RETHINKING MEMORY AND STORAGE FOR FUTURE COMPUTING SYSTEMS

Samira Khan
University of Virginia
April 5, 2016
DRAM is critical for performance
MAIN MEMORY TRENDS

1. Application pull
   - Increasing demand from applications

2. Technology push
   - Unfortunately DRAM scaling is ending
MAIN MEMORY TRENDS:
1. INCREASING DEMAND

Increasing demand for high performance, energy efficiency, and more capacity

1. Data-intensive applications
2. Exa-scale computing
DEMAND 1: DATA-INTENSIVE APPLICATIONS

Data to be analyzed is growing

Source: IDC 2012
EXAMPLE:
LARGE HADRON COLLIDER (LHC)

600 million collisions every second

1 PB/sec generated
1 out of 10000 collected
1 out of 100 selected

30 PB data analyzed per year
Increasing demand on memory
DEMAND 2: EXASCALE COMPUTING

Enables computational modeling, simulation, and prediction in science and technology

Can provide major improvement in different scientific fields
EXAMPLE:
AERODYNAMIC MODELING TREND

Reduces the number of physical tests for aircrafts

Increasing demand on memory

Source: Exascale Computing Report, DOE 2010
MAIN MEMORY TRENDS:
2. SCALING IS ENDING

MEGABITS/CHIP

START OF MASS PRODUCTION

2X/1.5 YEARS
2X/3 YEARS

DRAM scaling is getting difficult

Source: Flash Memory Summit 2013, Memcon 2014
DRAM SCALING CHALLENGE

Manufacturing reliable cells at low cost is getting difficult
IMPLICATION: DRAM ERRORS IN THE FIELD

1.52% of DRAM modules failed in Google Servers

A Study of DRAM Failures in the Field

1.6% of DRAM modules failed in LANL
GOAL

ENABLE HIGH CAPACITY MEMORY WITHOUT SACRIFICING RELIABILITY
TWO DIRECTIONS

Difficult to scale

ENABLE/RETHIN K DRAM

SIGMETRICS’14, SIGMETRICS’16, HPCA’15, DSN’15, DSN’16

Predicted to be highly scalable

LEVERAGE NEW TECHNOLOGIES

WEED’13, MICRO’15
MAIN MEMORY CHALLENGES

SYSTEM-LEVEL TECHNIQUES: IMPROVE DRAM SCALABILITY

NON-VOLATILE MEMORIES: UNIFIED MEMORY & STORAGE

WHAT ELSE CAN WE DO?
Can we make DRAM more scalable without any cost?
TRADITIONAL APPROACH TO ENABLE DRAM SCALING

Unreliable DRAM Cells → Make DRAM Reliable → Reliable DRAM Cells → Reliable System

Manufacturing Time

System in the Field

DRAM has strict reliability guarantee
OUR APPROACH

Unreliable DRAM Cells

Make DRAM Reliable

Reliable DRAM Cells

Reliable System

Manufacturing Time

System in the Field

Shift the responsibility to systems
VISION: SYSTEM-LEVEL DETECTION AND MITIGATION

Unreliable DRAM Cells

Detect and Mitigate

Reliable System

Detect and mitigate errors after the system has become operational

ONLINE PROFILING
Reduces cost, increases yield, and enables scaling
CHALLENGE: INTERMITTENT FAILURES

Unreliable DRAM Cells

Detect and Mitigate

Reliable System

If failures were permanent, a simple boot up test would have worked

How to detect and mitigate intermittent failures?
EFFICACY OF SYSTEM-LEVEL TECHNIQUES

Can we leverage existing techniques?

1. Testing
2. Guardbanding
3. Error Correcting Code
4. Higher Strength ECC

We analyze the effectiveness of these techniques using experimental data from real DRAMs.

Data set publicly available
METHODOLOGY

FPGA-based testing infrastructure

Evaluated 96 chips from three major vendors
Efficacy of Testing

- Only a few rounds can discover most of the failures.
- Even after hundreds of rounds, a small number of new cells keep failing.

