Rethinking Memory System Design for Data-Intensive Computing

Onur Mutlu
onur@cmu.edu
October 11-16, 2013
Three Key Problems in Systems

- **The memory system**
  - Data storage and movement limit performance & efficiency

- **Efficiency (performance and energy) → scalability**
  - Efficiency limits performance & scalability

- **Predictability and robustness**
  - Predictable performance and QoS become first class constraints as systems scale in size and technology
The Main Memory/Storage System

- 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
Memory System: A *Shared Resource* View

![Diagram of a memory system with cores and caches interconnected]
State of the Main Memory System

- Recent technology, architecture, and application trends
  - lead to new requirements
  - exacerbate old requirements

- DRAM and memory controllers, as we know them today, are (will be) unlikely to satisfy all requirements

- Some emerging non-volatile memory technologies (e.g., PCM) enable new opportunities: memory+storage merging

- We need to rethink the main memory system
  - to fix DRAM issues and enable emerging technologies
  - to satisfy all requirements
Agenda

- Major Trends Affecting Main Memory
- The Memory Scaling Problem and Solution Directions
  - New Memory Architectures
  - Enabling Emerging Technologies: Hybrid Memory Systems
- How Can We Do Better?
- Summary
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
Major Trends Affecting Main Memory (II)

- Need for main memory capacity, bandwidth, QoS increasing
  - Multi-core: increasing number of cores/agents
  - Data-intensive applications: increasing demand/hunger for data
  - Consolidation: cloud computing, GPUs, mobile, heterogeneity

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

- DRAM technology scaling is ending
Example: The Memory Capacity Gap

- Memory capacity per core expected to drop by 30% every two years.
- Trends worse for memory bandwidth per core!
Major Trends Affecting Main Memory (III)

- Need for main memory capacity, bandwidth, QoS increasing

- Main memory energy/power is a key system design concern
  - ~40-50% energy spent in off-chip memory hierarchy [Lefurgy, IEEE Computer 2003]
  - DRAM consumes power even when not used (periodic refresh)

- DRAM technology scaling is ending
Major Trends Affecting Main Memory (IV)

- Need for main memory capacity, bandwidth, QoS increasing

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

- **DRAM technology scaling is ending**
  - ITRS projects DRAM will not scale easily below X nm
  - Scaling has provided many benefits:
    - higher capacity (density), lower cost, lower energy
Agenda

- Major Trends Affecting Main Memory
- The Memory Scaling Problem and Solution Directions
  - New Memory Architectures
  - Enabling Emerging Technologies: Hybrid Memory Systems
- How Can We Do Better?
- Summary
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
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

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?

Hybrid Memory Systems

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

Yoon, Meza et al., “Row Buffer Locality Aware Caching Policies for Hybrid Memories,” ICCD 2012 Best Paper Award.
An Orthogonal Issue: Memory Interference

Cores’ interfere with each other when accessing shared main memory.
An Orthogonal Issue: Memory Interference

- **Problem:** Memory interference between cores is uncontrolled
  → unfairness, starvation, low performance
  → uncontrollable, unpredictable, vulnerable system

- **Solution:** QoS-Aware Memory Systems
  - Hardware designed to provide a configurable fairness substrate
    - Application-aware memory scheduling, partitioning, throttling
  - Software designed to configure the resources to satisfy different QoS goals

- QoS-aware memory controllers and interconnects can provide predictable performance and higher efficiency
Designing QoS-Aware Memory Systems: Approaches

- **Smart resources:** Design each shared resource to have a configurable interference control/reduction mechanism
  - **QoS-aware memory controllers** [Mutlu+ MICRO’07] [Moscibroda+, Usenix Security’07] [Mutlu+ ISCA’08, Top Picks’09] [Kim+ HPCA’10] [Kim+ MICRO’10, Top Picks’11] [Ebrahimi+ ISCA’11, MICRO’11] [Ausavarungnirun+, ISCA’12] [Subramanian+, HPCA’13]
  - **QoS-aware interconnects** [Das+ MICRO’09, ISCA’10, Top Picks ’11] [Grot+ MICRO’09, ISCA’11, Top Picks ’12]
  - **QoS-aware caches**
- **Dumb resources:** Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - **Source throttling to control access to memory system** [Ebrahimi+ ASPLOS’10, ISCA’11, TOCS’12] [Ebrahimi+ MICRO’09] [Nychis+ HotNets’10] [Nychis+ SIGCOMM’12]
  - **QoS-aware data mapping to memory controllers** [Muralidhara+ MICRO’11]
  - **QoS-aware thread scheduling to cores** [Das+ HPCA’13]
Some Current Directions

- **New memory/storage + compute architectures**
  - Rethinking DRAM
  - Processing close to data; accelerating bulk operations
  - Ensuring memory/storage reliability and robustness

- **Enabling emerging NVM technologies**
  - Hybrid memory systems with automatic data management
  - Coordinated management of memory and storage with NVM

- **System-level memory/storage QoS**
  - QoS-aware controller and system design
  - Coordinated memory + storage QoS
Agenda

- Major Trends Affecting Main Memory
- The Memory Scaling Problem and Solution Directions
  - New Memory Architectures
  - Enabling Emerging Technologies: Hybrid Memory Systems
- How Can We Do Better?
- Summary
Tolerating DRAM: Example Techniques

- Retention-Aware DRAM Refresh: Reducing Refresh Impact
- Tiered-Latency DRAM: Reducing DRAM Latency
- RowClone: Accelerating Page Copy and Initialization
- Subarray-Level Parallelism: Reducing Bank Conflict Impact
- Linearly Compressed Pages: Efficient Memory Compression
Today’s Memory: Bulk Data Copy

1) High latency
2) High bandwidth utilization
3) Cache pollution
4) Unwanted data movement
Future: RowClone (In-Memory Copy)

1) Low latency
2) Low bandwidth utilization
3) No cache pollution
4) No unwanted data movement
DRAM operation (load one byte)

Step 1: Activate row

Step 2: Read Transfer byte onto bus

Row Buffer (4 Kbits)

DRAM array

Data pins (8 bits)

Memory Bus
RowClone: in-DRAM Row Copy (and Initialization)

Step 1: Activate row A
Step 2: Activate row B

DRAM array
Row Buffer (4 Kbits)
Data pins (8 bits)
Memory Bus
RowClone: Latency and Energy Savings

