	**	**	**	40
Read energy (pJ)	**	**	2.0	**	**	**	**	**	**	2.0
Set time (ns)	100	150	150	180	**	80	300	**	400	150
Set current (μA)	200	**	300	200	**	55	**	**	**	150
Set voltage (V)	**	**	2.0	**	**	1.25	**	**	**	1.2
Set power (μW)	**	**	300	**	**	34.4	**	**	**	90
Set energy (pJ)	**	**	45	**	**	2.8	**	**	**	13.5
Reset time (ns)	50	10	40	10	**	60	50	**	50	40
Reset current (μA)	600	600	600	600	400	90	600	300	600	300
Reset voltage (V)	**	**	2.7	**	1.8	1.6	**	1.6	**	1.6
Reset power (μW)	**	**	1620	**	**	80.4	**	**	**	480
Reset energy (pJ)	**	**	64.8	**	**	4.8	**	**	**	19.2
Write endurance (MLC)	10^7	10^9	10^6	**	10^8	10^4	**	10^5	10^5	10^8

* BJT: bipolar junction transistor; FET: field-effect transistor; GST: $Ge_2Sb_2Te_5$; MLC: multilevel cells; N-d: nitrogen doped.
** This information is not available in the publication cited.

Phase Change Memory Properties: Latency

- Latency comparable to, but slower than DRAM



- Read Latency
  - 50ns: 4x DRAM, 10^{-3}x NAND Flash
- Write Latency
  - 150ns: 12x DRAM
- Write Bandwidth
  - 5-10 MB/s: 0.1x DRAM, 1x NAND Flash

Phase Change Memory Properties

- Dynamic Energy
  - 40 uA Rd, 150 uA Wr
  - 2-43x DRAM, 1x NAND Flash
- Endurance
  - Writes induce phase change at 650C
  - Contacts degrade from thermal expansion/contraction
  - 10^8 writes per cell
  - 10^{-8}x DRAM, 10^3x NAND Flash
- Cell Size
  - 9-12F^2 using BJT, single-level cells
  - 1.5x DRAM, 2-3x NAND (will scale with feature size)

Phase Change Memory: Pros and Cons

- Pros over DRAM
  - Better technology scaling
  - Non volatility
  - Low idle power (no refresh)
- Cons
  - Higher latencies: ~4-15x DRAM (especially write)
  - Higher active energy: ~2-50x DRAM (especially write)
  - Lower endurance (a cell dies after ~10^8 writes)
  - Reliability issues (resistance drift)
- Challenges in enabling PCM as DRAM replacement/helper:
  - Mitigate PCM shortcomings
  - Find the right way to place PCM in the system
  - Ensure secure and fault-tolerant PCM operation

PCM-based Main Memory: Some Questions

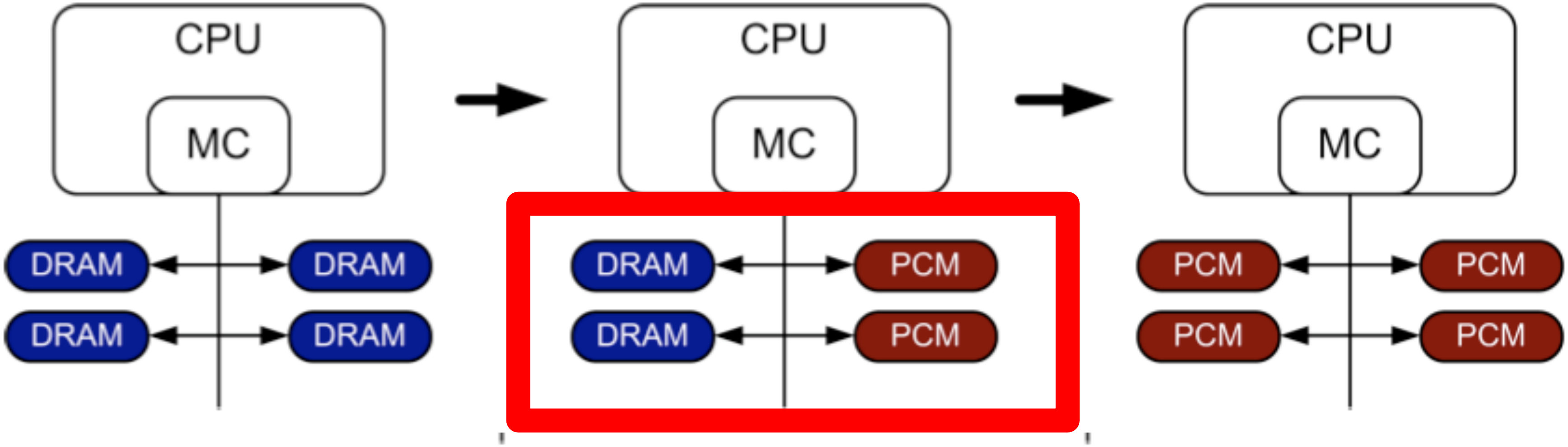
- Where to place PCM in the memory hierarchy?
  - Hybrid OS controlled PCM-DRAM
  - Hybrid OS controlled PCM and hardware-controlled DRAM
  - Pure PCM main memory

- How to mitigate shortcomings of PCM?

- How to take advantage of (byte-addressable and fast) non-volatile main memory?

PCM-based Main Memory (I)

- How should PCM-based (main) memory be organized?



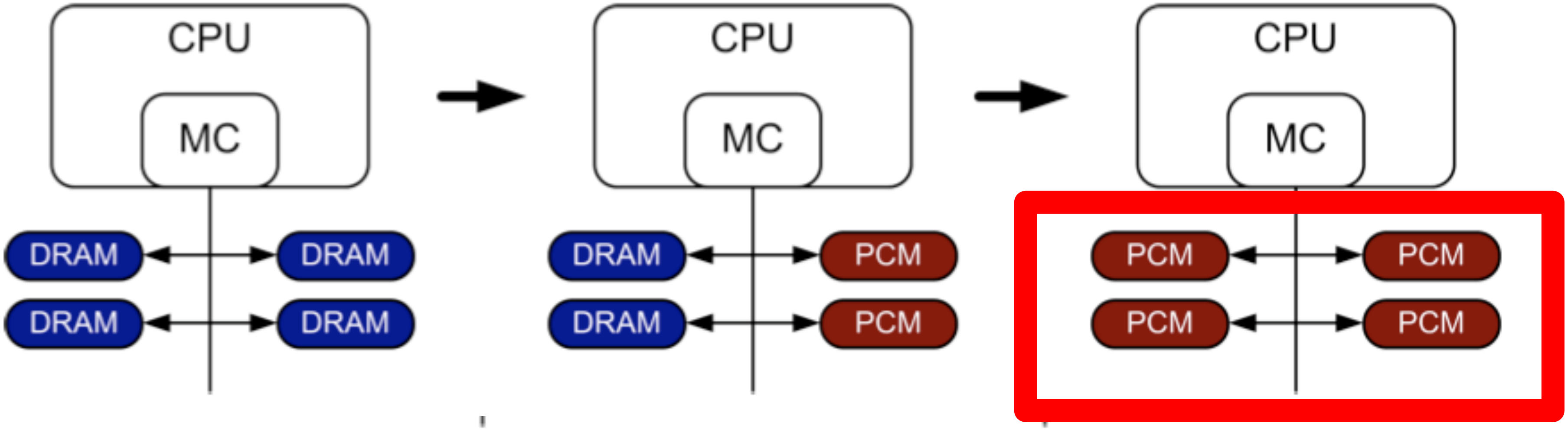
- Hybrid PCM+DRAM [Qureshi+ ISCA'09, Dhiman+ DAC'09, Meza+ IEEE CAL'12]:
  - How to partition/migrate data between PCM and DRAM

Hybrid Memory Systems: Challenges

- Partitioning
  - Should DRAM be a cache or main memory, or configurable?
  - What fraction? How many controllers?
- Data allocation/movement (energy, performance, lifetime)
  - Who manages allocation/movement?
  - What are good control algorithms?
  - How do we prevent degradation of service due to wearout?
- Design of cache hierarchy, memory controllers, OS
  - Mitigate PCM shortcomings, exploit PCM advantages
- Design of PCM/DRAM chips and modules
  - Rethink the design of PCM/DRAM with new requirements

PCM-based Main Memory (II)

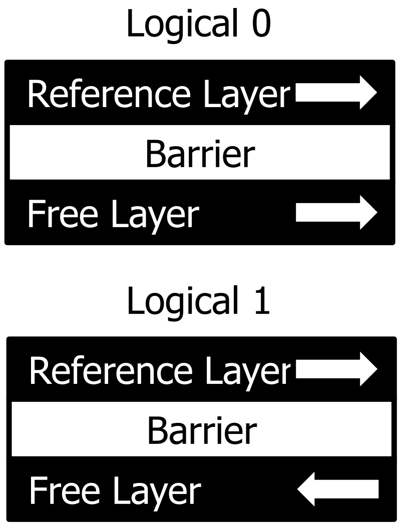
- How should PCM-based (main) memory be organized?

