Compilers and Runtimes
Support for Explicitly Managed
Memory Hierarchies

Jaejin Lee

Center for Manycore Programming
School of Computer Science and Engineering
Seoul National University
jlee@cse.snu.ac.kr
http://aces.snu.ac.kr
Course Outline

1. Explicitly-managed memory hierarchies
2. Preliminaries
   - Caches and virtual memory
   - Cache coherence and memory consistency
   - Code overlays
3. FaCSim: a fast and cycle-accurate architecture simulator for embedded systems
4. Instruction scratchpad memory management techniques
   - Without a memory management unit
   - With a memory management unit
5. Software-managed caches for multicores with local memory
6. COMIC: an SVM for multicores with local memory
Explicitly-Managed Memory Hierarchies
Explicitly Managed Memory Hierarchies

- A hierarchy of distinct memories managed explicitly in software
  - Typically a small and fast local memory at the higher level (often called scratchpad memory) and a conventional high-latency and low bandwidth external memory at the lower level
  - It is the program’s responsibility to manage the physical layout and movement of code and data from the lower level to the higher level
  - The local memory is not virtualized by hardware address translation
- Found in high performance architectures, embedded processors, and DSPs
  - Sony/Toshiba/IBM Cell Broadband Engine, ARM processors, TI DSPs, etc.
Scratchpad Memory

- Found in embedded application processors and DSPs
- Belong to the level 1 memory subsystem
  - Typically 1 cycle access
  - Or on-chip SRAM memory (more cycles to access)
- Program addresses it explicitly
  - Not transparent to the program
- No hardware support for consistency
  - Software guarantees consistency
Cache vs. Scratchpad Memory

Cache vs. SPM: Energy Consumption

- 0.13 μ m, using CACTI
Heterogeneous Multicore Architectures

- Asymmetric Chip-Multiprocessors (ACMP)
- General-purpose cores (resource management) + accelerator cores (compute intensive)
- Shared main memory between cores
- No caches for accelerator cores
  - Often caches and cache coherence adversely affect the performance
  - No hardware coherence support for the local memory
- Data transfers between the local memory and the main memory
  - Typically through DMA operations
Amdahl’s Law

- p: the proportion of a program that can be parallelized
- 1 - p: the proportion of a program that cannot be parallelized
- n: the number of processors

\[
\text{speedup} = \frac{1}{(1 - p) + \frac{p}{n}}
\]

\[
\text{speedup} = \frac{1}{(1 - 0.6) + \frac{0.6}{4}} = \frac{1}{0.55} = 1.82
\]
Homogeneous Multicore vs. Heterogeneous Multicore

- A large processor on chip helps accelerate inherently sequential code segments
- Assume,
  - 30% of the program is inherently sequential
  - A single large core runs the sequential code twice as fast

\[
\text{speedup}_{\text{homogeneous}} = \frac{1}{0.3 + 0.7/16} = 2.91 \\
\text{speedup}_{\text{heterogeneous}} = \frac{1}{0.3/2 + 0.7/8} = 4.21
\]
Cell Broadband Engine

- Explicit movement of code and data between the external memory and local stores
General-Purpose GPU System

- Main memory vs. device global memory
- Explicit data transfers between them
- Managed by software
Preliminaries:
Caches and Virtual Memory
Principle of Locality

- The reuse of data or instructions that were recently used, or near those that have been used recently
  - Predictable behavior
- Temporal locality
  - Recently referenced items are likely to be referenced in the near future
  - Within relatively small time durations
- Spatial locality
  - Items in nearby locations tend to be referenced close together in time
  - Within relatively close locations and relatively small time durations
For Data

```
sum = 0;
for (i = 0; i < 10; i++)
    sum += A[i];
```

- Spatial locality
  - Reference array elements \(A[i]\) in succession (stride = 1)
- Temporal locality
  - Reference \(\text{sum}\) in each iteration
For Instructions

\[
\begin{align*}
\text{sum} &= 0; \\
\text{for} \ (i &= 0; \ i < 10; \ i++) \\
\text{sum} &= A[i];
\end{align*}
\]

- Spatial locality
  - Reference instructions in sequence
- Temporal locality
  - Cycle through loop repeatedly

```
sum = 0;
for (i = 0; i < 10; i++)
    sum += A[i];
```

```
movl $0, -12(%ebp)
movl $0, -16(%ebp)
jmp L2
L3:
    movl -16(%ebp), %eax
    movl -56(%ebp,%eax,4), %edx
    leal -12(%ebp), %eax
    addl %edx, (%eax)
    leal -16(%ebp), %eax
    incl (%eax)
L2:
    cmpl $9, -16(%ebp)
    jle L3
```
Memory Hierarchies

- Hierarchical arrangement of storage
  - To exploit locality of reference
- Fast storage technologies cost more per byte and have less capacity
- The gap between CPU and main memory speed is widening
Caching

- Exploit temporal locality
  - Remember the contents of recently accessed locations
- Exploit spatial locality
  - Remember the blocks of recently accessed locations
- Cache block = cache line
  - The basic unit for cache storage
  - Multiple bytes or words
- Need an item d, which is stored in some block b
  - Cache hit
    - Find block b in the cache at level k
  - Cache miss
    - Block b is not in the cache at level k
    - The cache at level k must fetch b from level k+1
      - If the cache at level k is full, then some block in the cache must be replaced
L1 Cache between CPU and Main Memory

CPU register file

L1 Cache
many w-word cache lines

Main Memory
divided into many w-word data blocks

1 ~ 16 bytes

cache line = data block

a data block
(8 ~ 512 bytes)
Cache Organizations in General

- Cache size = L × S × B bytes
- A set is a collection of cache locations in which a given block may be placed
Locating Data in the Cache

- The word at the requested address is in the cache if the tag bits in one of the valid lines in the specified set match the tag bits in the address.
- The set index is specified by the set index field of the address.
- The location of the word in the block is specified by the offset field in the address.

\[ S = 2^s \]
\[ B = 2^b \]
Direct-Mapped Caches

- One cache line per set
- Simplest
- Data block can be only in one place in the cache
- Replacement is straightforward
- Collisions between data blocks for the same cache line can occur
Addressing Direct-Mapped Caches

- Find a valid line in the selected set with a matching tag
- If there is one such line, extract the word with the offset field
- Otherwise, fetch the line from the lower level memory, place it in the selected set, and update the valid bit
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- $B = 8$ bytes/block, $S = 4$ sets, $L = 1$ line/set
- Address size = 6 bits
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000\textsubscript{2})
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 \( (000000_2) \)
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
0 \ (000000_2)
\]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000<sub>2</sub>)

```
<table>
<thead>
<tr>
<th>Tag (1 bit)</th>
<th>Set index (2 bits)</th>
<th>Offset (3 bits)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
```

Valid (1 bit) | Tag (1 bit) | Data (8 bytes) |
--------------|-------------|----------------|
Set 0         | Set 1       | Set 2          | Set 3
1 0           | 0           | 0              | 0
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- $B = 8 \text{ bytes/block, } S = 4 \text{ sets, } L = 1 \text{ line/set}$
- Address size = 6 bits

$0 (000000_2)$, $4 (000100_2)$

![Diagram showing direct-mapped cache addressing]

- Tag (1 bit)
- Set index (2 bits)
- Offset (3 bits)

- Valid (1 bit)
- Tag (1 bit)
- Data (8 bytes)

<table>
<thead>
<tr>
<th>Address</th>
<th>Valid</th>
<th>Tag</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>32</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>48</td>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>56</td>
<td>0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Set 0
Set 1
Set 2
Set 3
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000₂) 4 (000100₂) 20 (010100₂)
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
0 \ (000000_2) \quad 4 \ (000100_2) \quad 20 \ (010100_2)
\]

![Diagram showing memory mapping and address bits](image)
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[ 0 (000000_2) \quad 4 (000100_2) \quad 20 (010100_2) \]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

\[ 0 \ (000000_2) \ 4 \ (000100_2) \ 20 \ (010100_2) \]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
0 (000000_2) \quad 4 (000100_2) \quad 20 (010100_2) \quad 48 (110000_2)
\]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
\begin{align*}
0 \ (000000_2) & \quad 4 \ (000100_2) & \quad 20 \ (010100_2) & \quad 48 \ (110000_2)
\end{align*}
\]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
\begin{align*}
0 \ (000000_2) & \quad 4 \ (000100_2) & \quad 20 \ (010100_2) & \quad 48 \ (110000_2) \\
\end{align*}
\]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
0 \ (000000_2) \quad 4 \ (000100_2) \quad 20 \ (010100_2) \quad 48 \ (110000_2)
\]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- $B = 8$ bytes/block, $S = 4$ sets, $L = 1$ line/set
- Address size = 6 bits

$$0 \ (000000_2) \ 4 \ (000100_2) \ 20 \ (010100_2) \ 48 \ (110000_2)$$
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
0 \ (000000_2) \quad 4 \ (000100_2) \quad 20 \ (010100_2) \quad 48 \ (110000_2) \quad 36 \ (100100_2)
\]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[
0 \ (000000_2) \ 4 \ (000100_2) \ 20 \ (010100_2) \ 48 \ (110000_2) \ 36 \ (100100_2)
\]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (000000₂) 4 (000100₂) 20 (010100₂) 48 (110000₂) 36 (100100₂)
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- \( B = 8 \) bytes/block, \( S = 4 \) sets, \( L = 1 \) line/set
- Address size = 6 bits

\[ \begin{align*}
0 \ (000000_2) & \quad 4 \ (000100_2) & \quad 20 \ (010100_2) & \quad 48 \ (110000_2) & \quad 36 \ (100100_2)
\end{align*} \]
Addressing Direct-Mapped Caches (contd.)