- Baseline
- Intra-Subarray
- Inter-Bank
- Inter-Subarray

RowClone: Overall Performance

50 Workloads

<table>
<thead>
<tr>
<th>Application</th>
<th>bootup</th>
<th>compile</th>
<th>mcached</th>
<th>mysql</th>
<th>shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy Reduction</td>
<td>40%</td>
<td>32%</td>
<td>15%</td>
<td>17%</td>
<td>67%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Workloads</td>
<td>138</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>Weighted Speedup Improvement</td>
<td>15%</td>
<td>20%</td>
<td>27%</td>
</tr>
<tr>
<td>Energy per Instruction Reduction Reduction</td>
<td>19%</td>
<td>17%</td>
<td>17%</td>
</tr>
</tbody>
</table>
Goal: Ultra-Efficient Processing Close to Data

- CPU core
- CPU core
- CPU core
- CPU core
- mini-CPU core
- video core
- imaging core
- GPU core
- GPU core
- GPU core
- GPU core
- LLC
- Memory Controller
- Memory Bus

Slide credit: Prof. Kayvon Fatahalian, CMU
Enabling Ultra-Efficient (Visual) Search

- What is the right partitioning of computation capability?
- What is the right low-cost memory substrate?
- What memory technologies are the best enablers?
- How do we rethink/ease (visual) search?
Tolerating DRAM: Example Techniques

- Retention-Aware DRAM Refresh: Reducing Refresh Impact
- Tiered-Latency DRAM: Reducing DRAM Latency
- RowClone: Accelerating Page Copy and Initialization
- Subarray-Level Parallelism: Reducing Bank Conflict Impact
- Linearly Compressed Pages: Efficient Memory Compression
DRAM latency continues to be a critical bottleneck
What Causes the Long Latency?

DRAM Chip

\[ \text{DRAM Latency} = \text{Subarray Latency} + \text{I/O Latency} \]

Subarray

channel

I/O

Dominant
Why is the Subarray So Slow?

- Long bitline
  - Amortizes sense amplifier cost $\rightarrow$ Small area
  - Large bitline capacitance $\rightarrow$ High latency & power
Trade-Off: Area (Die Size) vs. Latency

Long Bitline

Short Bitline

Faster

Smaller

Trade-Off: Area vs. Latency
Trade-Off: Area (Die Size) vs. Latency

- **Fancy DRAM**
  - Short Bitline
  - Normalized DRAM Area: 32
  - Latency (ns): 64
- **Commodity DRAM**
  - Long Bitline
  - Normalized DRAM Area: 128
  - Latency (ns): 256
- **512 cells/bitline**

**GOAL**: Cheaper and Faster
Approximating the Best of Both Worlds

Long Bitline

Small Area

High Latency

Our Proposal

Add Isolation Transistors

Short Bitline

Large Area

Low Latency

Need Isolation

Fast
Approximating the Best of Both Worlds

Long Bitline Tiered-Latency DRAM Short Bitline

Small Area Small Area Low Latency Low Latency

High Latency

Large Area

Small area using long bitline

Low Latency
Tiered-Latency DRAM

• Divide a bitline into two segments with an isolation transistor

Commodity DRAM vs. TL-DRAM

- DRAM Latency ($t_{RC}$)
- DRAM Power

- DRAM Area Overhead
  ~3%: mainly due to the isolation transistors
Trade-Off: Area (Die-Area) vs. Latency

- Cheaper vs. Normalized DRAM Area
- Faster vs. Latency (ns)

Key Points:
- Near Segment
- Far Segment
- GOAL
- Latency values: 32, 64, 128, 256, 512 cells/bitline

Graph showing the trade-off between area and latency for different cell counts.
Leveraging Tiered-Latency DRAM

• TL-DRAM is a substrate that can be leveraged by the hardware and/or software

• Many potential uses
  1. Use near segment as hardware-managed inclusive cache to far segment
  2. Use near segment as hardware-managed exclusive cache to far segment
  3. Profile-based page mapping by operating system
  4. Simply replace DRAM with TL-DRAM
Using near segment as a cache improves performance and reduces power consumption.
Agenda

- Major Trends Affecting Main Memory
- The Memory Scaling Problem and Solution Directions
  - New Memory Architectures
  - Enabling Emerging Technologies: Hybrid Memory Systems
- How Can We Do Better?
- Summary
Solution 2: Emerging Memory Technologies

- Some emerging resistive memory technologies seem more scalable than DRAM (and they are non-volatile)

- Example: Phase Change Memory
  - Data stored by changing phase of material
  - Data read by detecting material’s resistance
  - Expected to scale to 9nm (2022 [ITRS])
  - Prototyped at 20nm (Raoux+, IBM JRD 2008)
  - Expected to be denser than DRAM: can store multiple bits/cell

- But, emerging technologies have (many) shortcomings
  - Can they be enabled to replace/augment/surpass DRAM?
Phase Change Memory: Pros and Cons

- Pros over DRAM
  - Better technology scaling (capacity and cost)
  - 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)

- Challenges in enabling PCM as DRAM replacement/helper:
  - Mitigate PCM shortcomings
  - Find the right way to place PCM in the system
PCM-based Main Memory (I)

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

- Hybrid PCM+DRAM [Qureshi+ ISCA’09, Dhiman+ DAC’09]:
  - How to partition/migrate data between PCM and DRAM
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
An Initial Study: Replace DRAM with PCM

  - Surveyed prototypes from 2003-2008 (e.g. IEDM, VLSI, ISSCC)
  - Derived “average” PCM parameters for F=90nm

### Density
- $9 - 12F^2$ using BJT
- $1.5 \times$ DRAM

### Latency
- 50ns Rd, 150ns Wr
- $4 \times, 12 \times$ DRAM

### Endurance
- $1E+08$ writes
- $1E-08 \times$ DRAM

### Energy
- $40\mu$A Rd, $150\mu$A Wr
- $2 \times, 43 \times$ 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

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

[Diagram: CPU, 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

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

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
Aside: STT-RAM as Main Memory

- 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.