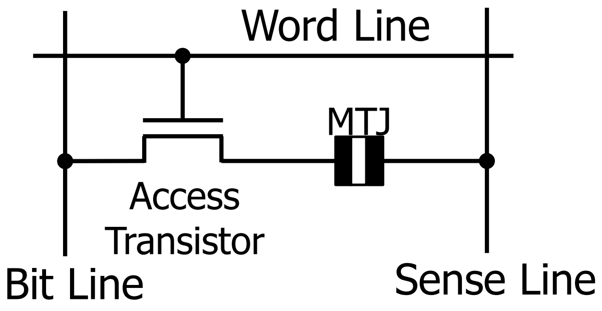


- Pure PCM main memory [Lee et al., ISCA'09, Top Picks'10]:
  - How to redesign entire hierarchy (and cores) to overcome PCM shortcomings

Aside: STT-RAM Basics

- Magnetic Tunnel Junction (MTJ)
  - Reference layer: Fixed
  - Free layer: Parallel or anti-parallel
- Cell
  - Access transistor, bit/sense lines
- Read and Write
  - Read: Apply a small voltage across bitline and senseline; read the current.
  - Write: Push large current through MTJ. Direction of current determines new orientation of the free layer.





- Kultursay et al., “Evaluating STT-RAM as an Energy-Efficient Main Memory Alternative,” ISPASS 2013

Aside: STT MRAM: Pros and Cons

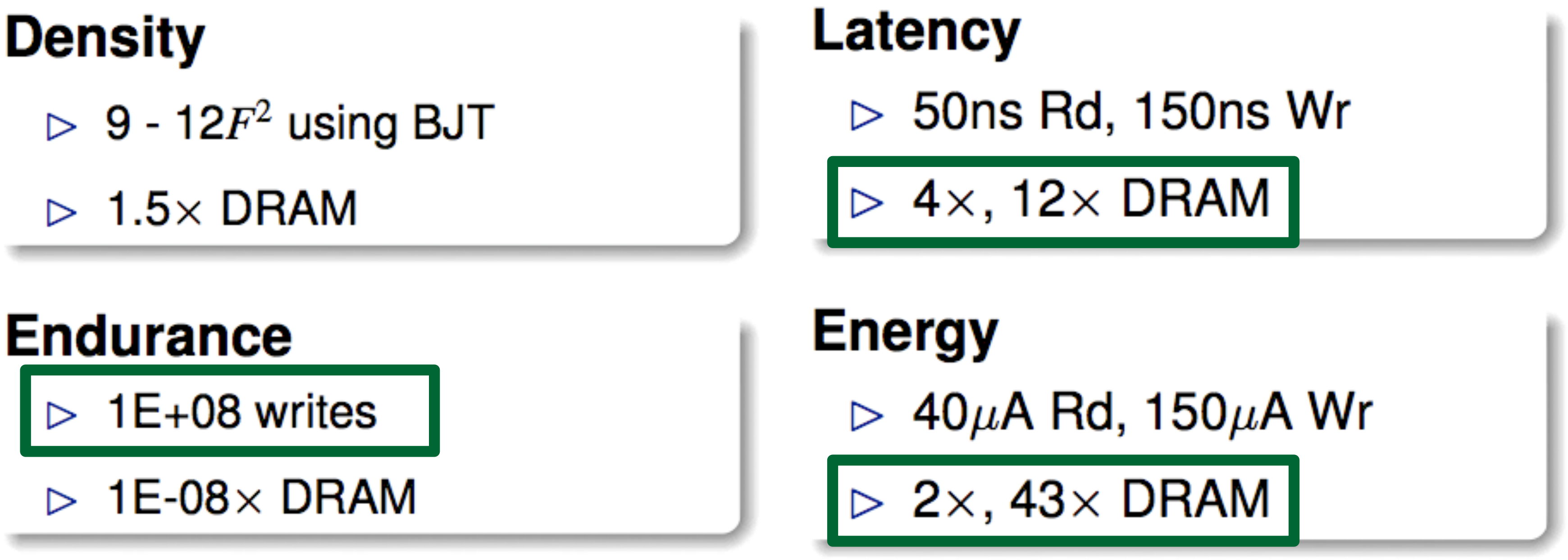
- Pros over DRAM
  - Better technology scaling
  - Non volatility
  - Low idle power (no refresh)

- Cons
  - Higher write latency
  - Higher write energy
  - Reliability?

- Another level of freedom
  - Can trade off non-volatility for lower write latency/energy (by reducing the size of the MTJ)

An Initial Study: Replace DRAM with PCM

- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.
  - Surveyed prototypes from 2003-2008 (e.g. IEDM, VLSI, ISSCC)
  - Derived "average" PCM parameters for F=90nm