- Lower level memory size = 64 bytes
- B = 8 bytes/block, S = 4 sets, L = 1 line/set
- Address size = 6 bits

0 (0000000₂) 4 (0001000₂) 20 (0101000₂) 48 (1100000₂) 36 (1001000₂)
Set Associative Caches

- Data block can be in a few places in the cache
  - Need a good replacement policy
  - Less collisions between data blocks for the same cache line than the direct-mapped cache
- Complex tag comparison hardware on the lines in a set
Addressing Set Associative Caches

- Find a valid line in the selected set with a matching tag
- If there is one such line, extract the word with the offset field
- Otherwise, fetch the line from the lower level memory, place it in the selected set by deciding which line should be used, and update the valid bit
- Need a sophisticated replacement policy
Fully Associative Caches

- Only one set
- Data block can be any place in the cache
  - Less collisions between data blocks for the same cache line than the set associative cache
- Complex tag comparison hardware on the lines in the cache
Addressing Fully Associative Caches

- Find a valid line with a matching tag
- If there is one such line, extract the word with the offset field
- Otherwise, fetch the line from the lower level memory, place it in the cache by deciding which line should be used, and update the valid bit
- Need a sophisticated replacement policy
Types of Cache Misses

- Cold (compulsory) miss
  - When the cache is empty
- Conflict miss
  - When the cache is large enough, but multiple data items map to the same cache line
- Capacity miss
  - When the set of active cache lines (working set) is larger than the cache
- Working set
  - The set of referenced blocks that are active during a given period of time
Replacement Policies

- After a miss, what cache block should be replaced with the block read from memory?
- Which way in a multiway (i.e., set associative or fully associative) cache should be replaced?
- Ideally, any cached data which is no longer needed would be chosen to be replaced
- LRU (Least Recently Used)
- Pseudo LRU
- FIFO (First In, First Out)
  - Select a block that has been in the set for the longest time
- Random
Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode $4! = 24$ states in a set
    - 8-way set associative cache: 16 bits to encode $8! = 40320$ states in a set
    - ...
  - A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
    - Too costly
  - Instead, use pseudo LRU

...
Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode 4! = 24 states in a set
    - 8-way set associative cache: 16 bits to encode 8! = 40320 states in a set
    - ...
  - A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
    - Too costly
  - Instead, use pseudo LRU

... A
Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode 4! = 24 states in a set
    - 8-way set associative cache: 16 bits to encode 8! = 40320 states in a set
    - ...
  - A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
    - Too costly
  - Instead, use pseudo LRU

... A
Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode 4! = 24 states in a set
    - 8-way set associative cache: 16 bits to encode 8! = 40320 states in a set
    - ...
  - A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
    - Too costly
  - Instead, use pseudo LRU

... A

```
|   |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| 1 |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
| B  |
+---+---+---+---+---+---+---+---+
| 1  |   |   |   |   |   |   |   |
+---+---+---+---+---+---+---+---+
```

0
Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode $4! = 24$ states in a set
    - 8-way set associative cache: 16 bits to encode $8! = 40320$ states in a set
    - ...

- A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
  - Too costly
  - Instead, use pseudo LRU

... A B

\[
\begin{array}{c|cccc|c}
A & 0 & 0 & 0 & 0 & 1 \\
B & 1 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode $4! = 24$ states in a set
    - 8-way set associative cache: 16 bits to encode $8! = 40320$ states in a set
    - ...
  - A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
    - Too costly
    - Instead, use pseudo LRU

... A B

```
A 1
B 1
```
Least Recently Used (LRU)

- Select a block that has not been used for the longest time
  - Need to maintain LRU statistics for each cache line in a set
    - 2-way set associative cache: 1 bit to encode 2 states in a set
    - 4-way set associative cache: 5 bits to encode $4! = 24$ states in a set
    - 8-way set associative cache: 16 bits to encode $8! = 40320$ states in a set
    - ...
  - A time consuming read/modify/write cycle is needed to maintain the set state on a cache access
    - Too costly
  - Instead, use pseudo LRU

... A B
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
  - N-way set associative cache: \((2^{\log_2N + 1} - 1) - N\) bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

<table>
<thead>
<tr>
<th>access</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11_</td>
</tr>
<tr>
<td>B</td>
<td>10_</td>
</tr>
<tr>
<td>C</td>
<td>0_1</td>
</tr>
<tr>
<td>D</td>
<td>0_0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>00X</td>
<td>A</td>
</tr>
<tr>
<td>01X</td>
<td>B</td>
</tr>
<tr>
<td>1X0</td>
<td>C</td>
</tr>
<tr>
<td>1X1</td>
<td>D</td>
</tr>
</tbody>
</table>
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
  - \(N\)-way set associative cache: \((2^{\log_2 N} + 1) - 1\) - \(N\) bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

<table>
<thead>
<tr>
<th>access</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11_</td>
</tr>
<tr>
<td>B</td>
<td>10_</td>
</tr>
<tr>
<td>C</td>
<td>0_1</td>
</tr>
<tr>
<td>D</td>
<td>0_0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>00X</td>
<td>A</td>
</tr>
<tr>
<td>01X</td>
<td>B</td>
</tr>
<tr>
<td>1X0</td>
<td>C</td>
</tr>
<tr>
<td>1X1</td>
<td>D</td>
</tr>
</tbody>
</table>
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
  - N-way set associative cache: \((2^{\log_2 N} + 1) - 1\) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

\[
\begin{array}{c|c}
\text{access} & \text{next state} \\
\hline
A & 11_1 \\
B & 10_1 \\
C & 0_1 \\
D & 0_0 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{state} & \text{replace} \\
\hline
00X & A \\
01X & B \\
1X0 & C \\
1X1 & D \\
\end{array}
\]
Pseudo LRU

- A binary decision tree
- 2-way set associative cache: 1 bit
- 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
- \(N\)-way set associative cache: \((2^{\log_2{N}} + 1) - 1\) - \(N\) bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

```

<table>
<thead>
<tr>
<th>access</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11_</td>
</tr>
<tr>
<td>B</td>
<td>10_</td>
</tr>
<tr>
<td>C</td>
<td>0_1</td>
</tr>
<tr>
<td>D</td>
<td>0_0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>state</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>00X</td>
<td>A</td>
</tr>
<tr>
<td>01X</td>
<td>B</td>
</tr>
<tr>
<td>1X0</td>
<td>C</td>
</tr>
<tr>
<td>1X1</td>
<td>D</td>
</tr>
</tbody>
</table>
```

... A
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
  - N-way set associative cache: \((2^{\log_2 N + 1} - 1) - N\) bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

\[
\begin{array}{c|c}
\text{access} & \text{next state} \\
\hline
A & 11_1 \\
B & 10_1 \\
C & 0_1 \\
D & 0_0 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{state} & \text{replace} \\
\hline
00X & A \\
01X & B \\
1X0 & C \\
1X1 & D \\
\end{array}
\]
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
  - N-way set associative cache: \((2^{\log_2 N + 1} - 1) - N\) bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

... A C
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \(2^3 - 1\) - 4 = 3 bits
  - N-way set associative cache: \(2^{\log_2 N + 1} - 1\) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

\[
\begin{array}{c|c}
\text{access} & \text{next state} \\
\hline
A & 11_2 \\
B & 10_2 \\
C & 0_1 \\
D & 0_0 \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{state} & \text{replace} \\
\hline
00X & A \\
01X & B \\
1X0 & C \\
1X1 & D \\
\end{array}
\]
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
  - \(N\)-way set associative cache: \((2^{\log_2 N + 1} - 1) - N\) bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

<table>
<thead>
<tr>
<th>access</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11_</td>
</tr>
<tr>
<td>B</td>
<td>10_</td>
</tr>
<tr>
<td>C</td>
<td>0_1</td>
</tr>
<tr>
<td>D</td>
<td>0_0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>state</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>00X</td>
<td>A</td>
</tr>
<tr>
<td>01X</td>
<td>B</td>
</tr>
<tr>
<td>1X0</td>
<td>C</td>
</tr>
<tr>
<td>1X1</td>
<td>D</td>
</tr>
</tbody>
</table>

... A C
Pseudo LRU

- A binary decision tree
  - 2-way set associative cache: 1 bit
  - 4-way set associative cache: \((2^3 - 1) - 4 = 3\) bits
  - N-way set associative cache: \((2^{\log_2 N} + 1) - 1\) - N bits
- The difference between pseudo LRU and true LRU is statistically small
- Each bit represents the left or right child in the binary decision tree
  - 1: the left side has been referenced more recently than the right side
  - 0: vice versa
- A write cycle to update the pseudo-LRU bits on a hit
- A read cycle for the pseudo-LRU bits during a line replacement

```
<table>
<thead>
<tr>
<th>access</th>
<th>next state</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>11_</td>
</tr>
<tr>
<td>B</td>
<td>10_</td>
</tr>
<tr>
<td>C</td>
<td>0_1</td>
</tr>
<tr>
<td>D</td>
<td>0_0</td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>state</th>
<th>replace</th>
</tr>
</thead>
<tbody>
<tr>
<td>00X</td>
<td>A</td>
</tr>
<tr>
<td>01X</td>
<td>B</td>
</tr>
<tr>
<td>1X0</td>
<td>C</td>
</tr>
<tr>
<td>1X1</td>
<td>D</td>
</tr>
</tbody>
</table>
```

![Diagram of binary decision tree and pseudo-LRU bits]

... A C
Write Policies

- For reads, the block can be read at the same time that the tag is compared
  - If a miss, just ignore the value read
- For writes, modifying the block cannot begin until the tag is compared
  - Only some part of the entire block is modified
## Write Policies when a Hit

<table>
<thead>
<tr>
<th>Write through</th>
<th>Write back</th>
</tr>
</thead>
</table>
| • Both the block in the cache and the block in the lower level memory are modified | • Only the block in the cache is modified
| • Simpler to implement | • Harder to implement
| • Writes are slower than reads | • Writes and reads are preformed at the same speed
| • The lower level memory is always consistent with the cache | • The lower level memory is not always consistent with the cache
| • Every write requires the lower level memory access (need more memory bandwidth) | • Multiple writes within a block require only one write to the lower level memory (need less memory bandwidth)
| • Read misses never result in writes to the lower level memory | • Read misses may cause writes of dirty blocks to the lower level memory due to replacement

### Diagram:

- **Dirty**: (1 bit)
- **Valid**: (1 bit)
- **Tag**: (t bits)
- **Data**: (B bytes)

---

*ACACES 2009*
Write Policies when a Miss

- Write allocate
  - The block is loaded into the cache on a write miss
- No write allocate
  - The block is modified in the lower level memory and not loaded into the cache

