Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-Core, Multi-Bank Systems

Heekwon Park
University of Pittsburgh

Seungjae Baek
University of Pittsburgh

Jongmoo Choi
Dankook University

Donghee Lee
University of Seoul

Sam H. Noh
Hongik University
Modern Computing Environment

‣ Multi-core Architecture
  • Increasing number of cores per socket
  • Sharing LLC (Last Level Cache) and main memory

‣ Multi-bank Architecture
  • Employing vast amount of main memory
  • Main memory (=Banks) is shared among multiple cores
  • Memory organization
    • Main memory consists of channels, ranks, banks, and rows
Components of main memory

- Channels, Ranks, & Banks

Relationship among components

- Channel : Multiple DIMMs
- DIMM : Multiple Ranks
- Rank : Multiple Chips or Banks
- Chip : Multiple Banks
  - A bank is distributed among multiple chips on same rank
Translation between VA and PA

- A program accesses memory via VA (Virtual Address)
- VA is translated to PA (Physical Address) by MMU (Memory Management Unit)
Translation between PA and DRAM IDs

• PA is translated to bank ID, rank ID, channel ID and row index by memory controller
Accessing DRAM chip
- Translated IDs determine location that contains data requested by the program
- Number of Banks is calculated by $n \times m \times c$
Each bank consists of a set of rows and a row-buffer

- Row stores program data
- Row-buffer is a cache area

Entire row is loaded into row-buffer before serving memory request
Row-buffer hit

- Access to a row that is present in the row-buffer
  - If data is cached in the row-buffer, the request can be served immediately from the row buffer

HIT
requested data \( \in \text{Row } n \)
Row-buffer = Row \( n \)

Access
a chunk
in Row 2

MEMORY CONTROLLER

R/W

Row-buffer
Row 0
Row 1
Row 2
Row 3
Row 4

Row 2

Row 2
Access to a row that is not present in the row-buffer

- DRAM controller performs precharge and activate operations
  - Precharge: Write back data in row-buffer into original row
  - Activate: Load data in row into row-buffer
- After the two operations, the request is served from the row-buffer
- Overhead is in 2~5 times more than row-buffer hit
Recent approaches to the row-buffer conflict problem

Hardware approach


Software approach


Assume a program alternates accesses between two virtual pages.

- Two ways to allocate the pages
  - Case 1: Allocate pages from same bank
  - Case 2: Allocate pages from different banks
- Case 1 is expected to cause more row-buffer conflicts than Case 2

Mapping between virtual page and physical page frames is governed by \textit{kernel-level memory allocator}.
Our Solution: \( \text{M}^3 \text{ Allocator} \)

- M-cube: **Multi-core Multi-bank Memory Allocator**
- Operating system approach to the row-buffer conflict problem
  - No hardware modification needed
- Kernel-level memory allocator
  - Memory container
  - Randomizing allocation
Outline

Understanding Your Memory Organization

Randomness & Memory Parallelism

M³ Design and Implementation

Performance Evaluation

Conclusion
Repeatedly accesses two variables **CL_1** and **CL_2**

- **CL_1** : Access fixed physical address
- **CL_2** : Access address in cache line size(64B) increments

Set CPU cache mode as uncacheable

Higher latency when two variables are in the same bank

Reveals memory controller address mapping pattern
Figure (d) ranges from 0 to 2,000,000 (roughly 128MB size)

Figure (c) zooms in on the 590,000 ~ 640,000 portion of Figure (d)

Figure (b) zooms in on a portion of iterations of Figure (c)

Figure (a) zooms in on a portion of iterations of Figure (b)
Analysis of Figure (a)

- Distinct pattern for every 3 addresses
  - 1st address: row-buffer conflict
  - 2nd & 3rd address: No conflict
- 3 consecutive cache lines are distributed in different banks
  - A page distributed in 3 rows
- "3" matches the number of channels in our system
- Implication: Channel interleaving is conducted on cache line
Analysis of Figure (d)