Density

▷ 9 - 12F^2 using BJT

▷ 1.5× DRAM

Latency

▷ 50ns Rd, 150ns Wr

▷ 4×, 12× DRAM

Endurance

▷ 1E+08 writes

▷ 1E-08× DRAM

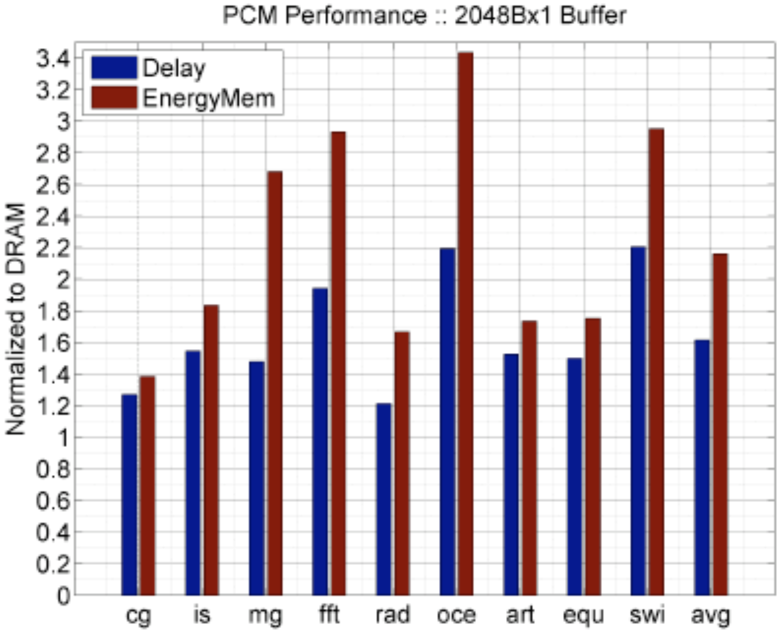
Energy

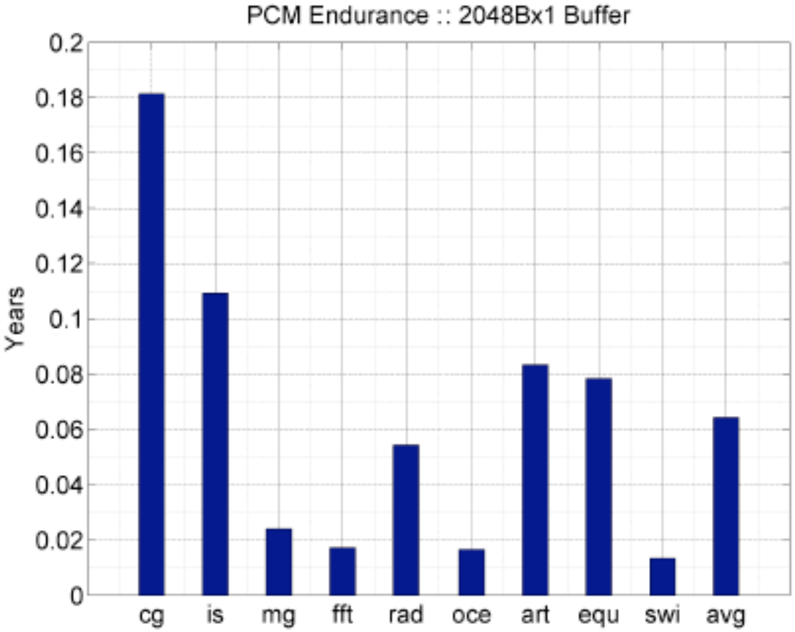
▷ 40μA Rd, 150μA Wr

▷ 2×, 43× DRAM

Results: Naïve Replacement of DRAM with PCM

- Replace DRAM with PCM in a 4-core, 4MB L2 system
- PCM organized the same as DRAM: row buffers, banks, peripherals
- 1.6x delay, 2.2x energy, 500-hour average lifetime



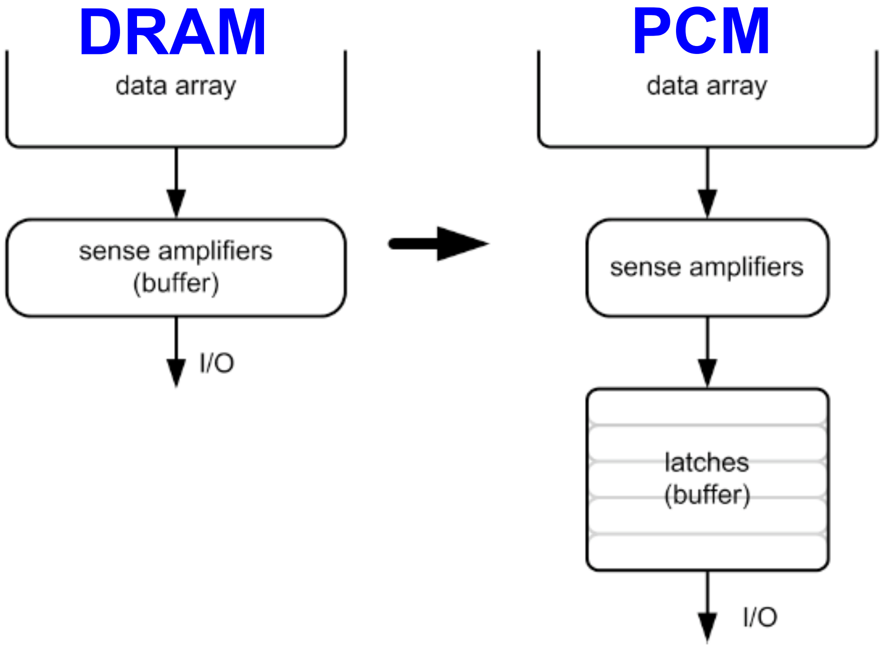


- Lee, Ipek, Mutlu, Burger, "Architecting Phase Change Memory as a Scalable DRAM Alternative," ISCA 2009.

Architecting PCM to Mitigate Shortcomings

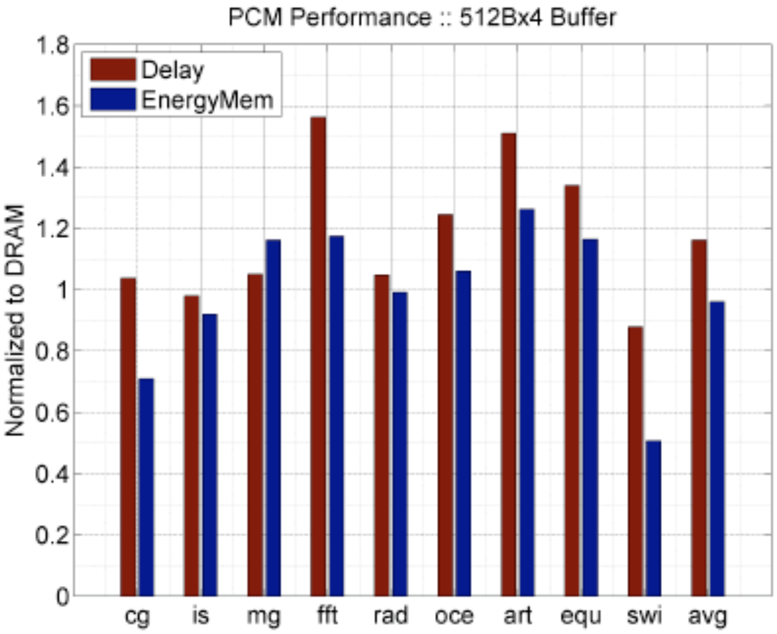
- Idea 1: Use multiple narrow row buffers in each PCM chip
  → Reduces array reads/writes → better endurance, latency, energy

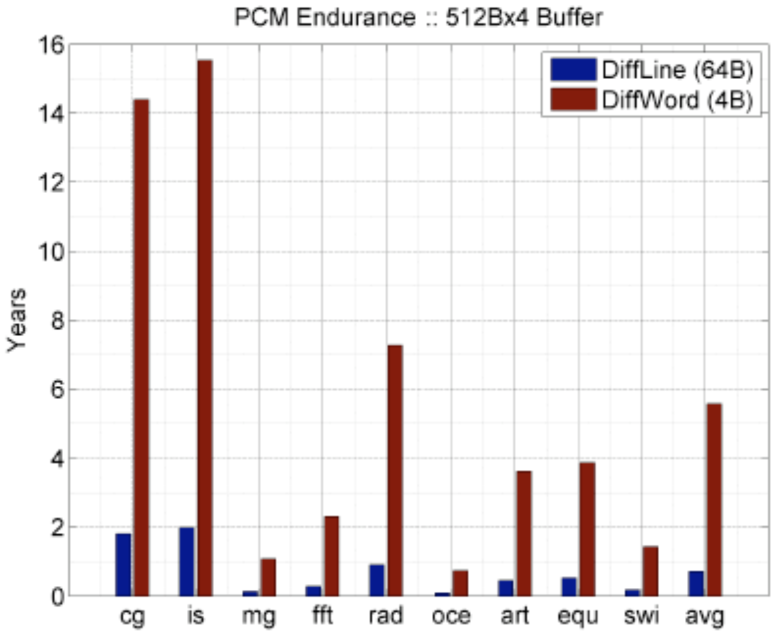
- Idea 2: Write into array at cache block or word granularity
  → Reduces unnecessary wear