<table>
<thead>
<tr>
<th>Write through and write allocate</th>
<th>Write back and write allocate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subsequent writes to the same block will generate a write to the lower level memory anyway</td>
<td>On a miss it updates the block in the lower level memory and brings the block to the cache</td>
</tr>
<tr>
<td>Bringing the block in the cache is a waste of time</td>
<td>Subsequent writes to the same block will hit in the cache</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write through and no write allocate</th>
<th>Write back and no write allocate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Not bringing the block in the cache on a miss saves time</td>
<td>Subsequent writes to the same block will generate misses</td>
</tr>
</tbody>
</table>
Non-Blocking/Lockup-Free Caches

- Most caches can handle only one outstanding request at a time
  - On a miss, the cache must wait for the lower level memory to supply the requested data and until then it is blocked
- A non-blocking cache continues to supply cache hits during a miss
  - Reduce effective miss penalty
  - Another option: supporting multiple outstanding misses
    - A special state need to be maintained for each outstanding miss
      - Miss Status/Information Holding Registers (MSHRs)
Cache Performance Metrics

- Miss Rate
  - Fraction of memory references not found in the cache (misses/references)

- Hit Time
  - Time to deliver a line in the cache to the processor (includes time to determine whether the line is in the cache)

- Miss Penalty
  - Additional time required due to the miss
Virtual Memory

- An abstraction of main memory by the operating system
  - Provide each process with a large and uniform address space
    - The size of the address space is bigger than that of main memory
  - Protect the address space of each process from corruption by other processes
  - Treat main memory as a cache of the permanent secondary storage (hard disk)
MMU and Pages

- Each byte in main memory has a unique physical address (PA)
- The CPU generates a virtual address (VA) to access the main memory
- The memory management unit (MMU) translates the virtual address to the corresponding physical address using a look-up table (page table) stored in main memory
- The virtual address space is divided into uniform virtual pages
  - Each page is indexed by its virtual page number
- The physical memory is divided into uniform physical pages (page frames)
  - Each frame is indexed by its page frame number
Page Tables

- Map virtual pages to physical pages
  - An array of page table entries (PTE)
  - A PTE consists of a valid bit and an n-bit address field (physical page frame number or secondary storage address) in addition to other page attributes
- The MMU reads the page table when it converts VA to PA
- The OS (page fault handler) takes care of maintaining the contents of the page table and transferring pages between main memory and secondary storage
- Swapping (paging)
  - The activity of transferring a page between the secondary storage and main memory
- Demand paging
  - Wait until the last moment to swap in a page when a miss (page fault) occurs
Address Translation

- Three types of virtual pages
  - Unallocated: Pages that have not yet been allocated by the VM (no space on secondary storage)
  - Cached: Allocated pages that are currently cached in main memory
  - Uncached: Allocated pages that are not cached in main memory (reside on secondary storage)

- A single page table for the entire address space is large
  - 32-bit address space, 4KB pages, and 4B PTEs result in 4MB page table resident in main memory
  - Use a hierarchy of page tables and demand paging for the tables
Page Hit and Page Fault

**Hit**

1. Virtual address

   ---

2. PTE address

   ---

3. PTE

   ---

4. Physical address

   ---

5. Data

**Miss**

6. Requested page

   ---

7. PTE update

   ---

8. Return to the original process, restart the faulting instruction

   ---

9. Virtual address

   ---

10. Virtual address (then a hit)

   ---

11. PTE

   ---

12. Physical address

   ---

13. Data

   ---

   (5) Victim page (if dirty)

   ---

   (6) Requested page

   ---

   (12) Physical address

   ---

   (13) Data
Page Replacement Policies

- LRU
- FIFO
- Second chance
- Clock
  - A bit (R) that indicates whether the page is referenced or not
    - When a page is first loaded in memory, R = 0
    - When the page is referenced, R = 1
  - Maintain a circular list of pages in memory
    - The hand points to the current page in the list
    - When it is time to replace a page, the first frame with R = 0 encountered is replaced
    - During the search for replacement, each reference bit set to 1 is changed to 0
Translation Lookaside Buffer (TLB)

- Every time the CPU generates a virtual address, the MMU must refer to the page table for address translation
  - High overhead
- A small, virtually addressed cache where each line holds a block consisting of a single PTE
  - Has a high degree of associativity
- Micro-TLB
  - A small TLB placed over the main TLB to boost the speed of address translation for cache accesses
  - The main TLB handles micro-TLB misses
  - Smaller number of entries than the main TLB
Caches and Virtual Memory

- Virtually-addressed caches vs. physically-addressed caches
  - Which address do we send to the cache?
  - Virtually-addressed cache: faster (no address translation) but security issues (requires cache flushing by the OS on context switching)
  - Physically-addressed cache: slower but no security issues (no OS intervention)
- Four possible combinations
  - Physically indexed, physically tagged
  - Physically indexed, virtually tagged
  - Virtually indexed, physically tagged
  - Virtually indexed, virtually tagged
Preliminaries: Cache Coherence and Memory Consistency
Cache Coherence Problem

- Caching is vital to reducing memory latency in multiprocessor systems
- Private caches in a multiprocessor system may create a coherence problem
  - Copies of a variable can be present in multiple caches
  - A write by one processor may not become visible to others
    - The result of the write are not observed by others
      - Stale values
Cache Coherence Problem (contd.)

- Assume write back, write allocate caches
- Core 0 and core 2 read location X, then Core 1 writes location X
- Core 0, Core 2 and memory have stale copies of X
Cache Coherence Problem (contd.)

- Assume write back, write allocate caches
- Core 0 and core 2 read location X, then Core 1 writes location X
- Core 0, Core 2 and memory have stale copies of X
Cache Coherence Problem (contd.)

- Assume write back, write allocate caches
- Core 0 and core 2 read location X, then Core 1 writes location X
- Core 0, Core 2 and memory have stale copies of X
Cache Coherence Problem (contd.)

- Assume write back, write allocate caches
- Core 0 and core 2 read location X, then Core 1 writes location X
- Core 0, Core 2 and memory have stale copies of X
Cache Coherence Problem (contd.)

- Some action must be taken
  - Update or invalidate (more common)
- Core 3 reads location X
- Core 1 supplies the updated copy of X to Core 3

```
Interconnect
```

```
<table>
<thead>
<tr>
<th>Core 0</th>
<th>Core 1</th>
<th>Core 2</th>
<th>Core 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>$</td>
<td>$</td>
<td>$</td>
</tr>
<tr>
<td>X: 46</td>
<td>X: 64</td>
<td>X: 46</td>
<td></td>
</tr>
</tbody>
</table>
```

```
Memory
```

```
X: 46
```
Cache Coherence Problem (contd.)

- Some action must be taken
  - Update or invalidate (more common)
- Core 3 reads location X
- Core 1 supplies the updated copy of X to Core 3
Cache Coherence Problem (contd.)

- Some action must be taken
  - Update or invalidate (more common)
- Core 3 reads location X
- Core 1 supplies the updated copy of X to Core 3
Cache Coherence Problem (contd.)

- Some action must be taken
  - Update or invalidate (more common)
- Core 3 reads location X
- Core 1 supplies the updated copy of X to Core 3
Cache Coherence Problem (contd.)

- Some action must be taken
  - Update or invalidate (more common)
- Core 3 reads location X
- Core 1 supplies the updated copy of X to Core 3
Cache Coherence Problem (contd.)

- Hardware cache coherence protocols
  - Snoopy and directory-based protocols
  - MSI, MESI, MOESI, etc.
- Shared caches do not have a coherence problem
- Easy in uniprocessors except I/O
- Coherence problems between I/O devices and the processor caches
  - Uncacheable memory region, uncachable operations, flushing pages, passing I/O data through caches, etc.
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols

![Diagram showing cache blocks and interconnect](image-url)
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols

![Diagram of Core Interconnects](image-url)
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
False Sharing

- Invalidations can lead to problems with false sharing
  - Different cores write to different locations in the same cache block
  - Force the cache line to ping-pong back and forth between the two cores
- Not occur in update-based protocols
A Motivating Example

- Expect memory to respect order between accesses to different locations in a thread

```c
data = 0;
done = false;

thread 0
data = 5;
done = true;

thread 1
while (not done);
print data;
```
The Effect of Reordering

data = 0;
done = false;

thread 0

data = 5;
done = true;

thread 1

while (not done);
print data;

execution 1

data = 5;
done = true;
while (not done);
print data;

prints 5

execution 2

done = true;
while (not done);
print data;
data = 5;

prints 0
Coherence Helps?

- Different orders of data accesses to shared memory yield different execution outcomes
- Data accesses may be reordered by
  - Compiler optimizations
  - Underlying architectures
- Coherence does not help
  - Only to a single location (i.e., a single cache block)
Memory Consistency Models

- Specify constraints on the order in which memory operations (from any processor) can appear to execute with respect to one another
- To balance programming complexity and performance
  - Used by the programmer to reason about correctness and possible results of a program
  - Used by the system designer to constrain how much accesses can be reordered by a compiler or hardware
- Contract between the programmer and the multiprocessor system
Memory Consistency Models

- Sequential consistency
- Relaxed memory consistency models
  - Processor consistency
  - Weak ordering
  - Release consistency
Sequential Consistency

- The observable outcome of a multithreaded program $P$ is the same as the outcome of a single thread executing all operations in $P$
- A total order between operations is defined (atomic operations)
- In this total order, two operations from the same thread of $P$ are executed in the order specified in $P$
Sequential Consistency (contd.)