Conclusion: Testing alone cannot detect all possible failures.
HIGHER STRENGTH ECC (HI-ECC)

No testing, use strong ECC
But amortize cost of ECC over larger data chunk

Can potentially tolerate errors at the cost of higher strength ECC
EFFECTIVENESS OF HI-ECC

Number of Rounds

4EC5ED, 2X Guardband
3EC4ED, 2X Guardband
DECTED, 2X Guardband
SECDED, 2X Guardband

Time to Failure (in years)

10 Years
EFFICACY OF HI-ECC

- 4EC5ED, 2X Guardband
- 3EC4ED, 2X Guardband
- DECTED, 2X Guardband
- SECDED, 2X Guardband

Time to Failure (in years)

Number of Rounds

10 Years
EFFICACY OF HI-ECC

After starting with 4EC5ED, can reduce to 3EC4ED code after 2 rounds of tests
Can reduce to DECTED code after 10 rounds of tests
Efficacy of HI-ECC

Can reduce to SECDED code, after 7000 rounds of tests (4 hours)

Conclusion: Testing can help to reduce the ECC strength, but *blocks memory for hours*
Key Observations:

• **Testing** alone cannot detect all possible failures

• **Testing** can help to reduce the **ECC strength**
  – Even when starting with a **higher strength ECC**
  – But degrades performance
TOWARDS AN ONLINE PROFILING SYSTEM

1. Initially Protect DRAM with Strong ECC
2. Periodically Test Parts of DRAM
3. Mitigate errors and reduce ECC

Run tests periodically after a short interval at smaller regions of memory
BREAKING THE ABSTRACTION

How to reduce test time?

How to schedule testing?

How to implement testing in the hardware?

What is the circuit-level characteristics of the failures?

- Samira Khan+, "The Efficacy of Error Mitigation Techniques for DRAM Retention Failures: A Comparative Experimental Study”, SIGMETRICS 2014
- Samira Khan+, "PARBOR: An Efficient System-Level Technique to Detect Data Dependent Failures in DRAM”, DSN 2016
MAIN MEMORY CHALLENGES

SYSTEM-LEVEL TECHNIQUES: IMPROVE DRAM SCALABILITY

WHAT ELSE CAN WE DO?

NON-VOLATILE MEMORIES: UNIFIED MEMORY & STORAGE
TWO-LEVEL STORAGE MODEL

CPU

MEMORY

DRAM

FILE

I/O

STORAGE

Ld/St

VOLATILE

FAST

BYTE ADDR

NONVOLATILE

SLOW

BLOCK ADDR
Non-volatile memories combine characteristics of memory and storage.
VISION: UNIFY MEMORY AND STORAGE

Provides an opportunity to manipulate persistent data directly
DRAM IS STILL FASTER

A hybrid unified memory-storage system
System crash can result in permanent data corruption in NVM
Add a node to a linked list

1. Link to next
2. Link to prev

System crash can result in inconsistent memory state
CURRENT SOLUTIONS

Explicit interfaces to manage consistency

– NV-Heaps [ASPLOS’11], BPFS [SOSP’09], Mnemosyne [ASPLOS’11]

Example Code

update a node in a persistent hash table

```c
void hashtable_update(hashtable_t* ht,
                     void *key, void *data)
{
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*) list_find(chain,
                                &updatePair);
    pair->second = data;
}
```
void `TMhashtable_update(TMARCGDECL hashtable_t* ht, void *key, void*data)`{
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*) TMLIST_FIND(chain, &updatePair);
    pair->second = data;
}
void TM_hashtable_update(TMARCGDECL hashtable_t* ht, void* key, void* data)
{
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*) TMLIST_FIND(chain, &updatePair);
    pair->second = data;
}
void TMhashtable_update(TMARCGDECL hashtable_t* ht, void *key, void*data){
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*) TMLIST_FIND (chain, &updatePair);
    pair->second = data;
}
CURRENT SOLUTIONS

Manual declaration of persistent components

```c
void TMIhashtable_update(TMARCDECL hashtable_t* ht, void* key, void* data){
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*) TMLIST_FIND (chain, &updatePair);
    pair->second = data;
}
```

Need a new implementation

Third party code can be inconsistent
CURRENT SOLUTIONS

Manual declaration of persistent components

```c
void TMhashtable_update(hashtable_t* ht, void *key, void*data){
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*)TMLIST_FIND(chain, &updatePair);
    pair->second = data;
}
```