Results: Architected PCM as Main Memory

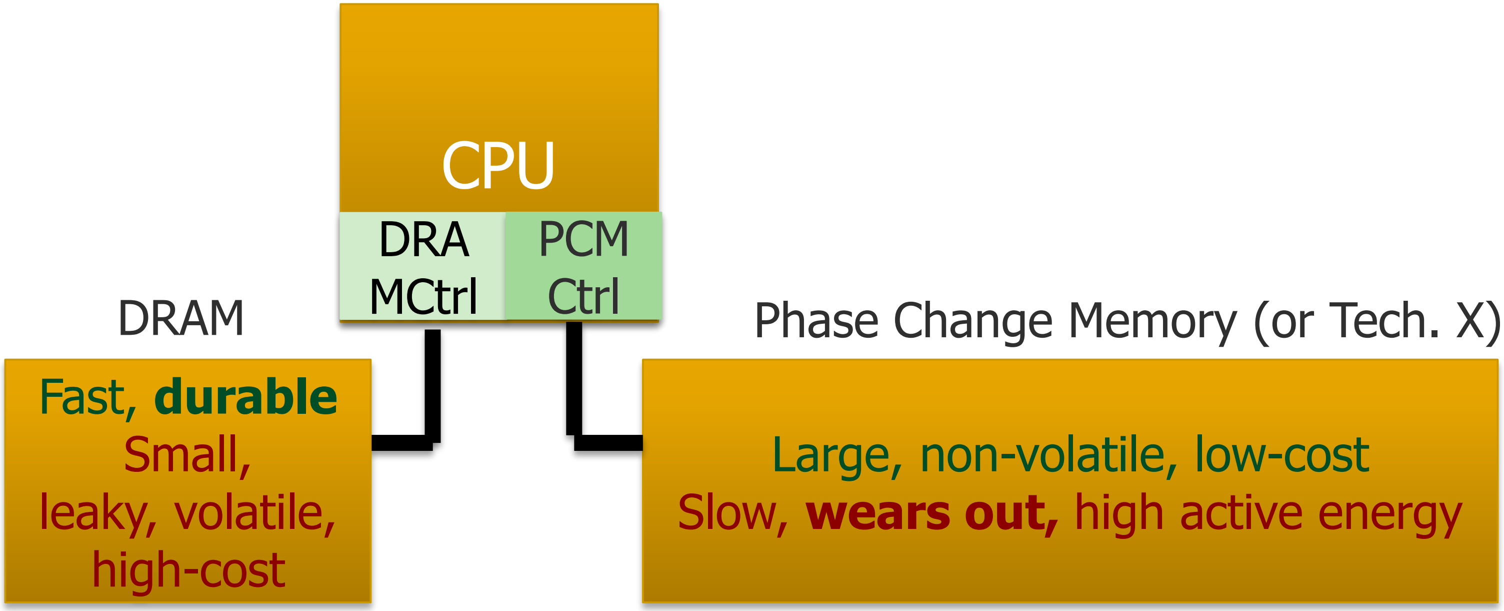
- 1.2x delay, 1.0x energy, 5.6-year average lifetime
- Scaling improves energy, endurance, density





- Caveat 1: Worst-case lifetime is much shorter (no guarantees)
- Caveat 2: Intensive applications see large performance and energy hits
- Caveat 3: Optimistic PCM parameters?

Hybrid Memory Systems


CPU
DRA MCtrl
PCM Ctrl
DRAM
Phase Change Memory (or Tech. X)
Fast, **durable**
Small,
leaky, volatile,
high-cost
Large, non-volatile, low-cost
Slow, **wears out,** high active energy

Hardware/software manage data allocation and movement
to achieve the best of multiple technologies

Meza+, "Enabling Efficient and Scalable Hybrid Memories," IEEE Comp. Arch. Letters, 2012.

Yoon, Meza et al., "Row Buffer Locality Aware Caching Policies for Hybrid Memories," ICCD 2012 Best Paper Award.

One Option: DRAM as a Cache for PCM

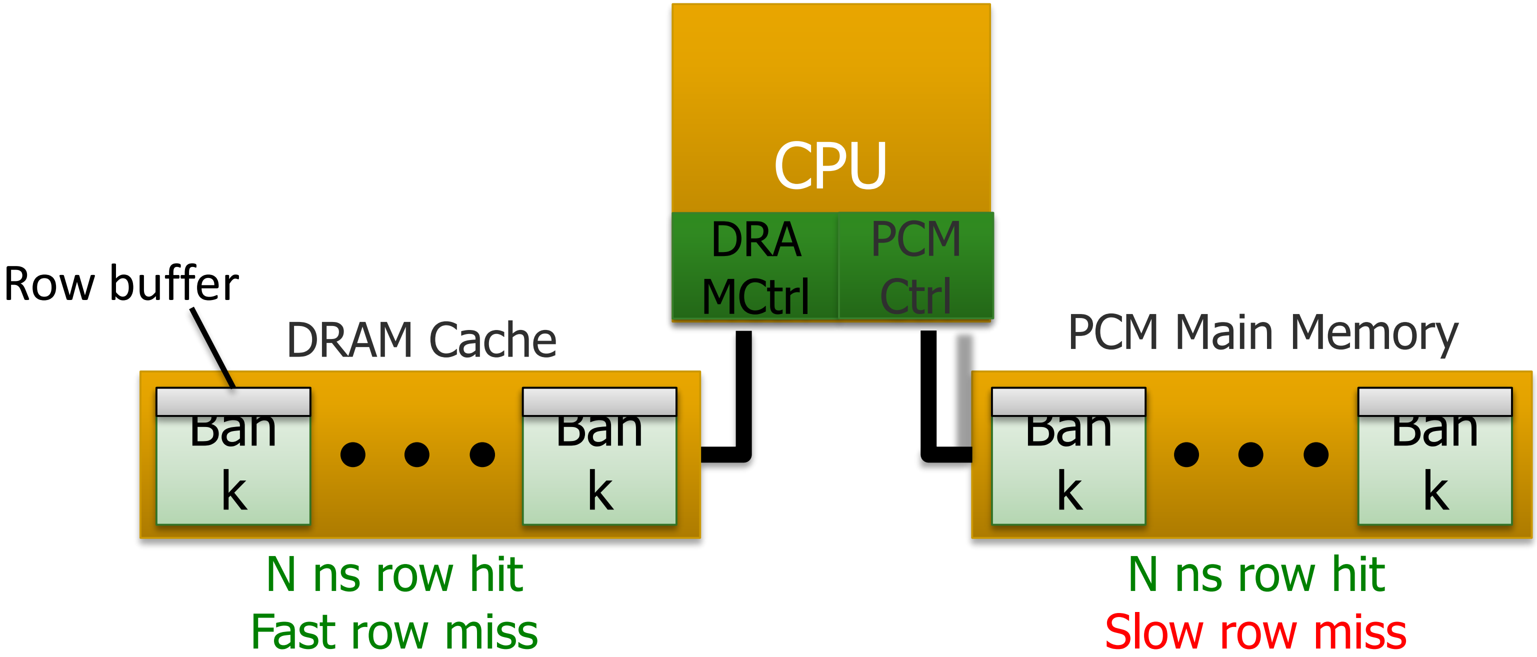
- PCM is main memory; DRAM caches memory rows/blocks
  - Benefits: Reduced latency on DRAM cache hit; write filtering
- Memory controller hardware manages the DRAM cache
  - Benefit: Eliminates system software overhead

- Three issues:
  - What data should be placed in DRAM versus kept in PCM?
  - What is the granularity of data movement?
  - How to design a huge (DRAM) cache at low cost?