- A multiprocessor system is sequentially consistent if the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program [Lamport, IEEE TOC, 1979]

- Known to be inefficient with hardware implementation
Program Order

- Order in which operations appear in source code
- Straightforward translation of source code to assembly code
- At most one memory operation per instruction
- But not the same as the order presented to hardware by the compiler
- Locally: specified by the programmer
- Globally: determined at run time
Possible Executions under SC

- Operations are atomic
- Interleaving semantics

\[
x=0; \ y = 0; \ x = 0; \ y = 0;
\]

\[
\text{thread 0}
\]

\[
\text{thread 1}
\]

\[
\begin{align*}
S11 & : X = x; \\
S12 & : y = 1;
\end{align*}
\]

\[
\begin{align*}
S21 & : y = y; \\
S22 & : x = 1;
\end{align*}
\]

\[
\text{print} \ X, \ Y;
\]

\[
\begin{array}{c}
S11 \quad \downarrow \\
S12 \quad \downarrow \\
S21 \quad \downarrow \\
S22 \quad \downarrow \\
X:0 \quad Y:1 \\
Y:0 \quad X:0
\end{array}
\]

\[
\begin{array}{c}
S11 \quad \downarrow \\
S12 \quad \downarrow \\
S21 \quad \downarrow \\
S22 \quad \downarrow \\
X:0 \quad Y:0 \\
Y:0 \quad X:0
\end{array}
\]

\[
\begin{array}{c}
S11 \quad \downarrow \\
S12 \quad \downarrow \\
S21 \quad \downarrow \\
S22 \quad \downarrow \\
X:0 \quad Y:0 \\
Y:0 \quad X:0
\end{array}
\]

\[
\begin{array}{c}
S11 \quad \downarrow \\
S12 \quad \downarrow \\
S21 \quad \downarrow \\
S22 \quad \downarrow \\
X:0 \quad Y:1 \\
Y:0 \quad X:1
\end{array}
\]

\[
\begin{array}{c}
S11 \quad \downarrow \\
S12 \quad \downarrow \\
S21 \quad \downarrow \\
S22 \quad \downarrow \\
X:1 \quad Y:1 \\
Y:1 \quad X:1
\end{array}
\]

Unordered, but sequentially consistent

Not sequentially consistent
Reasoning Based on SC

- If \( X \) is equal to 1 then \( Y \) should be 0
  - \( X \) is equal to 1 implies S22 is executed before S11
  - This implies S21 is executed before S12 due to the program order

\[
\begin{align*}
x &= 0; \quad y = 0; \quad X = 0; \quad Y = 0; \\
\text{thread 0} & \quad \text{thread 1} \\
S11: X &= x; \quad S21: Y = y; \\
S12: y &= 1; \quad S22: x = 1; \\
\text{print } X, Y;
\end{align*}
\]

\[
\begin{align*}
S11: X &= x \\
S12: y &= 1 \\
S21: Y &= y \\
S22: x &= 1 \\
\text{print } X, Y;
\end{align*}
\]
Reasoning Based on SC

- If X is equal to 1 then Y should be 0
- X is equal to 1 implies S22 is executed before S11
- This implies S21 is executed before S12 due to the program order

\[ x=0; \ y = 0; \ X = 0; \ Y = 0; \]

thread 0

\[ S11: \ X = x; \]
\[ S12: \ y = 1; \]

thread 1

\[ S12: \ y = 1 \]
\[ S21: \ Y = y; \]
\[ S22: \ x = 1; \]

print X, Y;

\[ S11: \ X = x \]
\[ S12: \ y = 1 \]
\[ S21: \ Y = y \]
\[ S22: \ x = 1 \]

\[ X: 0, \ Y: 0 \]
\[ X: 0, \ Y: 1 \]
\[ X: 0, \ Y: 1 \]
\[ X: 1, \ Y: 1 \]
Reasoning Based on SC

- If X is equal to 1 then Y should be 0
  - X is equal to 1 implies S22 is executed before S11
  - This implies S21 is executed before S12 due to the program order

\[
\begin{align*}
x &= 0; \ y &= 0; \ X &= 0; \ Y &= 0; \\
\text{thread 0} & \quad \text{thread 1} \\
S11: \ X &= x; \quad S21: \ Y &= y; \\
S12: \ y &= 1; \quad S22: \ x &= 1; \\
\text{print X, Y;}
\end{align*}
\]
Reasoning Based on SC

- If X is equal to 1 then Y should be 0
  - X is equal to 1 implies S22 is executed before S11
  - This implies S21 is executed before S12 due to the program order

```
x=0; y = 0; X = 0; Y = 0;
thread 0
S11: X = x;
S12: y = 1;
thread 1
S21: Y = y;
S22: x = 1;
print X, Y;
```

```
S12: y = 1
x: 0, y: 0
S21: Y = y
x: 0, y: 1
S22: x = 1
x: 0, y: 1
S11: X = x
x: 1, y: 1
S21: Y = y
S22: x = 1
S11: X = x
S12: y = 1
```
SC Violation

Not sequentially consistent

\[
\begin{align*}
x &= 0; \quad y = 0; \quad X = 0; \quad Y = 0; \\
\text{thread 0} & \\
S11: & \quad y = 1; \quad S12: \quad X = x; \\
\text{thread 1} & \\
S21: & \quad x = 1; \quad S22: \quad Y = y; \\
\end{align*}
\]

print X, Y;

\[
\begin{align*}
x = 0; \quad y = 0; \quad X = 0; \quad Y = 0; \\
\text{thread 0} & \\
S11: & \quad y = 1; \quad S12: \quad X = x; \\
\text{thread 1} & \\
S21: & \quad x = 1; \quad S22: \quad Y = y; \\
\end{align*}
\]
SC Violation

Not sequentially consistent

x=0; y = 0; X = 0; Y = 0;
thread 0

S11: y = 1;
S12: X = x;

S21: x = 1;
S22: Y = y;

print X, Y;

thread 1

S11: y = 1;
S12: X = x;

S21: x = 1;
S22: Y = y;

x: 0
y: 0
SC Violation

x = 0; y = 0; X = 0; Y = 0;

thread 0
S11: y = 1;
S12: X = x;

thread 1
S21: x = 1;
S22: Y = y;

print X, Y;

S11: y = 1;
S12: X = x;

S21: x = 1;
S22: Y = y;

x = 0; y = 0; X = 0; Y = 0;

Not sequentially consistent

x = 0; y = 0; X = 0; Y = 0;

thread 0
S11: y = 1;
S12: X = x;

thread 1
S21: x = 1;
S22: Y = y;

print X, Y;

S11: y = 1;
S12: X = x;

S21: x = 1;
S22: Y = y;
SC Violation

x = 0; y = 0; X = 0; Y = 0;

thread 0
S11: y = 1;
S12: X = x;

thread 1
S21: x = 1;
S22: Y = y;

print X, Y;

Not sequentially consistent

S12
S21
S22
S11

x = 0; y = 0; X = 0; Y = 0;

time

1

S11: y = 1;
S12: X = x;

S21: x = 1;
S22: Y = y;

2

3
SC Violation

x = 0; y = 0; X = 0; Y = 0;
thread 0
S11: y = 1;
S12: X = x;
print X, Y;

thread 1
S21: x = 1;
S22: Y = y;

x = 0; y = 0; X = 0; Y = 0;

Not sequentially consistent

S12
S21
S11

S12: X = x;
S11: y = 1;
S21: x = 1;
S22: Y = y;

X: 0
Y: 0

thread 0
thread 1

print X, Y;

S12: X = x;
S11: y = 1;
Relaxed Memory Consistency Models

- Relaxations in program order
  - Preserve dependences
  - write → write, write → read, read → write, or read → read
- High performance, but hard to guarantee correctness for the programmer (difficult to program)
- Since S21 and S22 have no dependence, they can be reordered
- Memory fence (memory barrier) instructions prevent reordering of memory instructions

```c
x = 0;
y = 0;
X = 0;
Y = 0;

S11: X = x
S12: y = 1
S21: Y = y
S22: x = 1

print X, Y;
```
Relaxed Memory Consistency Models

- Relaxations in program order
  - Preserve dependences
  - write → write, write → read, read → write, or read → read
- High performance, but hard to guarantee correctness for the programmer (difficult to program)
- Since S21 and S22 have no dependence, they can be reordered
- Memory fence (memory barrier) instructions prevent reordering of memory instructions
Performed with respect to ...

- A read by a processor P is performed with respect to another processor Q when a write by Q cannot affect the value returned by the read.
- A write by a processor P is performed with respect to another processor Q when a read by Q returns the value written by the write.
**Processor Consistency (PC)**

- **Definition**
  - Before a read is allowed to perform with respect to any other processor, all previous read must be performed
  - Before a write is allowed to perform with respect to any other processor all previous accesses must be performed
Processor Consistency (contd.)

- Relaxing write → read order
- Allow a read to bypass (complete before) an earlier write in program order
- Write buffer with read bypassing
Weak Ordering (Weak Consistency)

- Relax all program orders
  - No program orders are guaranteed by the underlying hardware or compiler optimizations
  - Except synchronization operations
    - Need to distinguish between ordinary reads/writes and synchronization operation

- Definition
  - Before an ordinary read/write is allowed to perform w.r.t. any processor, all previous synchronization operations must be performed w.r.t. every processor
  - Before a synchronization operation is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor
  - Synchronization operations obey sequential consistency
Weak Ordering (contd.)

- Multiple read/write requests can be outstanding at the same time
  - Hide read and write latency
- The ordering with respect to a synchronization operation (sync) must be guaranteed
  - read/write → sync, sync → read/write

Weak consistent, but not sequentially consistent

```
execution 1
data = 5;
done = true;
while (not done);
print data;
prints 5
```
```
execution 2
data = 5;
done = true;
while (not done);
print data;
data = 5;
prints 0
```
```
data = 0;
done = false;
processor 0
```
```
data = 5;
done = true;
while (not done);
print data;
processor 1
```
```
data = 5;
done = true;
```
```
while (not done);
print data;
```
Release Consistency

- Relax all program orders, but not w.r.t. synchronization operations
  - Two separate synchronization operations
    - Acquire: a read operation such as lock(V)
    - Release: a write operation such as unlock(V)
  - Need to distinguish between ordinary reads/writes and synchronization operations
    - Additionally distinguish between acquire and release operations

- Definition
  - Before an ordinary read/write is allowed to perform w.r.t. any processor, all previous acquires must be performed w.r.t. every processor
  - Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor
  - Acquire and release operations are sequentially consistent (or processor consistent)
Release Consistency (contd.)