Aside: STT-RAM: 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)
Architected STT-RAM as Main Memory

- 4-core, 4GB main memory, multiprogrammed workloads
- ~6% performance loss, ~60% energy savings vs. DRAM

Agenda

- Major Trends Affecting Main Memory
- The Memory Scaling Problem and Solution Directions
  - New Memory Architectures
  - Enabling Emerging Technologies: Hybrid Memory Systems
- How Can We Do Better?
- Summary
Principles (So Far)

- 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
  - Enables a more efficient design (No one size fits all)
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

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

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

~24X

~5X

HDD 2-level  NVM 2-level  Persistent Memory

SAFARI

Results for PostMark
Energy Benefits of a Single-Level Store

Results for PostMark

- HDD 2-level
- NVM 2-level
- Persistent Memory

~16X
~5X
Agenda

- Major Trends Affecting Main Memory
- The Memory Scaling Problem and Solution Directions
  - New Memory Architectures
  - Enabling Emerging Technologies: Hybrid Memory Systems
- How Can We Do Better?
- Summary
Summary: Memory/Storage Scaling

- Memory/storage scaling problems are a critical bottleneck for system performance, efficiency, and usability

- New memory/storage + compute architectures
  - Rethinking DRAM; processing close to data; accelerating bulk operations
  - Ensuring memory/storage reliability and robustness

- Enabling emerging NVM technologies
  - Hybrid memory systems with automatic data management
  - Coordinated management of memory and storage with NVM

- System-level memory/storage QoS
  - QoS-aware controller and system design
  - Coordinated memory + storage QoS

- Software/hardware/device cooperation essential
These slides are a very short version of the Scalable Memory Systems course at ACACES 2013

Website for Course Slides, Papers, and Videos

- http://users.ece.cmu.edu/~omutlu/acaces2013-memory.html
- http://users.ece.cmu.edu/~omutlu/projects.htm
- Includes extended lecture notes and readings

Overview Reading

- Onur Mutlu,
  "Memory Scaling: A Systems Architecture Perspective"
  Technical talk at MemCon 2013 (MEMCON), Santa Clara, CA, August 2013. Slides (pptx) (pdf)
Thank you.

Feel free to email me with any questions & feedback

onur@cmu.edu
Rethinking Memory System Design for Data-Intensive Computing

Onur Mutlu
onur@cmu.edu
October 11-16, 2013
Backup Slides
Backup Slides Agenda

- DRAM Retention Time Characterization: Summary of Findings
- Building Large DRAM Caches for Hybrid Memories
- Memory QoS and Predictable Performance
- Subarray-Level Parallelism (SALP) in DRAM
- Coordinated Memory and Storage with NVM
- New DRAM Architectures
RetentionPolicy Characterization of Modern DRAM Devices

Summary (I)

- DRAM requires periodic refresh to avoid data loss
  - Refresh wastes energy, reduces performance, limits DRAM density scaling
- Many past works observed that different DRAM cells can retain data for different times without being refreshed; proposed reducing refresh rate for strong DRAM cells
  - Problem: These techniques require an accurate profile of the retention time of all DRAM cells
- Our goal: To analyze the retention time behavior of DRAM cells in modern DRAM devices to aid the collection of accurate profile information
- Our experiments: We characterize 248 modern commodity DDR3 DRAM chips from 5 manufacturers using an FPGA based testing platform
- Two Key Issues:
  1. Data Pattern Dependence: A cell’s retention time is heavily dependent on data values stored in itself and nearby cells, which cannot easily be controlled.
  2. Variable Retention Time: Retention time of some cells change unpredictably from high to low at large timescales.
Summary (II)

- **Key findings on Data Pattern Dependence**
  - There is no observed single data pattern that elicits the lowest retention times for a DRAM device → very hard to find this pattern
  - DPD varies between devices due to variation in DRAM array circuit design between manufacturers
  - DPD of retention time gets worse as DRAM scales to smaller feature sizes

- **Key findings on Variable Retention Time**
  - VRT is common in modern DRAM cells that are weak
  - The timescale at which VRT occurs is very large (e.g., a cell can stay in high retention time state for a day or longer) → finding minimum retention time can take very long

- Future work on retention time profiling must address these issues
Building Large Caches for Hybrid Memories

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 low-cost hardware-managed DRAM cache?

Two ideas:
- Locality-aware data placement [Yoon+ , ICCD 2012]
- Cheap tag stores and dynamic granularity [Meza+, IEEE CAL 2012]
The Problem with Large DRAM Caches

- A large DRAM cache requires a large metadata (tag + block-based information) store
- How do we design an efficient DRAM cache?

![Diagram showing DRAM cache design](image)

Metadata: X \(\rightarrow\) DRAM

DRAM (small, fast cache) - CPU

Access X

PCM (high capacity)
Idea 1: Store Tags in Main Memory

- Store tags in the same row as data in DRAM
  - Data and metadata can be accessed together

- Benefit: No on-chip tag storage overhead
- Downsides:
  - Cache hit determined only after a DRAM access
  - Cache hit requires two DRAM accesses
Idea 2: Cache Tags in On-Chip SRAM

- Recall Idea 1: Store all metadata in DRAM
  - To reduce metadata storage overhead

- Idea 2: Cache in on-chip SRAM frequently-accessed metadata
  - Cache only a small amount to keep SRAM size small
Idea 3: Dynamic Data Transfer Granularity

- Some applications benefit from caching more data
  - They have good spatial locality
- Others do not
  - Large granularity wastes bandwidth and reduces cache utilization

- Idea 3: Simple dynamic caching granularity policy
  - Cost-benefit analysis to determine best DRAM cache block size

TIMBER Performance

TIMBER Energy Efficiency

Normalized Performance per Watt
(for Memory System)

18%

SRAM  |  Region  |  TIM  |  TIMBER  |  TIMBER-Dyn

Hybrid Main Memory: Research Topics

- Many research topics from technology layer to algorithms layer

- Enabling NVM and hybrid memory
  - How to maximize performance?
  - How to maximize lifetime?
  - How to prevent denial of service?

- Exploiting emerging technologies
  - How to exploit non-volatility?
  - How to minimize energy consumption?
  - How to minimize cost?
  - How to exploit NVM on chip?
Security Challenges of Emerging Technologies

1. Limited endurance $\rightarrow$ Wearout attacks

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

3. Multiple bits per cell $\rightarrow$ Information leakage (via side channel)
Memory QoS
Trend: Many Cores on Chip

- Simpler and lower power than a single large core
- Large scale parallelism on chip

- AMD Barcelona
  - 4 cores
  - 8 cores

- Intel Core i7
  - 8+1 cores

- IBM Cell BE
  - 8 cores

- IBM POWER7
  - 8 cores

- Nvidia Fermi
  - 448 “cores”

- Intel SCC
  - 48 cores, networked

- Tilera TILE Gx
  - 100 cores, networked
Many Cores on Chip

- What we want:
  - $N$ times the system performance with $N$ times the cores

- What do we get today?
Unfair Slowdowns due to Interference

Uncontrolled Interference: An Example

Multi-Core Chip

Shared DRAM Memory System

unfairness
A Memory Performance Hog

// initialize large arrays A, B
for (j=0; j<N; j++) {
    index = j*linesize;
    A[index] = B[index];
    …
}

STREAM
- Sequential memory access
- Very high row buffer locality (96% hit rate)
- Memory intensive

RANDOM
- Random memory access
- Very low row buffer locality (3% hit rate)
- Similarly memory intensive

What Does the Memory Hog Do?