- Two solutions:
  - Locality-aware data placement **[Yoon+ , ICCD 2012]**
  - Cheap tag stores and dynamic granularity **[Meza+, IEEE CAL 2012]**

DRAM vs. PCM: An Observation

- Row buffers are the same in DRAM and PCM
- Row buffer hit latency **same** in DRAM and PCM
- Row buffer miss latency **small** in DRAM, **large** in PCM



- Accessing the row buffer in PCM is fast
- What incurs high latency is the PCM array access → avoid this

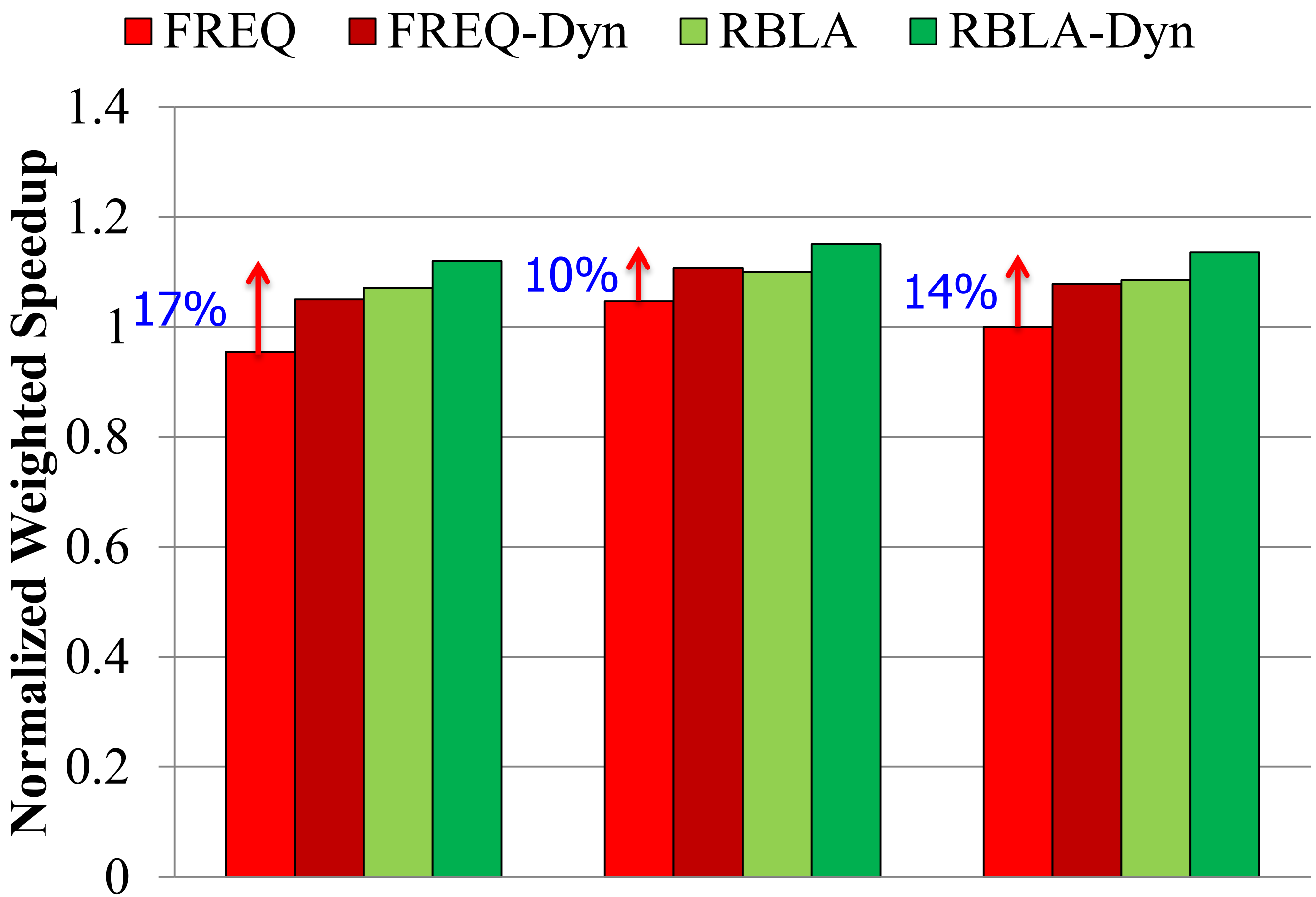
Row-Locality-Aware Data Placement

- Idea: Cache in DRAM only those rows that
  - Frequently cause row buffer conflicts → because row-conflict latency is smaller in DRAM
  - Are reused many times → to reduce cache pollution and bandwidth waste

- Simplified rule of thumb:
  - Streaming accesses: Better to place in PCM
  - Other accesses (with some reuse): Better to place in DRAM

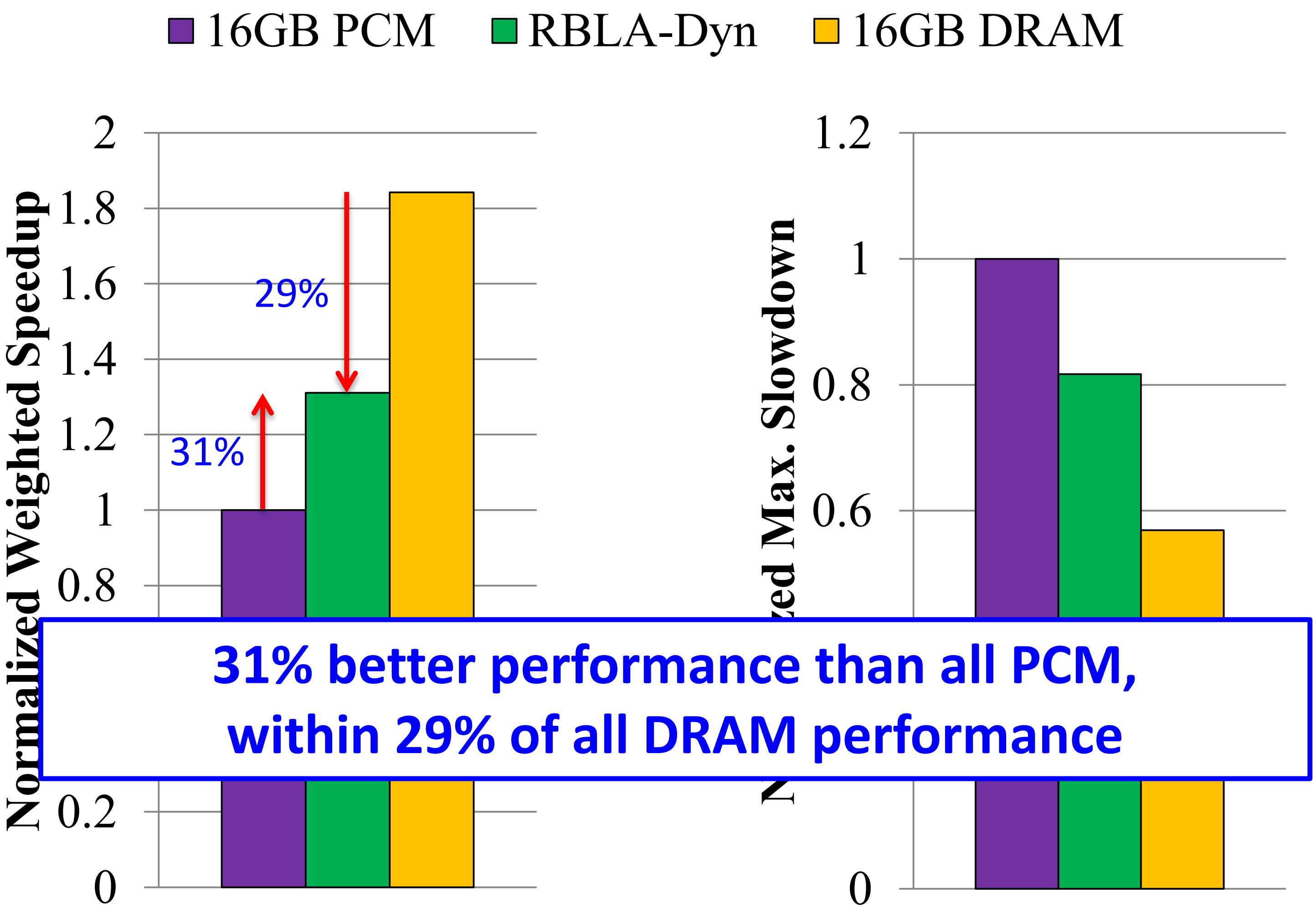
- Yoon et al., "Row Buffer Locality-Aware Data Placement in Hybrid Memories," ICCD 2012 Best Paper Award.

