Concurrent Data Structures for Near-Memory Computing

Zhiyu Liu (Brown)
Irina Calciu (VMware Research)
Maurice Herlihy (Brown)
Onur Mutlu (ETH)
Concurrent Data Structures

Are used everywhere: kernel, libraries, applications

Issues:
- Difficult to design and implement
- Data layout and memory/cache hierarchy play crucial role in performance
The Memory Wall

Data movement

CPU

L1 cache

L2 cache

L3 cache

Memory

2.2 GHz = 220K cycles during this time

< 10 ns

Tens of ns

Hundreds of ns

Data movement
Near Memory Computing

• Also called Processing In Memory (PIM)
• Avoid data movement by doing computation in memory
• Old idea
• New advances in 3D integration and die-stacked memory
• Viable in the near future
Near Memory Computing: Architecture

- Vaults: memory partitions
- PIM cores: lightweight
  - Fast access to its own vault
- Communication
  - Between a CPU and a PIM
  - Between PIMs
  - Via messages sent to buffers
Data Structures + Hardware

• Tight integration between algorithmic design and hardware characteristics

• Memory becomes an active component in managing data

• Managing data structures in PIM
  • Old work: pointer chasing for sequential data structures
  • Our work: concurrent data structures
Goals: PIM Concurrent Data Structures

1. How do PIM data structures compare to state-of-the-art concurrent data structures?

2. How to design efficient PIM data structures?

- LOW CONTENTION
  - Pointer-chasing

- HIGH
  - Cacheable
Goals: PIM Concurrent Data Structures

1. How do PIM data structures compare to state-of-the-art concurrent data structures?

2. How to design efficient PIM data structures?

LOW

CONTENTION

HIGH

Pointer-chasing

e.g., uniform distribution

skiplist & linkedlist

Cacheable
Pointer Chasing Data Structures
Goals: PIM Concurrent Data Structures

1. How do PIM data structures compare to state-of-the-art concurrent data structures?

2. How to design efficient PIM data structures?

- Pointer-chasing
  - e.g., uniform distribution
  - skiplist & linkedlist

- Cacheable
Naïve PIM Skiplist

PIM Memory Vault

Low Latency

add(30)
add(5)
delete(80)
Concurrent Data Structures

CPU

add(30)

CPU

add(5)

CPU

delete(80)

High Latency

DRAM
Flat Combining

Combiner lock

CPU

add(30)

CPU

add(5)

CPU

delete(80)

Sequential access

High Latency

DRAM
**Skiplist Throughput**

![Graph showing skiplist throughput with Intel Xeon E7-4850v3, 28 hardware threads, 2.2 GHz](image)

- **Ops/s** on the y-axis
- **Threads** on the x-axis

**Lock-free**

Intel Xeon E7-4850v3
28 hardware threads, 2.2 GHz
# PIM Performance

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>N</td>
<td>Size of the skiplist</td>
</tr>
<tr>
<td>p</td>
<td>Number of processes</td>
</tr>
<tr>
<td>L_CPU</td>
<td>Latency of a memory access from the CPU</td>
</tr>
<tr>
<td>L_LLCC</td>
<td>Latency of a LLC access</td>
</tr>
<tr>
<td>L_ATOMIC</td>
<td>Latency of an atomic instruction (by the CPU)</td>
</tr>
<tr>
<td>L_PIM</td>
<td>Latency of a memory access from the PIM core</td>
</tr>
<tr>
<td>L_MSG</td>
<td>Latency of a message from the CPU to the PIM core</td>
</tr>
</tbody>
</table>
PIM Performance

\[ \text{CPU} = r_1 \quad \text{PIM} = r_2 \quad \text{LLC} \quad r_1 = r_2 = 3 \]

\[ \text{MSG} = \text{CPU} \]
# PIM Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock-free</td>
<td>( \frac{p}{(\mathcal{B} \times \mathcal{L}_{CPU})} )</td>
</tr>
<tr>
<td>Flat Combining (FC)</td>
<td>( \frac{1}{(\mathcal{B} \times \mathcal{L}_{CPU})} )</td>
</tr>
<tr>
<td>PIM</td>
<td>( \frac{1}{(\mathcal{B} \times \mathcal{L}<em>{PIM} + \mathcal{L}</em>{MSG})} )</td>
</tr>
</tbody>
</table>

\( \mathcal{B} \) = average number of nodes accessed during a skiplist operation
Skiplist Throughput

Ops/s vs. Threads

- Lock-free
- PIM (expected)
- FC
Goals: PIM Concurrent Data Structures

1. How do PIM data structures compare to state-of-the-art concurrent data structures?

2. How to design efficient PIM data structures?

- LOW contention
  - Pointer-chasing
    - e.g., uniform distribution
    - skiplist & linkedlist

- CONTENTION

- HIGH contention
  - Cacheable
New PIM algorithm: Exploit Partitioning
PIM Skiplist w/ Partitioning
## PIM Performance

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock-free</td>
<td>$p / (B \times L_{CPU})$</td>
</tr>
<tr>
<td>Flat Combining (FC)</td>
<td>$1 / (B \times L_{CPU})$</td>
</tr>
<tr>
<td>PIM</td>
<td>$1 / (B \times L_{PIM} + L_{MSG})$</td>
</tr>
<tr>
<td>FC + $k$ partitions</td>
<td>$k / (B \times L_{CPU})$</td>
</tr>
<tr>
<td>PIM + $k$ partitions</td>
<td>$k / (B \times L_{PIM} + L_{MSG})$</td>
</tr>
</tbody>
</table>

$B = \text{average number of nodes accessed during a skiplist operation}$
Skiplist Throughput

- **PIM w/ 16 partitions (expected)**
- **PIM w/ 8 partitions (expected)**
- **Lock-free**
- **FC w/ 16 partitions**
- **FC w/ 8 partitions**
- **FC w/ 4 partitions**
- **FC**
Goals: PIM Concurrent Data Structures

1. How do PIM data structures compare to state-of-the-art concurrent data structures?

2. How to design efficient PIM data structures?

LOW

CONTENTION

HIGH

Pointer-chasing

Cacheable
e.g., FIFO queue
FIFO Queue

enqueue
TAIL
deque
HEAD

enqueue
PIM FIFO Queue

1. CPU Deq() retrieves request
2. Tail dequeues a node
3. CPU Deq() sends back the node

PIM Memory Vault
Pipelining

Can overlap the execution of the next request

Deq()

Timeline

1 2 3

1 2 3

1 2 3
Parallelize Enqs and Deqs
Conclusion

PIM is becoming feasible in the near future

We investigate Concurrent Data Structures (CDS) for PIM

Results:

• Naïve PIM data structures are less efficient than CDS

• New PIM algorithms can leverage PIM features
  • They outperform efficient CDS
  • They are simpler to design and implement
Thank you!

icalciu@vmware.com

https://research.vmware.com/