• Specific memory access ordering with respect to acquire/release must be guaranteed
  • read/write → release
  • acquire → read/write

• Memory accesses in the critical section do not wait or delay reads/writes outside the critical section
  • Reads/writes following a release (unlock) do not have to be delayed for the release to complete
  • An acquire (lock) needs not to be delayed for previous reads/writes to complete

Weak ordering  Release consistency
Understanding RC

processor 0  processor 1
Understanding RC

Initially $x = 0$
Initially $x = 0$  \[ r(x) \ (x = 0) \]
Understanding RC

Initially \( x = 0 \)

\( w(x, 5) \)

\( r(x) \ (x = 0) \)
Understanding RC

Initially $x = 0$

$w(x, 5)$

$r(x)$ ($x = 0$)

$r(x)$ ($x = 0$ or $5$)
Understanding RC

Initially $x = 0$

- $w(x, 5)$
- $r(x) (x = 0)$

Processor 0

Processor 1

$r(x) (x = 0$ or $5$)

$x$ can be 0 or 5
Understanding RC

Initially $x = 0$

$w(x, 5)$

release($L$)

processor 0

processor 1

$r(x) \ (x = 0)$

$r(x) \ (x = 0 \ or \ 5)$

$x$ can be 0 or 5
Understanding RC

Initially \( x = 0 \)

\[ w(x, 5) \]

release(L)

processor 0

\[ r(x) \ (x = 0) \]

processor 1

\[ r(x) \ (x = 0 \text{ or } 5) \]

\[ r(x) \ (x = 5) \]

\( x \) can be 0 or 5
Understanding RC

Initially $x = 0$

$w(x, 5)$

release(L)

processor 0

r(x) ($x = 0$)

processor 1

r(x) ($x = 0$ or $5$)

x can be 0 or 5

Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor.
Understanding RC

Initially $x = 0$

$w(x, 5)$

release($L$)

processor 0

$r(x) (x = 0)$

$x$ can be 0 or 5

$x$ is 5 due to the release

Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor

processor 1

$r(x) (x = 0)$

$r(x) (x = 0$ or 5)

$r(x) (x = 5)$
Understanding RC

Initially $x = 0$

$w(x, 5)$

release(L)

processor 0

$r(x) \ (x = 0)$

$r(x) \ (x = 0 \ or \ 5)$

$r(x) \ (x = 5)$

$w(x, 7)$

processor 1

$x$ can be 0 or 5

$x$ is 5 due to the release

Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor.
Understanding RC

Initially $x = 0$

$w(x, 5)$

release(L)

processor 0

processor 1

$r(x) (x = 0)$

$r(x) (x = 0 \text{ or } 5)$

$r(x) (x = 5)$

$w(x, 7)$

acquire(L)

Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor.
Understanding RC

Initially $x = 0$

$w(x, 5)$

release($L$)

$r(x) (x = 0)$

$r(x) (x = 0$ or 5)

Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor

$r(x) (x = 5)$

$w(x, 7)$

acquire($L$)

$r(x) (x = 5$ or 7)
Understanding RC

Initially \( x = 0 \)

- \( w(x, 5) \)
- \( \text{release}(L) \)

\( r(x) (x = 5 \text{ or } 7) \)

\( x \) can be 5 or 7

Processor 0:

- \( r(x) (x = 0) \)

Processor 1:

- \( r(x) (x = 0 \text{ or } 5) \)
- \( r(x) (x = 5) \)
- \( w(x, 7) \)
- \( \text{acquire}(L) \)

\( x \) is 5 due to the release

\( x \) can be 0 or 5

Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor.
Understanding RC

Initially $x = 0$

$w(x, 5)$

release(L)

$r(x) \ (x = 0)$

$r(x) \ (x = 0 \ or \ 5)$

$r(x) \ (x = 5)$

$r(x) \ (x = 5)$

Before a release is allowed to perform w.r.t. any processor, all previous ordinary reads/writes must be performed w.r.t. every processor

$x$ can be 0 or 5

$x$ is 5 due to the release

$x$ can be 5 or 7
Implementations of RC

- Reduce the number of messages exchanged between processors
- Two typical implementations in page-based software shared virtual memory (SVM)
  - Eager release consistency (ERC)
  - Lazy release consistency (LRC)
Software Shared Virtual Memory (SVM)

- Software-based distributed shared memory (DSM)
  - Provide an illusion of shared memory on a cluster
    - A private address space in each node, but a single globally shared address space on the cluster
  - Keep the main memories of the nodes coherent
    - Embedding a coherence protocol in the page fault handlers
- High overheads of,
  - The protocol processing
    - Implemented in software
  - The large granularity (page) of coherence and communication
    - Useless data transfer and false sharing
Software Shared Virtual Memory (SVM)

- Software-based distributed shared memory (DSM)
  - Provide an illusion of shared memory on a cluster
    - A private address space in each node, but a single globally shared address space on the cluster
  - Keep the main memories of the nodes coherent
    - Embedding a coherence protocol in the page fault handlers
- High overheads of,
  - The protocol processing
    - Implemented in software
  - The large granularity (page) of coherence and communication
    - Useless data transfer and false sharing
Software Shared Virtual Memory (SVM)

- Software-based distributed shared memory (DSM)
  - Provide an illusion of shared memory on a cluster
    - A private address space in each node, but a single globally shared address space on the cluster
  - Keep the main memories of the nodes coherent
    - Embedding a coherence protocol in the page fault handlers
- High overheads of,
  - The protocol processing
    - Implemented in software
  - The large granularity (page) of coherence and communication
    - Useless data transfer and false sharing
SVM Idea

Shared address space

node 0

<table>
<thead>
<tr>
<th>p</th>
<th>q</th>
</tr>
</thead>
</table>

node 1

| x | y |

SVM library (runtime)

r(p) → r(q) → r(x) → r(y)
SVM Idea

Shared address space

node 0

r(p)
r(q)
r(x)

node 1

r(x)
r(y)

SVM library (runtime)

p
q
x
y
SVM Idea

Shared address space

Node 0
- r(p)
- r(q)
- r(x)

Node 1
- r(x)
- r(y)

SVM library (runtime)
SVM Idea

Shared address space

node 0

node 1

r(p)
r(q)
r(x)

SVM library (runtime)

w(x)

r(x)
r(y)

x

y

p

q

p

q

x

y

CENTER FOR MANYCORE PROGRAMMING
SVM Idea

Shared address space

node 0

node 1

SVM library (runtime)

w(x)

r(p)

r(q)

r(x)

r(x)

r(y)

x

y

x

y
SVM Idea

Shared address space

node 0

r(p)
r(q)
r(x)

SVM library (runtime)

invalidate x

node 1

r(x)
r(y)

w(x)
SVM Idea

Shared address space

node 0

- r(p)
- r(q)
- r(x)

node 1

- r(x)
- r(y)

SVM library (runtime)

invalidate x

w(x)
SVM Idea

Shared address space

node 0

node 1

r(p)  

r(q)  

r(x)  

r(y)  

x  

y  

SVM library (runtime)

invalidate x

w(x)
SVM Idea
SVM Idea

Shared address space

node 0

- p
- q
- x
- y

node 1

- r(p)
- r(q)
- r(x)

SVM library (runtime)

- x

invalidate x

- w(x)

- r(x)
- r(y)
Write Notice

- An indication of coherence actions such as invalidations or updates
- Update-based protocols
  - Modifications are sent
- Invalidate-based protocols
  - Notifications of modifications are sent
  - Invalidations are smaller than updates
Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page: x, y

node 0  node 1  node 2
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
  - Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page p

\[
\begin{align*}
\text{node 0} & \quad \text{node 1} & \quad \text{node 2} \\
r(x) & &
\end{align*}
\]
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page p

\[ \begin{align*}
\text{node 0} & & \text{node 1} & & \text{node 2} \\
r(x) & & r(x) & & r(x) \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
& & & & \\
\end{align*} \]
ERC

- Carter et al. (SOSP’91)
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page

<table>
<thead>
<tr>
<th>node 0</th>
<th>node 1</th>
<th>node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(x)</td>
<td>acq(L)</td>
<td>r(y)</td>
</tr>
</tbody>
</table>
ERC

- Carter et al. (SOSP’91)
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page

<table>
<thead>
<tr>
<th></th>
<th>node 0</th>
<th>node 1</th>
<th>node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(x)</td>
<td></td>
<td></td>
<td>r(y)</td>
</tr>
<tr>
<td>w(x)</td>
<td></td>
<td>acq(L)</td>
<td></td>
</tr>
</tbody>
</table>

ACACES 2009
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page
ERC

- Carter et al. (SOSP’91)
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page p

node 0 node 1 node 2
r(x) acq(L) r(y)
acq(x) rel(L)

inv. p
**ERC**

- **Carter et al. [SOSP’91]**

- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page

- Upon a page fault, obtain the faulting page from the latest modifier

Variables $x$ and $y$ are in the same page page: $x \ y$

<table>
<thead>
<tr>
<th>node 0</th>
<th>node 1</th>
<th>node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r(x)$</td>
<td>$w(x)$</td>
<td>$r(y)$</td>
</tr>
</tbody>
</table>

$\text{inv. p}$

$\text{acq}(L)$

$\text{re}(L)$
ERC

- **Carter et al. (SOSP’91)**
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
• Carter et al. (SOSP’91)
• Postpone sending write notices to the next release point
  • At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
• Upon a page fault, obtain the faulting page from the latest modifier
**ERC**

- **Carter et al. [SOSP’91]**
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
• **Carter et al. [SOSP’91]**

• Postpone sending write notices to the next release point
  • At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page

• Upon a page fault, obtain the faulting page from the latest modifier

Variables x and y are in the same page p
ERC

- Carter et al. [SOSP'91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
Carter et al. [SOSP’91]

Postpone sending write notices to the next release point

At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page

Upon a page fault, obtain the faulting page from the latest modifier
• Carter et al. (SOSP’91)
• Postpone sending write notices to the next release point
  • At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
• Upon a page fault, obtain the faulting page from the latest modifier
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier

Variables $x$ and $y$ are in the same page $p$
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
ERC

- Carter et al. (SOSP’91)
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
• Carter et al. [SOSP’91]
• Postpone sending write notices to the next release point
  • At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
• Upon a page fault, obtain the faulting page from the latest modifier
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
ERC

- Carter et al. [SOSP’91]
- Postpone sending write notices to the next release point
  - At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
- Upon a page fault, obtain the faulting page from the latest modifier
• Carter et al. [SOSP’91]
• Postpone sending write notices to the next release point
  • At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
• Upon a page fault, obtain the faulting page from the latest modifier
• Carter et al. [SOSP’91]
• Postpone sending write notices to the next release point
  • At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
• Upon a page fault, obtain the faulting page from the latest modifier
• Carter et al. [SOSP’91]
• Postpone sending write notices to the next release point
  • At a release, write notices for each page that the node wrote since its previous release are propagated to all nodes that have copies of that page
• Upon a page fault, obtain the faulting page from the latest modifier
LRC

- Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals

Variables x and y are in the same page per node 0, node 1, node 2
LRC

- Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals

Variables x and y are in the same page

<table>
<thead>
<tr>
<th>node 0</th>
<th>node 1</th>
<th>node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(x)</td>
<td>acq(L)</td>
<td>r(y)</td>
</tr>
<tr>
<td>w(x)</td>
<td></td>
<td>rel(L)</td>
</tr>
</tbody>
</table>
LRC

- Keleher et al. (ISCA’92), Amza et al. (IEEE Computer 96)
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals

Variables x and y are in the same page p

node 0    node 1    node 2
r(x)      r(y)      r(y)
acq(L)    w(x)     rel(L)
Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]

- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals
LRC

- Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals
LRC

- Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals
Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]

- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals
• Keleher et al. [ISCA’92], Amza et al. (IEEE Computer 96)
• Propagate and apply invalidations to a given node only at the next acquire by that node
  • Invalidate-based protocol
• On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  • Need to filter the write notices that have been performed w.r.t. itself
    • Intervals (between sync operations)
    • Use vector timestamps to establish the happens-before ordering between intervals
LRC

- Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals

Variables x and y are in the same page p

```
node 0      node 1      node 2
r(x)  →   r(y)  →   
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
    |   \    |
```
LRC

- Keleher et al. (ISCA’92), Amza et al. (IEEE Computer 96)
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals

Variables x and y are in the same page.
LRC

• Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]
  
• Propagate and apply invalidations to a given node only at the next acquire by that node
  
• Invalidate-based protocol
  
• On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  
• Need to filter the write notices that have been performed w.r.t. itself
  
  • Intervals (between sync operations)
  
  • Use vector timestamps to establish the happens-before ordering between intervals

Variables x and y are in the same page

<table>
<thead>
<tr>
<th>node 0</th>
<th>node 1</th>
<th>node 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>r(x)</td>
<td></td>
<td>r(y)</td>
</tr>
<tr>
<td></td>
<td>acq(L)</td>
<td>w(x)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>rel(L)</td>
</tr>
<tr>
<td>r(y)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>acq(L)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>grant L + inv.p</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w(x)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td>rel(L)</td>
<td></td>
<td>grant L + inv.p</td>
</tr>
</tbody>
</table>
LRC

- Keleher et al. [ISCA’92], Amza et al. (IEEE Computer 96)
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals
LRC

- Keleher et al. [ISCA’92], Amza et al. [IEEE Computer 96]
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals
LRC

- Keleher et al. (ISCA’92), Amza et al. (IEEE Computer 96)
- Propagate and apply invalidations to a given node only at the next acquire by that node
  - Invalidate-based protocol
- On an acquire, the node obtains the write notices corresponding to all previous release operations that occurred between its previous acquire operation and its current acquire operation
  - Need to filter the write notices that have been performed w.r.t. itself
    - Intervals (between sync operations)
    - Use vector timestamps to establish the happens-before ordering between intervals
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
  - Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
  - Twins and diffs

Variables x and y are in the same page p

```
node 0
  r(x)  x  y

node 1
  r(y)  x  y
```
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs

Variables x and y are in the same page p

node 0
r(x)
acq(L0)
w(x)
rel(L0)

node 1
r(y)
acq(L1)
w(y)
rel(L1)
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Multiple Writers Protocol

- To mitigate the effect of false sharing
- Each node writes a page between synchronization points to modify its own copy locally
- Make the copies consistent only at the next synchronization point by merging them
- Twins and diffs
Twins and Diffs

- To implement efficient multiple writers protocol and to reduce communication cost
- Modifications to a shared page are captured by comparing the page with its twin
  - When first writing to an unmodified page, a copy (twin) of the page is created
  - At the next synchronization point, the twin and current copy are compared to create a diff
    - Diff is a compact encoded representation of the differences
- When a node incurs a page fault, it must obtain the diffs for that page from other nodes and merge them into its copy of the page
  - Sending a diff is much cheaper than sending an entire page
- Diffs need to be retained until they are never be requested
  - Garbage collection
Twins and Diffs

Variables x and y are in the same page

node 0

node 1
Twins and Diffs

Variables $x$ and $y$ are in the same page $p.$
Twins and Diffs

Variables $x$ and $y$ are in the same page $p$.

```
node 0
  $x$ $y$  $r(x)$

node 1
  $x$ $y$  $r(y)$  $x$ $y$
```
Variables $x$ and $y$ are in the same page $p$.

```
node 0
  x y
  r(x)
  acq(L0)

node 1
  x y
  r(y)
```
Variables $x$ and $y$ are in the same page $p$.

**node 0**
- $r(x)$
- $\text{acq}(L0)$
- $w(x)$

**node 1**
- $r(y)$
- $x$ $y$
Twins and Diffs

Variables $x$ and $y$ are in the same page.
Twins and Diffs

Variables x and y are in the same page p:

node 0:
- r(x)
- acq(L0)
- w(x)

twin:
- x
- y

node 1:
- r(y)
- x
- y
Twins and Diffs

Variables x and y are in the same page.
Twins and Diffs

Variables $x$ and $y$ are in the same page $p$.

**Node 0**
- $r(x)$
- $acq(L0)$
- $w(x)$

**Node 1**
- $r(y)$
- $acq(L1)$
- $w(y)$
Twins and Diffs

Variables $x$ and $y$ are in the same page $p$.

Node 0
- $r(x)$
- $acq(L0)$
- $w(x)$

Node 1
- $r(y)$
- $acq(L1)$
- $w(y)$

Twin
 Twins and Diffs

Variables x and y are in the same page p

```
node 0
  r(x)
  acq(L0)
  w(x)
```

```
node 1
  r(y)
  acq(L1)
  w(y)
```

Variables $x$ and $y$ are in the same page.
Twins and Diffs

Variables x and y are in the same page p

node 0

r(x)
acq(L0)
w(x)
rel(L0)
twin

node 1

r(y)
acq(L1)
w(y)
twin
Twins and Diffs

Variables x and y are in the same page p

Node 0
- r(x)
- acq(L0)
- w(x)
- rel(L0)

Node 1
- r(y)
- acq(L1)
- w(y)
- rel(L1)
Twins and Diffs

Variables x and y are in the same page p.
Variables $x$ and $y$ are in the same page.

- **node 0**
  - $r(x)$
  - $acq(L0)$
  - $w(x)$
  - $twin$ (between $x$ and $y$)
  - $rel(L0)$
  - $r(y)$

- **node 1**
  - $r(y)$
  - $acq(L1)$
  - $w(y)$
  - $twin$ (between $x$ and $y$)
  - $rel(L1)$
  - $y$
Twins and Diffs

Variables x and y are in the same page p.

node 0

node 1

r(x)
acq(L0)
w(x)
rel(L0)
r(y)
acq(L1)
w(y)
rel(L1)
Variables $x$ and $y$ are in the same page $p$. 

**Node 0**
- $r(x)$
- acq(L0)
- $w(x)$
- rel(L0)
- $x \oplus y = x$

**Node 1**
- $r(y)$
- acq(L1)
- $w(y)$
- rel(L1)
- $y \oplus x = y$
Variables x and y are in the same page p (x y)

Node 0

- r(x)
- acq(L0)
- w(x)
- rel(L0)

Node 1

- r(y)
- acq(L1)
- w(y)
- rel(L1)

x y + y = x y

r(x)

r(y)
Twins and Diffs

Variables x and y are in the same page p: x y

node 0

r(x)
acq(L0)
w(x)
rel(L0)

node 1

r(y)
acq(L1)
w(y)
rel(L1)

twin

x y + y = x y

r(y)

r(x)
Twins and Diffs

Variables x and y are in the same page p

node 0

\[ \begin{align*}
    \text{twin} & \quad x \quad y \\
    \text{acq(L0)} & \quad x \quad y \\
    \wedge & \quad x \\
    \text{rel(L0)} & \quad x \quad y + y = x \quad y \\
\end{align*} \]

node 1

\[ \begin{align*}
    \text{twin} & \quad x \quad y \\
    \text{acq(L1)} & \quad x \quad y \\
    \wedge & \quad y \\
    \text{rel(L1)} & \quad y = x \quad y + x \\
\end{align*} \]
Variables $x$ and $y$ are in the same page $p$.
Variables $x$ and $y$ are in the same page $p$

node 0

acq(L)

node 1
Lazy Diff Creation

Variables $x$ and $y$ are in the same page $p$

node 0

acq($L$)

w($x$)

node 1
Lazy Diff Creation

Variables $x$ and $y$ are in the same page $p$

node 0

acq(L)

w(x)

make a twin

node 1
Lazy Diff Creation

Variables x and y are in the same page p

node 0

acq(L)

node 1

w(x)

make a twin

rel(L)
Lazy Diff Creation

Variables $x$ and $y$ are in the same page $p$

Node 0
- $\text{acq}(L)$
- $w(x)$
- make a twin
- $\text{rel}(L)$

Node 1
- $\text{acq}(L)$
- grant $L + \text{inv. } p$
Lazy Diff Creation

Variables \(x\) and \(y\) are in the same page \(p\)

node 0

- \(\text{acq}(L)\)
- \(w(x)\)
- make a twin
- \(\text{rel}(L)\)
- create diff

node 1

- \(\text{acq}(L)\)

grant \(L + \text{inv. p}\)
Variables $x$ and $y$ are in the same page $p$

node 0

- acq($L$)
- w($x$)
- make a twin
- rel($L$)
- create diff

node 1

- acq($L$)
- grant $L + \text{inv. } p$
- r($y$)
Variables $x$ and $y$ are in the same page $p$.