Row-Locality-Aware Data Placement: Results



Memory energy-efficiency and fairness also improve correspondingly

Hybrid vs. All-PCM/DRAM



31% better performance than all PCM, within 29% of all DRAM performance

Other Opportunities with Emerging Technologies

- Merging of memory and storage
  - e.g., a single interface to manage all data

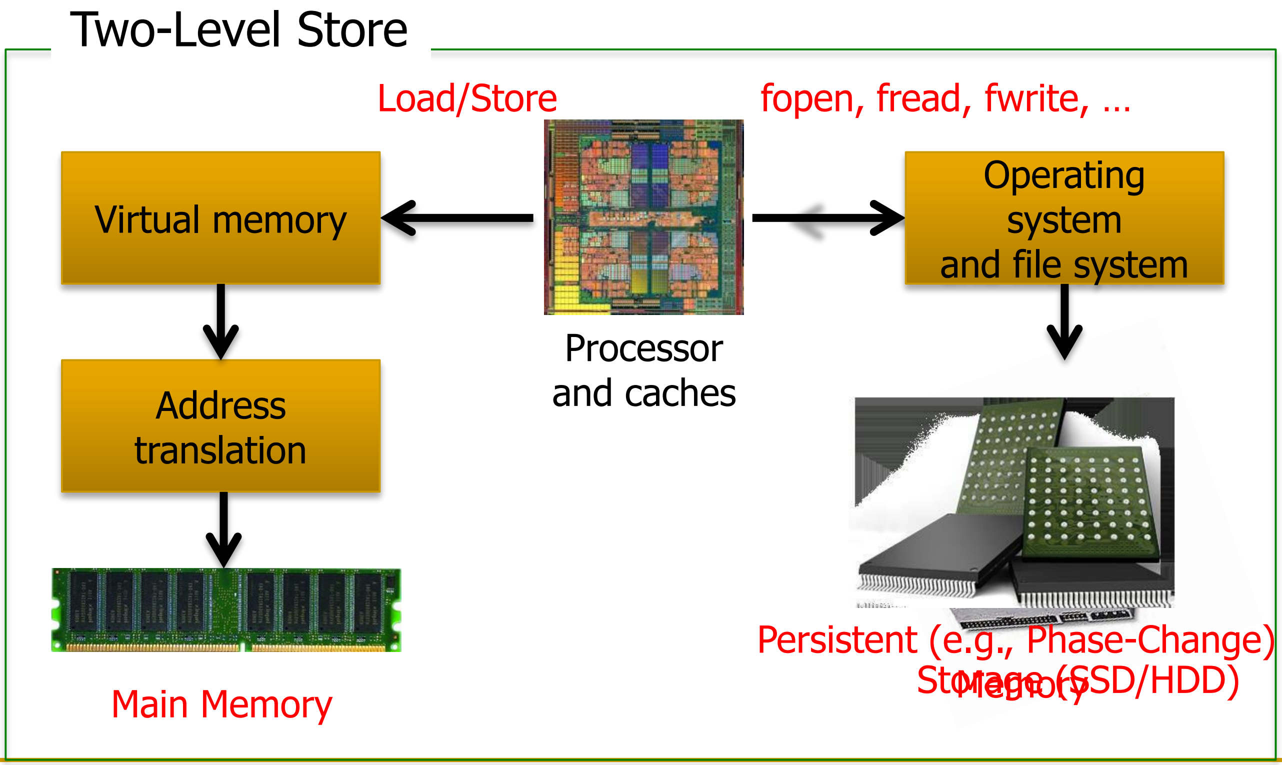
- New applications
  - e.g., ultra-fast checkpoint and restore

- More robust system design
  - e.g., reducing data loss

- Processing tightly-coupled with memory
  - e.g., enabling efficient search and filtering

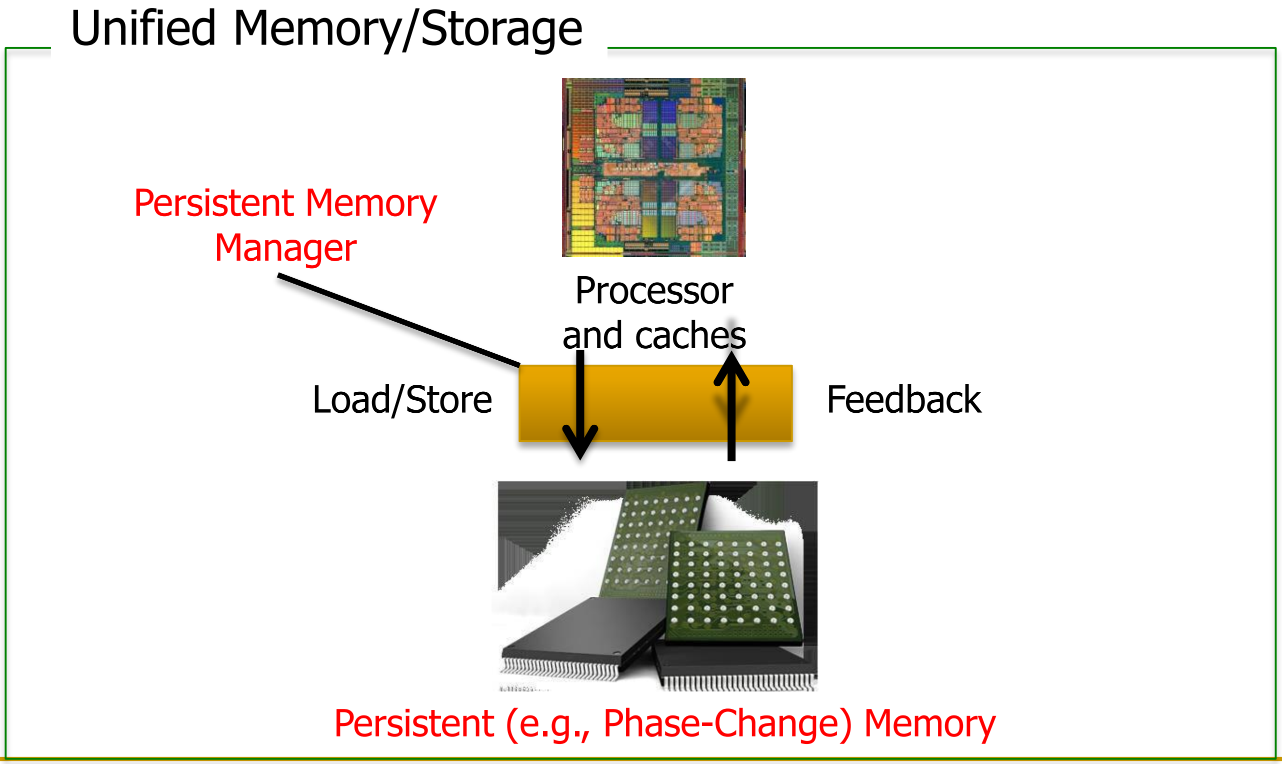
Coordinated Memory and Storage with NVM (I)

- The traditional two-level storage model is a bottleneck with NVM
  - **Volatile** data in memory → a **load/store** interface
  - **Persistent** data in storage → a **file system** interface
  - Problem: Operating system (OS) and file system (FS) code to locate, translate, buffer data become performance and energy bottlenecks with fast NVM stores



Coordinated Memory and Storage with NVM (II)

- Goal: Unify memory and storage management in a single unit to eliminate wasted work to locate, transfer, and translate data
    - Improves both energy and performance
    - Simplifies programming model as well



Unified Memory/Storage

Persistent Memory Manager

Processor and caches

Load/Store

Feedback

Persistent (e.g., Phase-Change) Memory

Meza+, "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory," WEED 2013.