Need a new implementation

Prohibited Operation

Third party code can be inconsistent

Burden on the programmers
OUR SOLUTION: ThyNVM

Software transparent consistency in persistent memory systems

- Execute *legacy applications*
- Reduce burden *on programmers*
- Enable *easier integration of NVM*
void hashtable_update(hashtable_t* ht, 
        void *key, void *data)
{
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*) list_find(chain, 
               &updatePair);
    pair->second = data;
}
RUN THE EXACT SAME CODE...

```c
void hashtable_update(hashtable_t* ht,
    void *key, void *data){
    list_t* chain = get_chain(ht, key);
    pair_t* pair;
    pair_t updatePair;
    updatePair.first = key;
    pair = (pair_t*) list_find(chain,
        &updatePair);
    pair->second = data;
}
```

Persistent Memory System

Software transparent memory crash consistency
ThyNVM APPROACH

Periodic checkpointing of data managed by hardware

Transparency to application and system
CHECKPOINTING OVERHEAD

1. Metadata overhead

Metadata Table

<table>
<thead>
<tr>
<th>Working location</th>
<th>Checkpoint location</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X'</td>
</tr>
<tr>
<td>Y</td>
<td>Y'</td>
</tr>
</tbody>
</table>

Running

Checkpointing

Running

Checkpointing

Epoch 0

Epoch 1

2. Checkpointing latency

49
1. DUAL GRANULARITY CHECKPOINTING

- **Page Writeback**
  - in DRAM

- **Block Remapping**
  - in NVM

**DRAM**

**GOOD FOR STREAMING WRITES**

**NVM**

**GOOD FOR RANDOM WRITES**

High write locality pages in DRAM, low write locality pages in NVM
2. OVERLAPPING CHECKPOINTING AND EXECUTION

Hides the long latency of Page Writeback
TRANSPARENT DATA CONSISTENCY

Cost of consistency compared to systems with zero-cost consistency

- Provides consistency without significant performance overhead

UNMODIFIED LEGACY CODE

<table>
<thead>
<tr>
<th></th>
<th>DRAM</th>
<th>NVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>-3.5%</td>
<td>+2.7%</td>
</tr>
</tbody>
</table>
What interface do applications use to manipulate persistent data directly?

Who provides consistency, persistency guarantee, instant recovery?

How does hardware manage the hybrid memory?

What is the circuit-level characteristics of the underlying technology?

- Jinglei Ren+, “Dual-Scheme Checkpointing: A Software-Transparent Mechanism for Supporting Crash Consistency in Persistent Memory Systems”, MICRO 2015
- Justin Meza+, “A Case for Efficient Hardware-Software Cooperative Management of Storage and Memory”, WEED 2013
MAIN MEMORY CHALLENGES

Technology Scaling

DRAM Cells

NON-VOLATILE MEMORIES:
UNIFIED MEMORY & STORAGE

SYSTEM-LEVEL TECHNIQUES:
IMPROVE DRAM LATENCY & SCALABILITY

WHAT ELSE CAN WE DO?

Detect and Mitigate

Reliable System

Non-Volatile Memory

UNIFY

Storage
Data movement is a major bottleneck
Move compute closer to memory
BREAKING THE ABSTRACTION

Which functions to offload?

What is the interface? Who provides data consistency?
How does hardware manage both data access and computation?
What is the circuit-level characteristics of the underlying technology that enables computations?
Take advantage of new technologies to move memory closer to compute
Which applications can take advantage of dense on-chip memory?

How to manage TBs of on-chip memory?

How should memory hierarchy look like when dense memory is on-chip?

What is the circuit-level characteristics of the underlying technology that enables the integration?
SUMMARY

Fix DRAM in innovative ways
- *Enable high density, reliable memory*

Integrate new memory technologies
- *Do not just replace, enhance*
- *Enable new applications*

Rethink memory and storage hierarchy
- *Take advantage of non-volatility and in-memory computation*
THANK YOU

QUESTIONS?
RETHINKING MEMORY AND STORAGE FOR FUTURE COMPUTING SYSTEMS

Samira Khan
University of Virginia
April 5, 2016

BEYOND CMOS WORKSHOP