Row Buffer
Row decoder
Column mux
Data
Row 0

T0: Row 0
T0: Row 6
T0: Row 101
T0: Row 16

Memory Request Buffer

Row size: 8KB, cache block size: 64B
128 (8KB/64B) requests of T0 serviced before T1

Effect of the Memory Performance Hog

Results on Intel Pentium D running Windows XP
(Similar results for Intel Core Duo and AMD Turion, and on Fedora Linux)

Greater Problem with More Cores

- Vulnerable to denial of service (DoS)
- Unable to enforce priorities or SLAs
- Low system performance

Uncontrollable, unpredictable system
Distributed DoS in Networked Multi-Core Systems

Cores connected via packet-switched routers on chip

\(~5000\times\) slowdown

How Do We Solve The Problem?

- Inter-thread interference is uncontrolled in all memory resources
  - Memory controller
  - Interconnect
  - Caches

- We need to control it
  - i.e., design an interference-aware (QoS-aware) memory system
QoS-Aware Memory Systems: Challenges

- How do we reduce inter-thread interference?
  - Improve system performance and core utilization
  - Reduce request serialization and core starvation

- How do we control inter-thread interference?
  - Provide mechanisms to enable system software to enforce QoS policies
  - While providing high system performance

- How do we make the memory system configurable/flexible?
  - Enable flexible mechanisms that can achieve many goals
    - Provide fairness or throughput when needed
    - Satisfy performance guarantees when needed
Designing QoS-Aware Memory Systems: Approaches

- **Smart resources**: Design each shared resource to have a configurable interference control/reduction mechanism
  - QoS-aware memory controllers [Mutlu+ MICRO’07] [Moscibroda+, Usenix Security’07] [Mutlu+ ISCA’08, Top Picks’09] [Kim+ HPCA’10] [Kim+ MICRO’10, Top Picks’11] [Ebrahimi+ ISCA’11, MICRO’11] [Ausavarungnirun+, ISCA’12]
  - QoS-aware interconnects [Das+ MICRO’09, ISCA’10, Top Picks’11] [Grot+ MICRO’09, ISCA’11, Top Picks’12]
  - QoS-aware caches

- **Dumb resources**: Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS’10, ISCA’11, TOCS’12] [Ebrahimi+ MICRO’09] [Nychis+ HotNets’10]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO’11]
  - QoS-aware thread scheduling to cores
A Mechanism to Reduce Memory Interference

Memory Channel Partitioning

- Idea: System software maps badly-interfering applications’ pages to different channels [Muralidhara+, MICRO’11]

- Separate data of low/high intensity and low/high row-locality applications
- Especially effective in reducing interference of threads with “medium” and “heavy” memory intensity
- 11% higher performance over existing systems (200 workloads)
Designing QoS-Aware Memory Systems: Approaches

- **Smart resources:** Design each shared resource to have a configurable interference control/reduction mechanism
  
  - **QoS-aware memory controllers**
    - [Mutlu+ MICRO'07] [Moscibroda+, Usenix Security'07] [Mutlu+ ISCA'08, Top Picks'09] [Kim+ HPCA'10] [Kim+ MICRO’10, Top Picks’11] [Ebrahimi+ ISCA’11, MICRO’11] [Ausavarungnirun+, ISCA’12] [Subramanian+, HPCA’13]
    - QoS-aware interconnects [Das+ MICRO’09, ISCA’10, Top Picks’11] [Grot+ MICRO’09, ISCA’11, Top Picks’12]
    - QoS-aware caches

- **Dumb resources:** Keep each resource free-for-all, but reduce/control interference by injection control or data mapping
  
  - Source throttling to control access to memory system [Ebrahimi+ ASPLOS’10, ISCA’11, TOCS’12] [Ebrahimi+ MICRO’09] [Nychis+ HotNets’10] [Nychis+ SIGCOMM’12]
  - QoS-aware data mapping to memory controllers [Muralidhara+ MICRO’11]
  - QoS-aware thread scheduling to cores [Das+ HPCA’13]
QoS-Aware Memory Scheduling

- How to schedule requests to provide
  - High system performance
  - High fairness to applications
  - Configurability to system software

- Memory controller needs to be aware of threads

Resolves memory contention by scheduling requests
QoS-Aware Memory Scheduling: Evolution

- **Stall-time fair memory scheduling** [Mutlu+ MICRO’07]
  - Idea: Estimate and balance thread slowdowns
  - Takeaway: Proportional thread progress improves performance, especially when threads are “heavy” (memory intensive)

- **Parallelism-aware batch scheduling** [Mutlu+ ISCA’08, Top Picks’09]
  - Idea: Rank threads and service in rank order (to preserve bank parallelism); batch requests to prevent starvation
  - Takeaway: Preserving within-thread bank-parallelism improves performance; request batching improves fairness

- **ATLAS memory scheduler** [Kim+ HPCA’10]
  - Idea: Prioritize threads that have attained the least service from the memory scheduler
  - Takeaway: Prioritizing “light” threads improves performance
Throughput vs. Fairness

**Throughput biased approach**
Prioritize less memory-intensive threads

**Fairness biased approach**
Take turns accessing memory

- Good for throughput
  - Less memory intensive
  - Higher priority

- Does not starve
  - Starvation ➔ Unfairness
  - Not prioritized ➔ Reduced throughput

Single policy for all threads is insufficient
Achieving the Best of Both Worlds

For Throughput
- Prioritize memory-non-intensive threads

For Fairness
- Unfairness caused by memory-intensive being prioritized over each other
  - Shuffle thread ranking
- Memory-intensive threads have different vulnerability to interference
  - Shuffle asymmetrically
Thread Cluster Memory Scheduling [Kim+ MICRO’10]

1. Group threads into two **clusters**
2. Prioritize **non-intensive cluster**
3. Different policies for each cluster

**Memory-non-intensive**

**Memory-intensive**

**Non-intensive cluster**

**Intensive cluster**

**Throughput**

**Fairness**

TCM: Quantum-Based Operation

**Previous quantum** (~1M cycles)

- Monitor thread behavior
  1. Memory intensity
  2. Bank-level parallelism
  3. Row-buffer locality

**Current quantum** (~1M cycles)

- Shuffle interval (~1K cycles)

**Beginning of quantum**:
- Perform clustering
- Compute niceness of intensive threads

---

TCM: Throughput and Fairness

24 cores, 4 memory controllers, 96 workloads

Better fairness

Better system throughput

TCM, a heterogeneous scheduling policy, provides best fairness and system throughput
TCM: Fairness-Throughput Tradeoff

When configuration parameter is varied...