- 3MB pattern of Figure (c) occurs in 12MB intervals
  - 12MB ÷ 3MB = 4
  - “4” matches the number of ranks in our system
- Implication: rank **interleaving** is conducted on 3MB units
- 12MB pattern is repeated until the end of memory is reached
2 Key lessons from memory organization analysis

- Regularities repeated in 12MB patterns
  - Regularities at bank, rank, channel levels
- Every bank evenly used in 12MB units
  - Total **96 banks** and **32 rows** from each bank
  - **96 Banks × 32 Rows × 4KB = 12MB**

**12MB Size with Same Pattern**

*B0~7 : Bank 0 ~ Bank 7*
Regularity Considered Harmful

Case 1: Sequential Access Pattern

- All threads have same access pattern
  - Sequential pattern
- If a row-buffer conflict occurs, conflicts should keep occurring
Case 2: Random Access Pattern

- All threads have different access pattern
- The current row-buffer conflict does not affect the next conflict
Regularity Considered Harmful

- Figure (a): Sequential access pattern
  - Fluctuating performance
  - Correlated conflicts
    - Define as a set of successive conflict.
    - A conflict affects subsequent conflicts

- Figure (b): Random access pattern
  - Constant performance in most iterations
  - Conflicts by all 4 threads does not exist
  - Independent conflicts
    - Each conflict occurs independently from previous conflicts
  - Conflict probability = \( \frac{1}{\text{Total number of banks}} \)
Memory Parallelism

- How to maximize parallelism
  - Use as many banks as possible
  - Use the banks as evenly as possible

(a) Biased use of banks

(b) Evenly use all banks
Memory Parallelism

- Architectural consideration of our memory organization
  - Exclusively allocate 12MB to a core
    - 12MB is maximum size that evenly distributes all banks
M³ Allocator

› Design consideration
  • Reduce row-buffer conflict by randomizing page access pattern
  • Support parallelism by using as many banks as possible and using the banks as evenly as possible

› Key feature of M³ allocator
  • Propose novel randomizing algorithm to reduce row-buffer conflict
  • Introduce notion of memory container to improve parallelism
Randomizing Algorithm

- Non-deterministic page frame free (deallocation) sequence
  - Our algorithm based on this characteristic

- Two concepts of the algorithm
  - Individual page frame management
    - Keep randomness
  - Downward search
  - Disperse allocation sequence

(a) Traditional Buddy Algorithm

(b) Randomized Algorithm
Buddy Algorithm: Free mechanism

- Non-deterministic page frame free(deallocation) sequence
  - Random sequence of free request

- Traditional buddy algorithm
  - Two main data structures
    - Bitmap & Freelist
  - Always coalesce buddy pages
    - Pages are sorted in ascending order

Assumed Free Sequence:

1 → 3 → 0 → 2

Stack Heap Data Text

Page Frame:

* # : Page Frame

Bit map

Free list

Coalesce

Insert

Check

Buddy

Page Free Mechanism on Traditional Buddy Algorithm
Randomizing Algorithm: Free mechanism

- Same data structure but different management
  - No coalescing involved during free operation
- State of Freelist varies with free sequence
- Freed pages are randomly dispersed

Assumed Free Sequence

```
1 ➞ 3 ➞ 0 ➞ 2
```

Stack | Heap | Data | Text

![Page Free Mechanism on Randomized Algorithm](image)
Buddy algorithm
- Regular allocation sequence

Randomizing algorithm
- Use downward search to find a candidate page frame
- No splitting involved during allocating operation
- Each order (bin) of Freelist has double meaning
  - e.g.) order $2^2$ of Freelist can allocate 4 consecutive pages or less
- Individual page can be allocated
- Corresponding multiple pages can be allocated

(a) Allocation Sequence on Traditional Buddy Algorithm

(b) Allocation Sequence on Randomized Algorithm
Randomizing Algorithm

- Relationship between free and allocation sequence
  - **Buddy algorithm**: Same allocation sequence regardless of free sequence
  - **Randomized algorithm**: Different allocation sequence according to free sequence
Memory Container