- Node 0
  - acq($L$)
  - $w(x)$
  - make a twin
  - rel($L$)

- Node 1
  - acq($L$)
  - grant $L + inv.p$

- create diff
  - $r(y)$
Lazy Diff Creation

Variables x and y are in the same page p

node 0

acq(L)

w(x)

make a twin

rel(L)

create diff

grant L + inv. p

acq(L)

r(y)

diff

node 1
Variables $x$ and $y$ are in the same page $p$

Node 0

- `acq(L)`
- `w(x)`
- make a twin
- `rel(L)`

Node 1

- `acq(L)`
- `grant L + inv. p`
- `r(y)`
- `diff`
- `apply diff`
Home-based Lazy Release Consistency (HLRC)

- Zhou et al. [OSDI’96]
- Each page has its own home node
  - To collect updates from multiple writers
- Diffs are computed at the end of each interval and sent to the home nodes of the corresponding pages
  - The lifetime of diffs is very short
- Home nodes apply arriving diffs as soon as they are sent
- On a page fault, the entire page is fetched from home
Variables $x$ and $y$ are in the same page $p$

node 0

node 1

node 2 (home of $p$)
Variables $x$ and $y$ are in the same page $p$

node 0

$\text{acq}(L)$

node 1

node 2

(home of $p$)
HLRC (contd.)

Variables x and y are in the same page p

node 0
acq(L)
w(x)

node 1

node 2 (home of p)
Variables x and y are in the same page p
node 0

acq(L)
w(x)

make a twin

node 1

node 2
(home of p)
Variables x and y are in the same page p

node 0

acq(L)

w(x)

make a twin

rel(L)

node 1

node 2
(home of p)
Variables $x$ and $y$ are in the same page $p$

Node 0
- $\text{acq}(L)$
- $w(x)$
- Make a twin
- $\text{rel}(L)$

Node 1
- $\text{acq}(L)$

Node 2 (home of $p$)
Variables $x$ and $y$ are in the same page $p$

- Node 0:
  - `acq(L)`
  - `w(x)`
  - make a twin
  - `rel(L)`

- Node 1:
  - `acq(L)`

- Node 2 (home of $p$):
  - `grant L + inv. p`
Variables x and y are in the same page p

node 0

acq(L)

w(x)

make a twin

rel(L)

create diff

node 1

acq(L)

node 2
(home of p)

grant L + inv. p
Variables $x$ and $y$ are in the same page $p$

node 0

- $acq(L)$
- $w(x)$

make a twin

- $rel(L)$

create diff

node 1

- $acq(L)$

node 2 (home of $p$)

- $grant \ L + inv. \ p$
Variables $x$ and $y$ are in the same page $p$

- $\text{node 0}$
  - $\text{acq}(L)$
  - $w(x)$
- $\text{make a twin}$
  - $\text{rel}(L)$
- $\text{create diff}$

- $\text{node 1}$
  - $\text{acq}(L)$
  - $\text{grant } L + \text{ inv. } p$

- $\text{node 2}$ (home of $p$)
  - $\text{apply diff}$
Variables $x$ and $y$ are in the same page $p$

- `node 0`
  - `acq(L)`
  - `w(x)`
  - `make a twin`
  - `rel(L)`
  - `create diff`
  - `grant L + inv. p`

- `node 1`
  - `acq(L)`
  - `r(y)`

- `node 2`
  - `apply diff`
Variables x and y are in the same page p

node 0

acq(L)

w(x)

make a twin

rel(L)

create diff

acq(L) grant L + inv. p

r(y)

apply diff

node 1

node 2 (home of p)
Further Readings

Further Readings

Preliminaries:

Code Overlays
Code Overlays

- For memory constrained environments
  - DSPs that have constrained program address space, Instruction scratchpad memory, etc.
- No hardware support for virtual memory
- Dates back to before 1960
- DOS linkers in the 1980s
- When logical address space specified by the programmer is bigger than the given physical address space

![Diagram of Code Overlays]

- Unoverlaid functions: main(), foo(), bar(), lido()
- Linker
- Overlay structure
- Overlay manager
- Overlaid functions: foo(), lido(), main(), bar()
- Physical memory
Basic Idea

- Divide the code into a tree of functions
- Sibling functions in the tree share the same memory region
- The Linker relocates the code in each function appropriately based on the overlay structure
- The linker places glue code in front of each call
  - The glue code hooks into the overlay manager
  - The overlay manager ensures that the target function is loaded

```
main()
  foo()
    bar()
  fido()
```

![Diagram of function calls]

ACACES 2009
Basic Idea

- Divide the code into a tree of functions
- Sibling functions in the tree share the same memory region
- The Linker relocates the code in each function appropriately based on the overlay structure
- The linker places glue code in front of each call
  - The glue code hooks into the overlay manager
  - The overlay manager ensures that the target function is loaded
Basic Idea

- Divide the code into a tree of functions
- Sibling functions in the tree share the same memory region
- The Linker relocates the code in each function appropriately based on the overlay structure
- The linker places glue code in front of each call
  - The glue code hooks into the overlay manager
  - The overlay manager ensures that the target function is loaded
Basic Idea

- Divide the code into a tree of functions
- Sibling functions in the tree share the same memory region
- The Linker relocates the code in each function appropriately based on the overlay structure
- The linker places glue code in front of each call
  - The glue code hooks into the overlay manager
  - The overlay manager ensures that the target function is loaded
Automatic Overlay Structure Generation

- Cytron and Loewner (IBM J. of R&D 1986)
  1. Generate the call graph
     - Nodes correspond to the functions
     - A directed edge exists between node fi and fj if fi calls fj
  2. Create a directed acyclic graph (DAG)
     - Collapse each strongly connected component (SCC) of the call graph
     - Functions that participate in recursive invocations require residency of each other

```
root
  f1
  f2
  f3
  f4
  f5
  f6
  f7
  f8
  f9
  f10

root
  f1
  f2
  f3
  f4
  f5
  f6
  f7
  f8
  f9
  f10
```

ACACES 2009
3. Generation of the overlay structure
   - Nodes that lie along disjoint paths cannot be simultaneously active and can be overlaid
   - Overlay generation is trivial if the DAG is a tree
     - After applying absorbing procedure to the DAG in post-order traversal, it becomes a tree
3. Generation of the overlay structure
   - Nodes that lie along disjoint paths cannot be simultaneously active and can be overlaid
   - Overlay generation is trivial if the DAG is a tree
     - After applying absorbing procedure to the DAG in post-order traversal, it becomes a tree

Multiple edges incident on node f5 imply the residency of f5 in memory for multiple call paths.
3. Generation of the overlay structure
   - Nodes that lie along disjoint paths cannot be simultaneously active and can be overlaid
   - Overlay generation is trivial if the DAG is a tree
     - After applying absorbing procedure to the DAG in post-order traversal, it becomes a tree

Multiple edges incident on node f5 imply the residency of f5 in memory for multiple call paths.
3. Generation of the overlay structure
   - Nodes that lie along disjoint paths cannot be simultaneously active and can be overlaid
   - Overlay generation is trivial if the DAG is a tree
     - After applying absorbing procedure to the DAG in post-order traversal, it becomes a tree

**Diagram:**
- Nodes f1, f3, f4, f5, f6, f7
- Multiple edges incident on node f5 imply the residency of f5 in memory for multiple call paths.
- Absorbing a node such as f5 into its immediate dominator f1. F5 is resident whenever f1 is resident, i.e., f5 and f1 cannot share the same memory region.
3. Generation of the overlay structure
   • Nodes that lie along disjoint paths cannot be simultaneously active and can be overlaid
   • Overlay generation is trivial if the DAG is a tree
     • After applying absorbing procedure to the DAG in post-order traversal, it becomes a tree

   - Node m dominates node n if every path from the root to n contains m
   - Node m immediately dominates node n if m dominates n, and there is no intervening node p such that m dominates p and p dominates n

   ![Diagram of overlay structure]

   Multiple edges incident on node f5 imply the residency of f5 in memory for multiple call paths.

   Absorbing a node such as f5 into its immediate dominator f1. F5 is resident whenever f1 is resident, i.e., f5 and f1 cannot share the same memory region.
3. Generation of the overlay structure
   • Nodes that lie along disjoint paths cannot be simultaneously active and can be overlaid
   • Overlay generation is trivial if the DAG is a tree
     • After applying absorbing procedure to the DAG in post-order traversal, it becomes a tree

- Node m dominates node n if every path from the root to n contains m
- Node m immediately dominates node n if m dominates n, and there is no intervening node p such that m dominates p and p dominates n
3. Generation of the overlay structure
   • Nodes that lie along disjoint paths cannot be simultaneously active and can be overlaid
   • Overlay generation is trivial if the DAG is a tree
     • After applying absorbing procedure to the DAG in post-order traversal, it becomes a tree

- Node m dominates node n if every path from the root to n contains m
- Node m immediately dominates node n if m dominates n, and there is no intervening node p such that m dominates p and p dominates n
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure

---

**Automatic Overlay Structure Generation (contd.)**

4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure

![Diagram of overlay structure]
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure
4. Emit the overlay structure

Diagram of overlay structure:

- **Root**: Represents the highest level of the structure.
- **f8**: Located at the bottom, indicating a leaf node.
- **f2**, **f9**, and **f10**: Internal nodes connecting various branches.
- **f11**, **f12**, and **f13**: Additional internal nodes branching out to different parts of the structure.

The diagram illustrates the hierarchical nature of the overlay, with each node leading to a sub-node, showing how the overlay structure is generated and emitted.
4. Emit the overlay structure

Automatic Overlay Structure Generation (contd.)
Further Readings

FaCSim: A Fast and Cycle-Accurate Architecture Simulator for Embedded Systems
FaCSim

- Lee et al. (LCTES’08)
- Cycle-accurate ARM architecture simulator
  - Interpretive
  - Implemented in C++ from scratch
    - Easy modification
  - Simulation engine for the virtual platform
- Accurately models the pipeline of ARM9E-S and the memory subsystem of ARM926EJ-S
  - Enables full system simulation
  - Includes MMU, exceptions, and interrupts
- Open source
  - Downloadable at the URL, http://aces.snu.ac.kr
- FaCSim facilitates,
  - Embedded architecture explorations
  - Instrumenting and profiling embedded applications and operating systems
  - Functional and performance debugging of embedded software
The Structure of FaCSim
Key Ideas

- Caching decoded instruction objects
  - Typically used in dynamically compiled simulators
    - JIT compilation and code caches
    - But, FaCSim is an interpretive simulator
  - At the functional front-end (FaCSim-func)
- No cycle-by-cycle simulation
  - ARM9E 5-stage pipeline
  - A multi-layer AHB system
    - Two separate AHB masters
      - Instruction and data
  - Lazy synchronization of the bus clock and the core clock
- Parallelized to exploit multicore host systems
  - Lock-free synchronization
  - For two cores
Key Ideas (contd.)