Adjusted ClusterThreshold

Better system throughput

TCM allows robust fairness-throughput tradeoff
More on TCM

Memory Control in CPU-GPU Systems

- **Observation:** Heterogeneous CPU-GPU systems require memory schedulers with large request buffers

- **Problem:** Existing monolithic application-aware memory scheduler designs are hard to scale to large request buffer sizes

- **Solution:** Staged Memory Scheduling (SMS) decomposes the memory controller into three simple stages:
  1) Batch formation: maintains row buffer locality
  2) Batch scheduler: reduces interference between applications
  3) DRAM command scheduler: issues requests to DRAM

- Compared to state-of-the-art memory schedulers:
  - SMS is significantly simpler and more scalable
  - SMS provides higher performance and fairness

Key Idea: Decouple Tasks into Stages

- **Idea:** Decouple the functional tasks of the memory controller
  - Partition tasks across several simpler HW structures (stages)

1) Maximize row buffer hits
   - **Stage 1:** Batch formation
   - Within each application, groups requests to the same row into batches

2) Manage contention between applications
   - **Stage 2:** Batch scheduler
   - Schedules batches from different applications

3) Satisfy DRAM timing constraints
   - **Stage 3:** DRAM command scheduler
   - Issues requests from the already-scheduled order to each bank
SMS: Staged Memory Scheduling

Core 1  Core 2  Core 3  Core 4  GPU

Stage 1
Batch Formation

Stage 2
Monolithic Scheduler

Stage 3
DRAM Command Scheduler

Batch Scheduler

To DRAM
SMS: Staged Memory Scheduling

Stage 1

Batch Formation

Core 1 -> Core 2 -> Core 3 -> Core 4 -> GPU

Stage 2

Batch Scheduler

Stage 3

DRAM Command Scheduler

Bank 1 -> Bank 2 -> Bank 3 -> Bank 4

To DRAM
SMS: Staged Memory Scheduling

Stage 1: Batch Formation

Stage 2: Batch Scheduler

Stage 3: DRAM Command Scheduler

Current Batch Scheduling Policy: RR

SMS Complexity

- Compared to a row hit first scheduler, SMS consumes*
  - 66% less area
  - 46% less static power

- Reduction comes from:
  - Monolithic scheduler → stages of simpler schedulers
  - Each stage has a simpler scheduler (considers fewer properties at a time to make the scheduling decision)
  - Each stage has simpler buffers (FIFO instead of out-of-order)
  - Each stage has a portion of the total buffer size (buffering is distributed across stages)

* Based on a Verilog model using 180nm library
SMS Performance

![Graph showing system performance vs. GPU weight with best previous scheduler highlighted. ATLAS, TCM, and FR-FCFS are indicated on the x-axis.]}
At every GPU weight, SMS outperforms the best previous scheduling algorithm for that weight.
CPU-GPU Performance Tradeoff

**CPU Performance**

- Weighted Speedup vs. SJF Probability
  - SJF Probability: 1, 0.5, 0.1, 0.05, 0
  - Weighted Speedup: 5, 5, 5, 4, 3

**GPU Frame Rate**

- Frame Rate vs. SJF Probability
  - SJF Probability: 1, 0.5, 0.1, 0.05, 0
  - Frame Rate: 50, 50, 50, 60, 90
More on SMS

- Rachata Ausavarungnirun, Kevin Chang, Lavanya Subramanian, Gabriel Loh, and Onur Mutlu,

"Staged Memory Scheduling: Achieving High Performance and Scalability in Heterogeneous Systems"

Proceedings of the 39th International Symposium on Computer Architecture (ISCA), Portland, OR, June 2012. Slides (pptx)
Stronger Memory Service Guarantees

Uncontrolled memory interference slows down applications unpredictably.

Goal: Estimate and control slowdowns.

MISE: An accurate slowdown estimation model

- Request Service Rate is a good proxy for performance
  - Slowdown = \frac{\text{Request Service Rate}_{\text{Alone}}}{\text{Request Service Rate}_{\text{Shared}}}
- Request Service Rate_{\text{Alone}} estimated by giving an application highest priority in accessing memory
- Average slowdown estimation error of MISE: 8.2% (3000 data pts)

Memory controller leverages MISE to control slowdowns

- To provide soft slowdown guarantees
- To minimize maximum slowdown

Lavanya Subramanian, Vivek Seshadri, Yoongu Kim, Ben Jaiyen, and Onur Mutlu,

"MISE: Providing Performance Predictability and Improving Fairness in Shared Main Memory Systems"

Proceedings of the 19th International Symposium on High-Performance Computer Architecture (HPCA), Shenzhen, China, February 2013. Slides (pptx)
Memory QoS in a Parallel Application

- Threads in a multithreaded application are inter-dependent
- Some threads can be on the critical path of execution due to synchronization; some threads are not
- How do we schedule requests of inter-dependent threads to maximize multithreaded application performance?

- Idea: Estimate limiter threads likely to be on the critical path and prioritize their requests; shuffle priorities of non-limiter threads to reduce memory interference among them [Ebrahimi+, MICRO’11]

- Hardware/software cooperative limiter thread estimation:
  - Thread executing the most contended critical section
  - Thread that is falling behind the most in a parallel for loop

Ebrahimi+, “Parallel Application Memory Scheduling,” MICRO 2011.
More on PAMS

- Eiman Ebrahimi, Rustam Miftakhutdinov, Chris Fallin, Chang Joo Lee, Onur Mutlu, and Yale N. Patt,

"Parallel Application Memory Scheduling"
Proceedings of the 44th International Symposium on Microarchitecture (MICRO), Porto Alegre, Brazil, December 2011. Slides (pptx)
Summary: Memory QoS Approaches and Techniques

- Approaches: **Smart** vs. **dumb** resources
  - Smart resources: QoS-aware memory scheduling
  - Dumb resources: Source throttling; channel partitioning
  - Both approaches are effective in reducing interference
  - No single best approach for all workloads

- Techniques: Request **scheduling**, source **throttling**, memory **partitioning**
  - All approaches are effective in reducing interference
  - Can be applied at different levels: hardware vs. software
  - No single best technique for all workloads

- Combined approaches and techniques are the most powerful
  - Integrated Memory Channel Partitioning and Scheduling [MICRO’11]
SALP: Reducing DRAM Bank Conflict Impact

Kim, Seshadri, Lee, Liu, Mutlu
A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM
ISCA 2012.
SALP: Reducing DRAM Bank Conflicts

- **Problem:** Bank conflicts are costly for performance and energy
  - serialized requests, wasted energy (thrashing of row buffer, busy wait)
- **Goal:** Reduce bank conflicts without adding more banks (low cost)
- **Key idea:** Exploit the internal subarray structure of a DRAM bank to parallelize bank conflicts to different subarrays
  - Slightly modify DRAM bank to reduce subarray-level hardware sharing