The Persistent Memory Manager (PMM)

- Exposes a load/store interface to access persistent data
  - Applications can directly access persistent memory → no conversion, translation, location overhead for persistent data

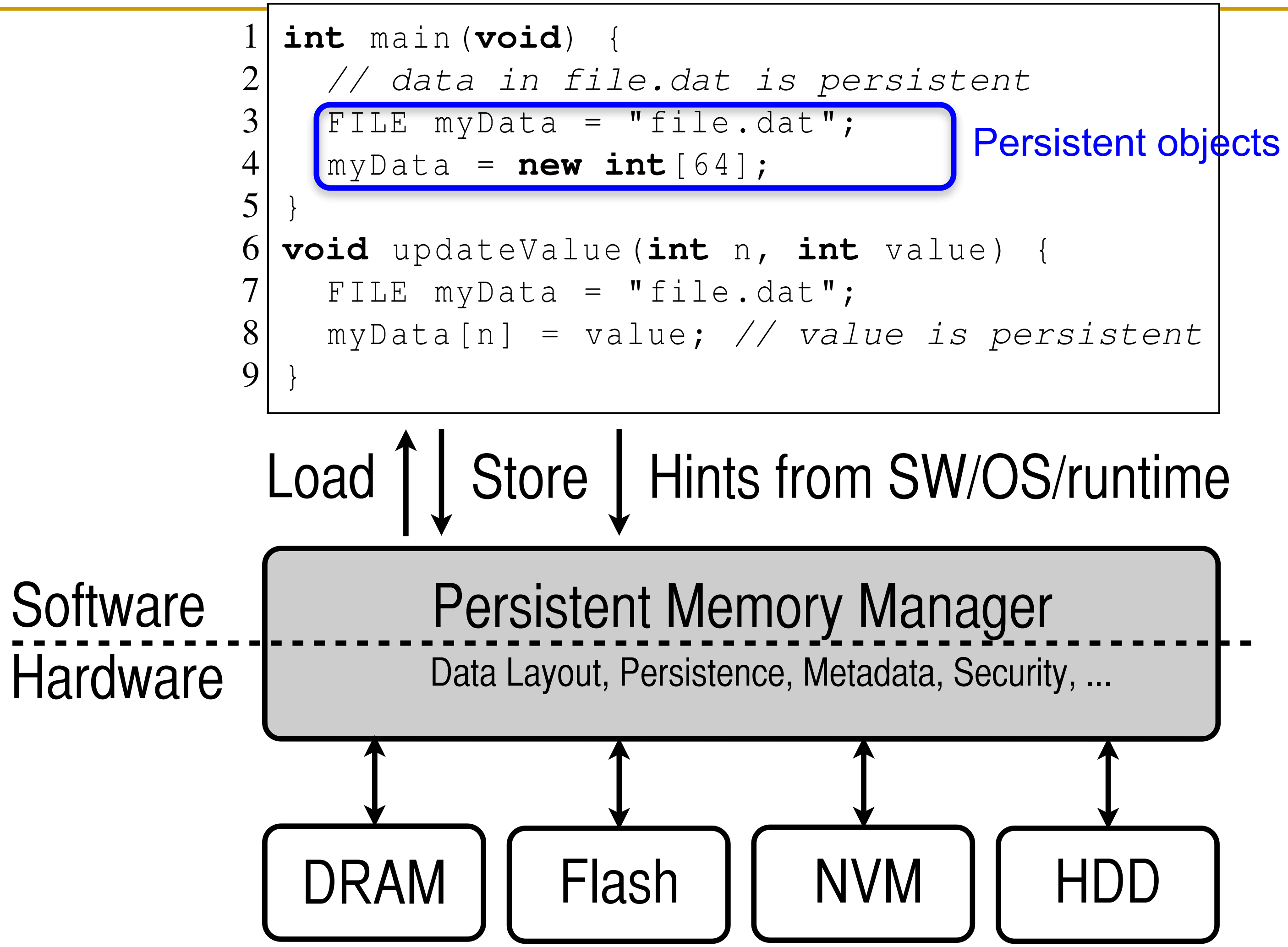
- Manages data placement, location, persistence, security
  - To get the best of multiple forms of storage

- Manages metadata storage and retrieval
  - This can lead to overheads that need to be managed

- Exposes hooks and interfaces for system software
  - To enable better data placement and management decisions

- Meza+, "A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory," WEED 2013.

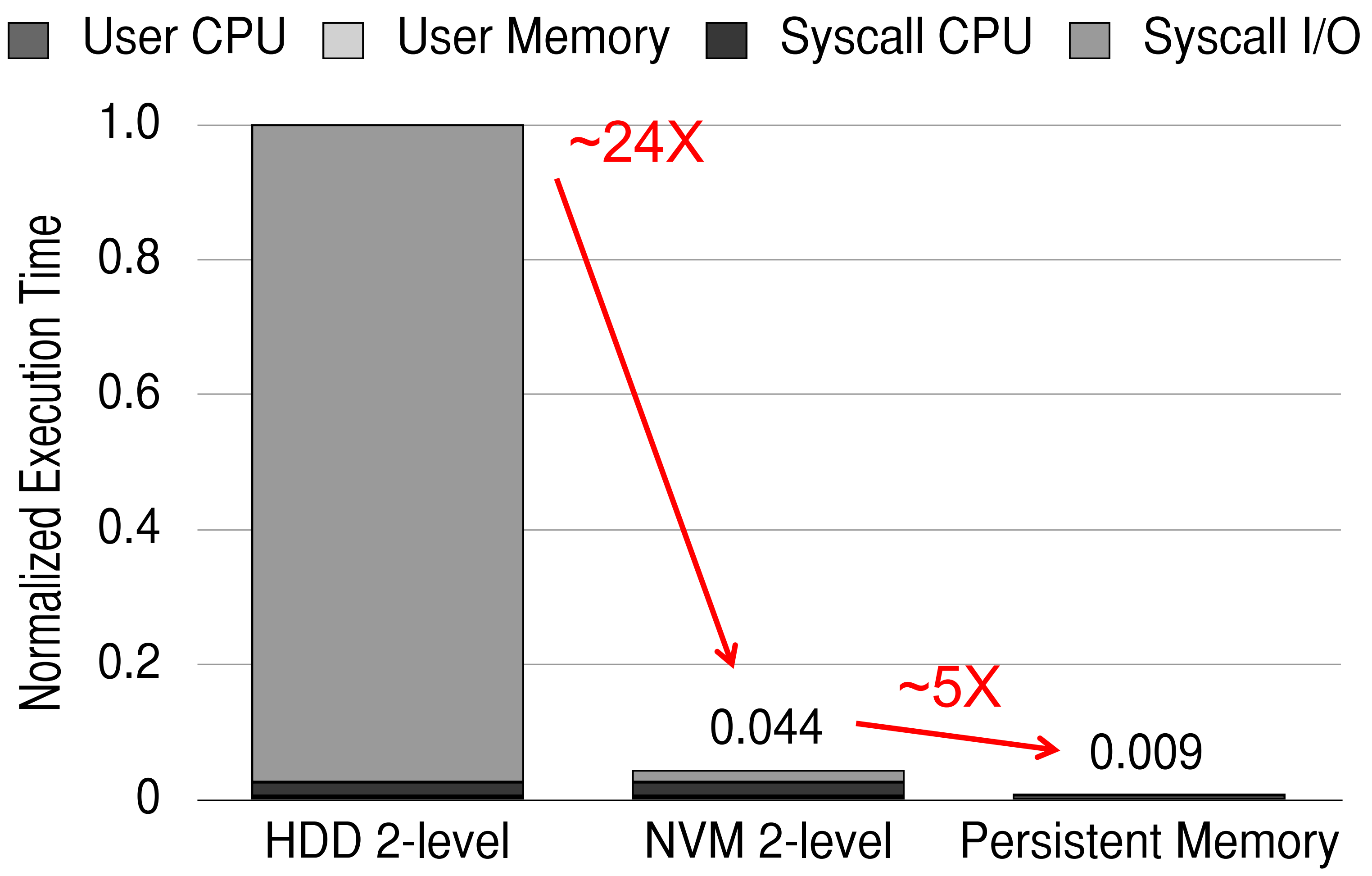
The Persistent Memory Manager (PMM)



```
int main(void) {
  // data in file.dat is persistent
  FILE myData = "file.dat";
  myData = new int[64];
}
void updateValue(int n, int value) {
  FILE myData = "file.dat";
  myData[n] = value; // value is persistent
}
```