- Consist of multiple consecutive page frames
- Exclusively assigned to a core
- Independently managed by randomized algorithm
  - Probability of accessing same bank: $\frac{1}{\text{Total number of banks}}$
- Dynamically assigned to a core

*MC # : Memory Container

Dynamically assigned

Statically configured at booting time
Memory Container

- MC size
  - OS compatibility
    - Power of 2
    - Optimal size for our system
      - 12MB
    - Largest divisor of 12MB
    - Smallest multiple of 12MB
  - Candidate size: 4MB

- Implementation of MC
  - Size: 4MB (1024 pages)
  - Same as maximum size of Linux buddy system
  - Total of 8192 MCs
Experimental Environment

Hardware
- IBM x3650 M2 server system
  - Two Intel XEON x5570 quad core processor
  - 32GB DDR3 main memory

Software
- Linux 2.6.32

<table>
<thead>
<tr>
<th>System Specifications</th>
<th>Architecture</th>
<th>NUMA</th>
</tr>
</thead>
<tbody>
<tr>
<td># of cores</td>
<td>16 core</td>
<td></td>
</tr>
<tr>
<td>Size of Memory</td>
<td>32GB</td>
<td></td>
</tr>
<tr>
<td># of node</td>
<td>2 nodes</td>
<td></td>
</tr>
<tr>
<td># of Bank</td>
<td>192 Banks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(96 banks per node)</td>
<td></td>
</tr>
<tr>
<td># of Rank</td>
<td>32 Ranks</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(16 ranks per node)</td>
<td></td>
</tr>
<tr>
<td># of Channel</td>
<td>6 channels</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(3 channels per node)</td>
<td></td>
</tr>
</tbody>
</table>

Operating System
- Distributed O/S: CentOS 5.2
- Kernel Version: Linux-2.6.32
<table>
<thead>
<tr>
<th>Category</th>
<th>Benchmark name</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Memory Intensive Benchmark</strong></td>
<td>RAMspeed</td>
<td>Measuring sustainable memory bandwidth</td>
</tr>
<tr>
<td></td>
<td>STREAM</td>
<td>Memory allocation and transfer speed</td>
</tr>
<tr>
<td></td>
<td>Sysbench-Memory</td>
<td>Utility to measure cache and memory performance</td>
</tr>
<tr>
<td><strong>CPU or I/O Intensive Benchmark</strong></td>
<td>Kernel Compile</td>
<td>Compiling the Linux kernel version 2.6.32</td>
</tr>
<tr>
<td></td>
<td>Unixbench</td>
<td>File system benchmark</td>
</tr>
<tr>
<td></td>
<td>Dbench</td>
<td>General-purpose benchmark</td>
</tr>
<tr>
<td><strong>Application Benchmark</strong></td>
<td>PARSEC</td>
<td>Representing diverse application domains</td>
</tr>
<tr>
<td>(Composed of 12 programs)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Overall benchmark results

- Performance improvement: 6.5% ~ 85.2%
- Relative to original Linux

Average Improvements

- RAMspeed: 42.1%
- STREAM: 26.6%
- Sysbench-Memory: 50.4%
Overall benchmark results

- Performance improvement: -0.6% ~ 7.8%

Average Improvements

- Kernel compile: 1.0%
- Unixbench: 0.9%
- Dbench: 1.9%
PARSEC Benchmark

- Overall benchmark results
  - Performance improvement: -2.9% ~ 21.7%

- Average Improvements
  - Group 1: 8.7%
    - Noticeable performance improvements
  - Group 2: 1.7%
    - Some performance improvements
  - Group 3: -0.01%
    - Does not show noticeable performance differences
Conclusion

The M³ Allocator

- A novel kernel-level memory allocator for multi-core, multi-bank systems

**Randomizing algorithm**

- Reduce cases where multiple cores access the same bank at the same time

**Memory Container**

- As many banks as possible
- As evenly as possible

} to maximize parallelism
Thank You

Q & A