![Figure 1. DRAM bank organization](chart.png)
SALP: Key Ideas

- A DRAM bank consists of mostly-independent subarrays
  - Subarrays share some **global structures** to reduce cost

Key Idea of SALP: Minimally reduce sharing of global structures

Reduce the sharing of …
- Global decoder → Enables pipelined access to subarrays
- Global row buffer → Utilizes multiple local row buffers
SALP: Reduce Sharing of Global Decoder

Instead of a global latch, have \textit{per-subarray latches}.
SALP: Reduce Sharing of Global Row-Buffer

Selectively connect local row-buffers to global row-buffer using a *Designated* single-bit latch.
SALP: Baseline Bank Organization

Global Decoder

Latch

Global row-buffer

Local row-buffer

Global bitlines

Local row-buffer

Global row-buffer
SALP: Proposed Bank Organization

- **Global Decoder**
  - **Latch**
  - **D**

- **Local row-buffer**
  - **Global bitlines**

**Overhead of SALP in DRAM chip**: 0.15%

1. Global latch → per-subarray local latches
2. Designated bit latches and wire to selectively enable a subarray
SALP: Results

- Wide variety of systems with different #channels, banks, ranks, subarrays
- Server, streaming, random-access, SPEC workloads
- **Dynamic DRAM energy reduction: 19%**
  - DRAM row hit rate improvement: 13%
- **System performance improvement: 17%**
  - Within 3% of ideal (all independent banks)
- **DRAM die area overhead: 0.15%**
  - vs. 36% overhead of independent banks
More on SALP

- Yoongu Kim, Vivek Seshadri, Donghyuk Lee, Jamie Liu, and Onur Mutlu, "A Case for Exploiting Subarray-Level Parallelism (SALP) in DRAM"
  Proceedings of the 39th International Symposium on Computer Architecture (ISCA), Portland, OR, June 2012. Slides (pptx)
Coordinated Memory and Storage with NVM

Overview

- Traditional systems have a two-level storage model
  - Access *volatile* data in memory with a *load/store* interface
  - Access *persistent* data in storage with a *file system* interface
  - Problem: Operating system (OS) and file system (FS) code and buffering for storage lead to energy and performance inefficiencies

- Opportunity: New non-volatile memory (NVM) technologies can help provide fast (similar to DRAM), persistent storage (similar to Flash)
  - Unfortunately, OS and FS code can easily become energy efficiency and performance bottlenecks if we keep the traditional storage model

- This work: makes a case for hardware/software cooperative management of storage and memory within a single-level
  - We describe the idea of a Persistent Memory Manager (PMM) for efficiently coordinating storage and memory, and quantify its benefit
  - And, examine questions and challenges to address to realize PMM
Two-level storage arose in systems due to the widely different access latencies and methods of the commodity storage devices:

- Fast, low capacity, volatile DRAM → working storage
- Slow, high capacity, non-volatile hard disk drives → persistent storage

Data from slow storage media is buffered in fast DRAM:

- After that it can be manipulated by programs → programs cannot directly access persistent storage
- It is the programmer’s job to translate this data between the two formats of the two-level storage (files and data structures)

Locating, transferring, and translating data and formats between the two levels of storage can waste significant energy and performance.
Opportunity: New Non-Volatile Memories

- Emerging memory technologies provide the potential for unifying storage and memory (e.g., Phase-Change, STT-RAM, RRAM)
  - **Byte-addressable** (can be accessed like DRAM)
  - **Low latency** (comparable to DRAM)
  - **Low power** (idle power better than DRAM)
  - **High capacity** (closer to Flash)
  - **Non-volatile** (can enable persistent storage)
  - **May have limited endurance** (but, better than Flash)

- Can provide fast access to *both* volatile data and persistent storage

- **Question**: if such devices are used, is it efficient to keep a two-level storage model?
Eliminating Traditional Storage Bottlenecks

- Today (DRAM + HDD) and two-level storage model
- Replace HDD with NVM (PCM-like), keep two-level storage model
- Replace HDD and DRAM with NVM (PCM-like), eliminate all OS+FS overhead

Results for PostMark
Where is Energy Spent in Each Model?

- HDD access wastes energy
- Additional DRAM energy due to buffering overhead of two-level model
- FS/OS overhead becomes important
- No FS/OS overhead
- No additional buffering overhead in DRAM

Results for PostMark

HDD Baseline  NVM Baseline  Persistent Memory
Our Proposal: Coordinated HW/SW Memory and Storage Management

- Goal: Unify memory and storage to eliminate wasted work to locate, transfer, and translate data
  - Improve both energy and performance
  - Simplify programming model as well
Our Proposal: Coordinated HW/SW Memory and Storage Management

- **Goal:** Unify memory and storage to eliminate wasted work to locate, transfer, and translate data
  - Improve both energy and performance
  - Simplify programming model as well

Before: Traditional Two-Level Store

- Virtual memory
  - Address translation
  - Main Memory

- Load/Store

- Processor and caches

- Operating system and file system
  - fopen, fread, fwrite, ...

- Storage (SSD/HDD)
Our Proposal: Coordinated HW/SW Memory and Storage Management

- Goal: Unify memory and storage to eliminate wasted work to locate, transfer, and translate data
  - Improve both energy and performance
  - Simplify programming model as well
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
The Persistent Memory Manager

- Persistent Memory Manager
  - Exposes a load/store interface to access persistent data
  - Manages data placement, location, persistence, security
  - Manages metadata storage and retrieval
  - Exposes hooks and interfaces for system software

- Example program manipulating a persistent object:

```c
1 int main(void) {
2   // data in file.dat is persistent
3   FILE myData = "file.dat";
4   myData = new int[64];
5 }
6 void updateValue(int n, int value) {
7   FILE myData = "file.dat";
8   myData[n] = value; // value is persistent
9 }
```
PMM uses access and hint information to allocate, locate, migrate and access data in the heterogeneous array of devices.

```c
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
}
```
Opportunities and Benefits

- We’ve identified at least five opportunities and benefits of a unified storage/memory system that gets rid of the two-level model:
  1. Eliminating system calls for file operations
  2. Eliminating file system operations
  3. Efficient data mapping/location among heterogeneous devices
  4. Providing security and reliability in persistent memories
  5. Hardware/software cooperative data management
Evaluation Methodology

- Hybrid real system / simulation-based approach
  - System calls are executed on host machine (functional correctness) and timed to accurately model their latency in the simulator
  - Rest of execution is simulated in Multi2Sim (enables hardware-level exploration)
- Power evaluated using McPAT and memory power models
- 16 cores, 4-wide issue, 128-entry instruction window, 1.6 GHz
- Volatile memory: 4GB DRAM, 4KB page size, 100-cycle latency
- Persistent memory
  - HDD (measured): 4ms seek latency, 6Gbps bus rate
  - NVM: (modeled after PCM) 4KB page size, 160-/480-cycle (read/write) latency
Evaluated Systems