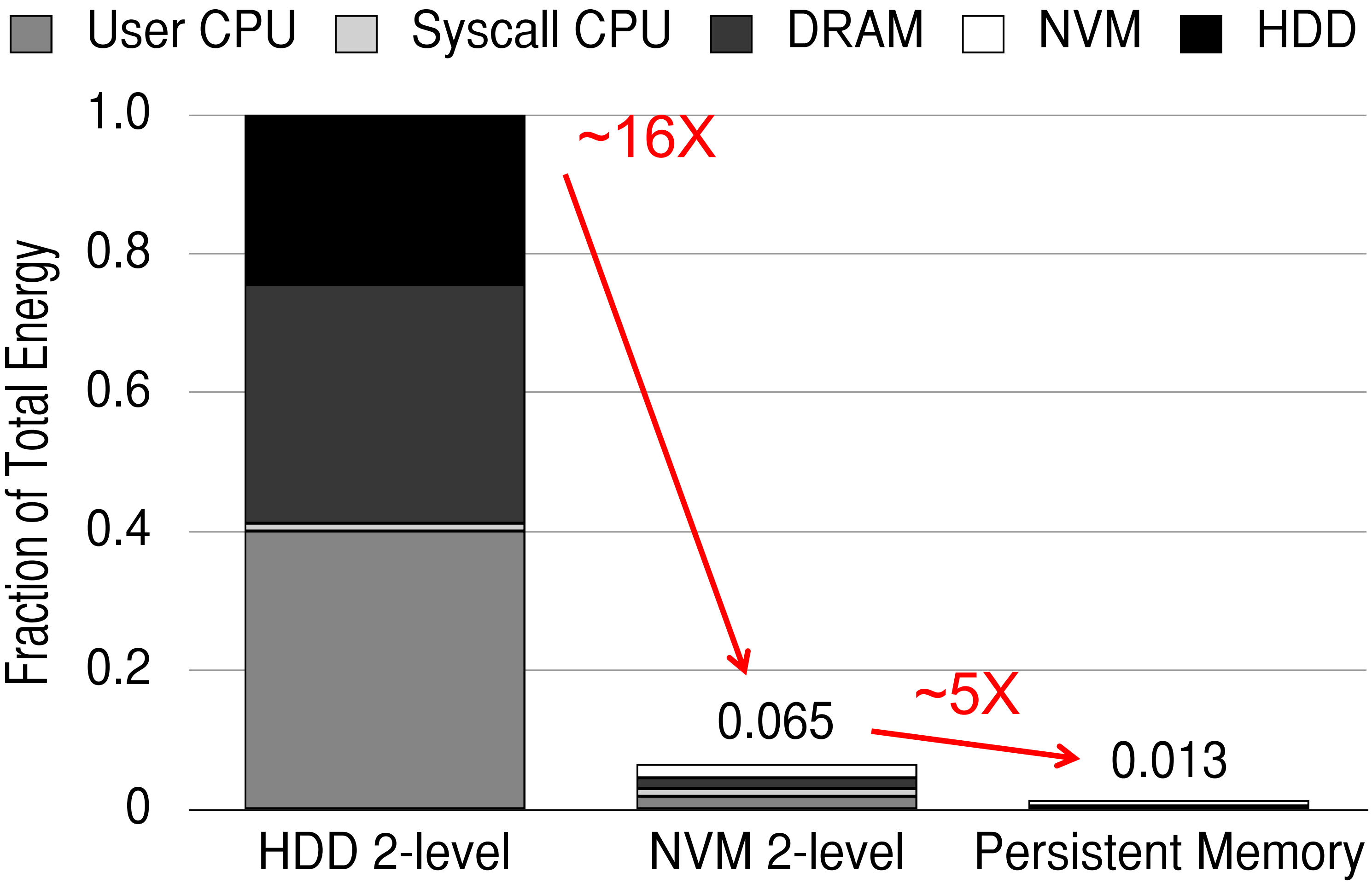
PMM uses access and hint information to allocate, locate, migrate and access data in the heterogeneous array of devices

Performance Benefits of a Single-Level Store



Results for PostMark

Energy Benefits of a Single-Level Store



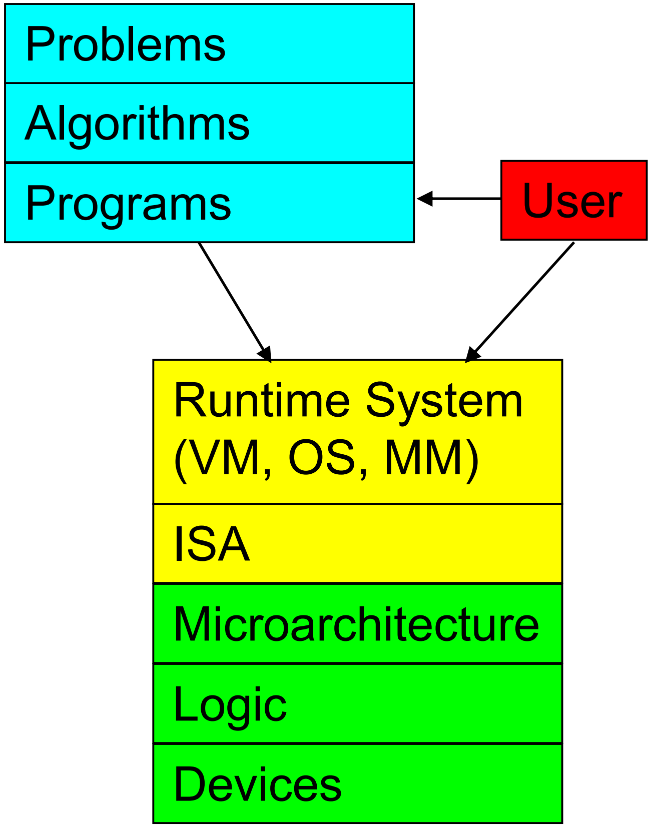
Results for PostMark

Enabling and Exploiting NVM: Issues

- Many issues and ideas from technology layer to algorithms layer

- Enabling NVM and hybrid memory
  - How to tolerate errors?
  - How to enable secure operation?
  - How to tolerate performance and power shortcomings?
  - How to minimize cost?

- Exploiting emerging tecnologies
  - How to exploit non-volatility?
  - How to minimize energy consumption?
  - How to exploit NVM on chip?



Three Principles for (Memory) Scaling

- Better cooperation between devices and the system
  - Expose more information about devices to upper layers
  - More flexible interfaces

- Better-than-worst-case design
  - Do not optimize for the worst case
  - Worst case should not determine the common case

- Heterogeneity in design (specialization, asymmetry)
  - Enables a more efficient design (No one size fits all)

- These principles are related and sometimes coupled

Security Challenges of Emerging Technologies

1. Limited endurance → Wearout attacks

2. Non-volatility → Data persists in memory after powerdown
→ Easy retrieval of privileged or private information

3. Multiple bits per cell → Information leakage (via side channel)

Securing Emerging Memory Technologies

1. Limited endurance → Wearout attacks

    Better architecting of memory chips to absorb writes

    Hybrid memory system management

    Online wearout attack detection

2. Non-volatility → Data persists in memory after powerdown

    → Easy retrieval of privileged or private information

    Efficient encryption/decryption of whole main memory

    Hybrid memory system management

3. Multiple bits per cell → Information leakage (via side channel)

    System design to hide side channel information

Summary of Emerging Memory Technologies

- Key trends affecting main memory
  - End of DRAM scaling (cost, capacity, efficiency)
  - Need for high capacity
  - Need for energy efficiency

- Emerging NVM technologies can help
  - PCM or STT-MRAM more scalable than DRAM and non-volatile
  - But, they have shortcomings: latency, active energy, endurance

- We need to enable promising NVM technologies by overcoming their shortcomings

- Many exciting opportunities to reinvent main memory at all layers of computing stack