- **HDD Baseline (HB)**
  - Traditional system with volatile DRAM memory and persistent HDD storage
  - Overheads of operating system and file system code and buffering

- **HDD without OS/FS (HW)**
  - Same as HDD Baseline, but with the ideal elimination of all OS/FS overheads
  - System calls take 0 cycles (but HDD access takes normal latency)

- **NVM Baseline (NB)**
  - Same as HDD Baseline, but HDD is replaced with NVM
  - Still has OS/FS overheads of the two-level storage model

- **Persistent Memory (PM)**
  - Uses only NVM (no DRAM) to ensure full-system persistence
  - All data accessed using loads and stores
  - Does not waste energy on system calls
  - Data is manipulated directly on the NVM device
Evaluated Workloads

- Unix utilities that manipulate files
  - `cp`: copy a large file from one location to another
  - `cp -r`: copy files in a directory tree from one location to another
  - `grep`: search for a string in a large file
  - `grep -r`: search for a string recursively in a directory tree

- PostMark: an I/O-intensive benchmark from NetApp
  - Emulates typical access patterns for email, news, web commerce

- MySQL Server: a popular database management system
  - OLTP-style queries generated by Sysbench
  - MySQL (simple): single, random read to an entry
  - MySQL (complex): reads/writes 1 to 100 entries per transaction
Performance Results

The workloads that see the greatest improvement from using a Persistent Memory are those that spend a large portion of their time executing system call code due to the two-level storage model.
Between systems with and without OS/FS code, energy improvements come from:
1. reduced code footprint, 2. reduced data movement

Large energy reductions with a PMM over the NVM based system
Scalability Analysis: Effect of PMM Latency

Even if each PMM access takes a non-overlapped 50 cycles (conservative), PMM still provides an overall improvement compared to the NVM baseline.

Future research should target keeping PMM latencies in check.
New Questions and Challenges

- We identify and discuss several open research questions

  - Q1. How to tailor applications for systems with persistent memory?

  - Q2. How can hardware and software cooperate to support a scalable, persistent single-level address space?

  - Q3. How to provide efficient backward compatibility (for two-level stores) on persistent memory systems?

  - Q4. How to mitigate potential hardware performance and energy overheads?
Single-Level Stores: Summary and Conclusions

- Traditional two-level storage model is inefficient in terms of performance and energy
  - Due to OS/FS code and buffering needed to manage two models
  - Especially so in future devices with NVM technologies, as we show

- New non-volatile memory based persistent memory designs that use a single-level storage model to unify memory and storage can alleviate this problem

- We quantified the performance and energy benefits of such a single-level persistent memory/storage design
  - Showed significant benefits from reduced code footprint, data movement, and system software overhead on a variety of workloads

- Such a design requires more research to answer the questions we have posed and enable efficient persistent memory managers
  - Can lead to a fundamentally more efficient storage system
New DRAM Architectures
Tolerating DRAM: Example Techniques

- Retention-Aware DRAM Refresh: Reducing Refresh Impact
- Tiered-Latency DRAM: Reducing DRAM Latency
- RowClone: Accelerating Page Copy and Initialization
- Subarray-Level Parallelism: Reducing Bank Conflict Impact
DRAM Refresh

- DRAM capacitor charge leaks over time

- The memory controller needs to refresh each row periodically to restore charge
  - Activate each row every N ms
  - Typical N = 64 ms

- Downsides of refresh
  - **Energy consumption**: Each refresh consumes energy
  - **Performance degradation**: DRAM rank/bank unavailable while refreshed
  - **QoS/predictability impact**: (Long) pause times during refresh
  - **Refresh rate limits DRAM capacity scaling**
Refresh Overhead: Performance

% time spent refreshing

Present

Future

Device capacity

2 Gb  4 Gb  8 Gb  16 Gb  32 Gb  64 Gb

8%  46%
Refresh Overhead: Energy

% DRAM energy spent refreshing

Present

Future

Device capacity

2 Gb 4 Gb 8 Gb 16 Gb 32 Gb 64 Gb

15% 47%

20% 40% 60% 80% 100%
Retention Time Profile of DRAM

- 64-128ms
- >256ms
- 128-256ms
RAIDR: Eliminating Unnecessary Refreshes

- Observation: Most DRAM rows can be refreshed much less often without losing data [Kim+, EDL’09][Liu+ ISCA’13]

- Key idea: Refresh rows containing weak cells more frequently, other rows less frequently
  1. Profiling: Profile retention time of all rows
  2. Binning: Store rows into bins by retention time in memory controller
     *Efficient storage with Bloom Filters* (only 1.25KB for 32GB memory)
  3. Refreshing: Memory controller refreshes rows in different bins at different rates

- Results: 8-core, 32GB, SPEC, TPC-C, TPC-H
  - 74.6% refresh reduction @ 1.25KB storage
  - ~16%/20% DRAM dynamic/idle power reduction
  - ~9% performance improvement
  - Benefits increase with DRAM capacity

Going Forward

- How to find out and expose weak memory cells/rows
  - Early analysis of modern DRAM chips:

- Low-cost system-level tolerance of DRAM errors

- Tolerating cell-to-cell interference at the system level
  - For both DRAM and Flash. Early analysis of Flash chips:
Tolerating DRAM: Example Techniques

- Retention-Aware DRAM Refresh: Reducing Refresh Impact
- Tiered-Latency DRAM: Reducing DRAM Latency
- RowClone: Accelerating Page Copy and Initialization
- Subarray-Level Parallelism: Reducing Bank Conflict Impact
DRAM latency continues to be a critical bottleneck
What Causes the Long Latency?

DRAM Latency = Subarray Latency + I/O Latency

Dominant Subarray Latency
Why is the Subarray So Slow?

- Long bitline
  - Amortizes sense amplifier cost → Small area
  - Large bitline capacitance → High latency & power
Trade-Off: Area (Die Size) vs. Latency

Long Bitline

Short Bitline

Faster

Smaller

Trade-Off: Area vs. Latency
Trade-Off: Area (Die Size) vs. Latency

Normalized DRAM Area vs. Latency (ns)

- 32 cells/bitline: Faster, Cheaper
- 64 cells/bitline: Faster
- 128 cells/bitline: Faster
- 256 cells/bitline: Faster
- 512 cells/bitline: Cheaper

Fancy DRAM Short Bitline
Commodity DRAM Long Bitline

GOAL: Cheaper and Faster
Approximating the Best of Both Worlds

Long Bitline
- Small Area
- High Latency

Our Proposal

Short Bitline
- Large Area
- Low Latency

Need Isolation

Add Isolation Transistors

Bitline ➔ Fast
Approximating the Best of Both Worlds

Long Bitline Tiered-Latency DRAM Short Bitline

Small Area

Small Area

Large Area

High Latency

Low Latency

Low Latency

Small area using long bitline

Low Latency
Tiered-Latency DRAM

- Divide a bitline into two segments with an isolation transistor

Commodity DRAM vs. TL-DRAM

- DRAM Latency ($t_{RC}$) • DRAM Power

- DRAM Area Overhead

$\sim 3\%$: mainly due to the isolation transistors
Trade-Off: Area (Die-Area) vs. Latency

- Cheaper
- Normalized DRAM Area
- Latency (ns)

- 32, 64, 128, 256, 512 cells/bitline
- Near Segment
- Far Segment

Goal
Leveraging Tiered-Latency DRAM

• TL-DRAM is a *substrate* that can be leveraged by the hardware and/or software

• Many potential uses

1. Use near segment as hardware-managed *inclusive* cache to far segment
2. Use near segment as hardware-managed *exclusive* cache to far segment
3. Profile-based page mapping by operating system
4. Simply replace DRAM with TL-DRAM
Using near segment as a cache improves performance and reduces power consumption
Tolerating DRAM: Example Techniques

- Retention-Aware DRAM Refresh: Reducing Refresh Impact
- Tiered-Latency DRAM: Reducing DRAM Latency
- RowClone: Accelerating Page Copy and Initialization
- Subarray-Level Parallelism: Reducing Bank Conflict Impact
Today’s Memory: Bulk Data Copy

1) High latency
2) High bandwidth utilization
3) Cache pollution
4) Unwanted data movement
Future: RowClone (In-Memory Copy)

1) Low latency
2) Low bandwidth utilization
3) No cache pollution
4) No unwanted data movement
DRAM operation (load one byte)

Step 1: Activate row

Step 2: Read transfer byte onto bus

Row Buffer (4 Kbits)

Data pins (8 bits)

Memory Bus
RowClone: in-DRAM Row Copy (and Initialization)

Step 1: Activate row A

Step 2: Activate row B

DRAM array

Row Buffer (4 Kbits)

Data pins (8 bits)

Memory Bus
RowClone: Latency and Energy Savings

RowClone: Overall Performance

Figure 10: Performance improvement of RowClone-ZI. Value on top indicates percentage improvement compared to baseline.

<table>
<thead>
<tr>
<th>Application</th>
<th>bootup</th>
<th>compile</th>
<th>mcached</th>
<th>mysql</th>
<th>shell</th>
</tr>
</thead>
<tbody>
<tr>
<td>Energy Reduction</td>
<td>40%</td>
<td>32%</td>
<td>15%</td>
<td>17%</td>
<td>67%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Number of Cores</th>
<th>2</th>
<th>4</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Workloads</td>
<td>138</td>
<td>50</td>
<td>40</td>
</tr>
<tr>
<td>Weighted Speedup Improvement</td>
<td>15%</td>
<td>20%</td>
<td>27%</td>
</tr>
<tr>
<td>Energy per Instruction Reduction</td>
<td>19%</td>
<td>17%</td>
<td>17%</td>
</tr>
</tbody>
</table>
Goal: Ultra-efficient heterogeneous architectures

CPU core
CPU core
CPU core
CPU core
mini-CPU core
video core
imaging core
GPU (throughput)
core
GPU (throughput)
core
GPU (throughput)
core
GPU (throughput)
core
LLC
Memory Controller
Memory Bus

Specialized compute-capability in memory

Slide credit: Prof. Kayvon Fatahalian, CMU
Enabling Ultra-efficient (Visual) Search

- What is the right partitioning of computation capability?
- What is the right low-cost memory substrate?
- What memory technologies are the best enablers?
- How do we rethink/ease (visual) search?

Picture credit: Prof. Kayvon Fatahalian, CMU
Tolerating DRAM: Example Techniques

- Retention-Aware DRAM Refresh: Reducing Refresh Impact
- Tiered-Latency DRAM: Reducing DRAM Latency
- RowClone: In-Memory Page Copy and Initialization
- Subarray-Level Parallelism: Reducing Bank Conflict Impact
SALP: Reducing DRAM Bank Conflicts

- **Problem:** Bank conflicts are costly for performance and energy
  - serialized requests, wasted energy (thrashing of row buffer, busy wait)
- **Goal:** Reduce bank conflicts without adding more banks (low cost)
- **Key idea:** Exploit the internal subarray structure of a DRAM bank to parallelize bank conflicts to different subarrays
  - Slightly modify DRAM bank to reduce subarray-level hardware sharing

- **Results on Server, Stream/Random, SPEC**
  - 19% reduction in dynamic DRAM energy
  - 13% improvement in row hit rate
  - 17% performance improvement
  - 0.15% DRAM area overhead

A Bit About Me and My Research
Brief Self Introduction

- **Onur Mutlu**
  - Carnegie Mellon University ECE/CS
  - PhD from UT-Austin 2006, worked at Microsoft Research, Intel, AMD
  - [http://www.ece.cmu.edu/~omutlu](http://www.ece.cmu.edu/~omutlu)
  - [onur@cmu.edu](mailto:onur@cmu.edu) (Best way to reach me)
  - [http://users.ece.cmu.edu/~omutlu/projects.htm](http://users.ece.cmu.edu/~omutlu/projects.htm)

- **Research, Teaching, Consulting Interests**
  - Computer architecture and systems, hardware/software interaction
  - Memory and storage systems, emerging technologies
  - Many-core systems, heterogeneous systems
  - Interconnects

Interested in developing efficient, high-performance, and scalable systems; solving difficult architectural problems at low cost & complexity...
What is SAFARI?

SAFARI is the research group of Professor Onur Mutlu in the Computer Architecture Lab (CALCM) at Carnegie Mellon University. We investigate safe, fair, robust and intelligent computer architecture, finding novel ways to provide a substrate with all of these properties for next-generation multicore and manycore systems.
Overview of My Recent Research

- Memory and storage systems: DRAM, Flash, NVM, emerging
  - Scalability, energy, latency, parallelism, fault tolerance
  - Compute in/near memory; emerging technologies

- Predictable performance, QoS

- Heterogeneous systems, accelerating bottlenecks

- Efficient system design: interconnects, cores, caches, …

- Bioinformatics algorithms and architectures

- Acceleration of important applications, software/hardware co-design
End of Backup Slides