- Decoupling the functional model and the timing model
  - Run them in parallel
  - Similar to FAST (Chiou et al., Micro’07)
    - The timing model is implemented in an FPGA
    - Boosting performance, but lacking observability
- FaCSim’s timing model (back-end) is a pure software implementation
  - Good observability and flexibility of modification
- Producer-consumer model
  - Producer: FaCSim-func, Consumer: FaCSim-acc
  - Using a lock-free circular queue
    - No synchronization for shared variables
      - Boost performance
Accuracy of FaCSim

- 21 applications from EEMBC suite
- Host system
  - Intel Core2Quad 2.4GHz, 4GB main memory
  - Using only two cores for running FaCSim
- Comparing FaCSim to a real ARM926EJ-S development board, ARMulator, and SimpleScalar
- Validating the core cycles (i.e., core busy cycles) against ARMulator
  - Less than ±1% error margin
- Validating the accuracy against the ARM926EJ-S development board
  - Less than ±7% error margin

<table>
<thead>
<tr>
<th>Image Name</th>
<th>Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>bezier01fixed</td>
<td>-4.10%</td>
</tr>
<tr>
<td>dither01</td>
<td>-3.37%</td>
</tr>
<tr>
<td>rotate01</td>
<td>-12.73%</td>
</tr>
<tr>
<td>text01</td>
<td>4.77%</td>
</tr>
<tr>
<td>aes</td>
<td>-9.99%</td>
</tr>
<tr>
<td>jpegv2data3</td>
<td>-5.83%</td>
</tr>
<tr>
<td>jpegv2data3</td>
<td>-1.51%</td>
</tr>
<tr>
<td>huffde</td>
<td>-13.32%</td>
</tr>
<tr>
<td>mp2decoddata5</td>
<td>-6.35%</td>
</tr>
<tr>
<td>mp2enf32data2</td>
<td>-13.24%</td>
</tr>
<tr>
<td>mp2enfixdata2</td>
<td>-5.44%</td>
</tr>
<tr>
<td>mp3playerdata3</td>
<td>-10.21%</td>
</tr>
<tr>
<td>mp4decodedata4</td>
<td>-0.43%</td>
</tr>
<tr>
<td>mp4encodedata2</td>
<td>0.12%</td>
</tr>
<tr>
<td>rgbhpgv2data3</td>
<td>-2.85%</td>
</tr>
<tr>
<td>rgbyiqv2data1</td>
<td>-2.42%</td>
</tr>
<tr>
<td>rsa</td>
<td>-6.43%</td>
</tr>
<tr>
<td>autcor00data_3</td>
<td>4.63%</td>
</tr>
<tr>
<td>conven00data_1</td>
<td>-1.15%</td>
</tr>
<tr>
<td>fbital00data_6</td>
<td>-3.55%</td>
</tr>
<tr>
<td>viterb00data_1</td>
<td>-0.47%</td>
</tr>
<tr>
<td>Root Mean Square</td>
<td>6.79%</td>
</tr>
</tbody>
</table>
FaCSim’s Speed

- On average, 3 times faster than ARMulator and 6 times faster than SimpleScalar
FaCSim’s Speed (contd.)

<table>
<thead>
<tr>
<th></th>
<th>Board</th>
<th>ARMulator</th>
<th>SimpleScalar</th>
<th>FacSim</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIPS</td>
<td>77.10</td>
<td>1.43</td>
<td>0.80</td>
<td>4.25</td>
</tr>
</tbody>
</table>

- Overall, FaCSim is 18 times slower than the 140MHz ARM926EJ-S development board
- On average 77.10 MIPS vs. 4.25 MIPS
- Currently, FaCSim runs an unmodified Linux image
  - Linux-2.6.21-arm1
  - With some functionally modeled ARM peripherals
    - PrimeXSys System Controller (SP810)
    - PrimeCell Vector Interrupt Controller (PL190)
    - PrimeCell Multiport Memory Controller (GX175)
    - PrimeCell Static Memory Controller (PL093)
    - PrimeCell Timer Module (SP804)
    - PrimeCell UART (PL011)
Instruction Scratchpad Memory Management Techniques: without a Memory Management Unit
Dynamic Code Placement Technique without an MMU

- For low-end embedded systems with no virtual memory
- Based on demand paging and compiler postpass optimizations
- Whole program analysis including libraries

![Diagram of target and reference architectures with core, instruction SPM, instruction cache, bus, and memory.]
Related Work

- Static allocation
  - Verma et al. (DATE’04)
    - Cache-aware scratchpad allocation algorithm

- Dynamic allocation
  - Steinke et al. (ISSS’02)
    - Reducing energy consumption by dynamic copying of instructions onto on-chip memory

- Udayakumaran et al. (CASES’03), Poletti et al. (DAC’04), Angiolini et al. (CASES’04)
Postpass Optimization

- Input: ARM ELF object files
- Application’s object files and libraries
- Output: SPM-optimized image
- Natural loop extraction
  - Extracted loops are transformed into independent functions (outlining)
- Break down functions in the executable into three classes: SPM, Ext, and Paged
  - Based on profiling and integer linear programming
- Paged functions are transformed into segments
- Modify calls/returns for functions
Dynamic Code Placement without an MMU in a Nutshell

- Page manager manages the execution buffer and the paged function table
  - Load paged functions on demand
  - Evict functions if necessary (round robin replacement)
  - Modify the contents of the paged function table
  - Perform function lookups
    - Necessary for control transfers to targets that cannot be resolved at compile-time
      - Calls with function pointers and returns

```
bl pagetable[bar]
```

At runtime
Dynamic Code Placement without an MMU in a Nutshell (contd.)

- Paged Function Table
  - One entry per paged function
  - Each entry consists of two instructions
  - Contents change at run-time by the page manager

Miss

foo:

push PC
b page_manager

Paged function table

Hit

foo:
b exec_buffer(foo)
(ignored)

Paged function table
Natural Loop Extraction and Outlining

- Natural loop
  - Single entry point: header
    - Header dominates all nodes in the loop
  - Extracted loops are transformed into independent functions (outlining)
Code Classification/Segmentation

- Code classification
  - Based on profiling and an Integer Linear Programming (ILP) formulation
- Three classes
  - SPM: SPM-resident functions
  - Ext: functions residing in and executed from the external memory
  - Paged:
    - Initially reside in the external memory
    - copied on-demand to the SPM before execution
- Segmentation
  - Paged functions are transformed into segments
Call/Return Expansion

- Calls/Returns to paged functions must be intercepted
  - Branch and Link
    ```
    bl foo
    (return address)
    ```
  - Branch
    ```
    b foo
    ```
  - Branch with function pointers
    ```
    mov lr, pc
    b rx
    (return address)
    ```
  - Returns
    ```
    mov pc, lr
    ```

```
L1: ldr lr, L2
    b pagetable[foo]
L2: the return address in the external memory
```

```
L1: push rx
    ldr lr, L2
    b pagemanager
L2: the return address in the external memory
```

```
push lr
    b pagemanager_return
```
Integer Linear Programming Formulation: 0-1 Knapsack Problem

Given a knapsack with maximum capacity $S_{spm}$ and $N$ items (each item $f_i$ has some weight $S_i$, and penalty $E_i$), how to pack the knapsack to achieve minimum total penalty of packed items?

- **Given**
  - $A_i$: the number of instruction fetches and read-only data word accesses located in a function $f_i$
  - $S_i$: the size of a function $f_i$ in bytes
  - $N$: the number of functions in the application
  - $S_{spm}$: the SPM size in bytes
  - $E_{spm}$: the energy consumed to fetch an instruction (or a word) from the SPM
  - $E_{ext}$: the energy consumed to fetch an instruction (or a word) from the external memory

- **Binary integer variables**

  $$I_{spm}(i) = \begin{cases} 1 & \text{if } f_i \text{ is placed in the SPM} \\ 0 & \text{otherwise} \end{cases} \quad I_{ext}(i) = \begin{cases} 1 & \text{if } f_i \text{ is placed in the external memory} \\ 0 & \text{otherwise} \end{cases}$$

- **Minimize**

  $$\sum_{i=1}^{N} (I_{spm}(i) \cdot A_i \cdot E_{spm} + I_{ext}(i) \cdot A_i \cdot E_{ext})$$

- **Constraints**

  $$I_{spm}(i) + I_{ext}(i) = 1 \text{ for all } 0 < i \leq N$$
  $$\sum_{i=1}^{N} I_{spm}(i) \cdot S_i \leq S_{spm}$$
Integer Linear Programming Formulation: Demand Paging

- Additional definitions
  - $P$: page size
  - $I_{\text{buffer}}$: the size of the execution buffer (a new general integer variable)
  - $C_i$: the number of calls to $f_i$
  - $R_i$: the number of returns to $f_i$
  - $E_c$: the energy consumed by extra instructions generated by the call expansion
  - $E_r$: the energy consumed by extra instructions generated by the return expansion

- Additional binary integer variable
  $$I_{\text{paged}}(i) = \begin{cases} 1 & \text{if } f_i \text{ is in Paged and } S_i \leq I_{\text{buffer}} \times P \\ 0 & \text{otherwise} \end{cases}$$

- Minimize
  $$\sum_{i=1,N} \left( I_{\text{spm}}(i) \cdot A_i \cdot E_{\text{spm}} + I_{\text{ext}}(i) \cdot A_i \cdot E_{\text{ext}} + I_{\text{paged}}(i) \cdot [A_i \cdot E_{\text{spm}} + \text{Penalty}_i] \right)$$
  $$\text{Penalty}_i = (C_i + R_i)(E_{\text{spm}} + E_{\text{ext}})[S_i/P](P/4) + C_i E_c + R_i E_r$$
  $$I_{\text{spm}}(i) + I_{\text{ext}}(i) + I_{\text{paged}}(i) = 1 \text{ for all } 0 < i \leq N$$

- Additional constraints
  $$0 \leq I_{\text{buffer}} \cdot P \leq S_{\text{spm}}$$
  $$\sum_{i=1,N} I_{\text{spm}}(i) \cdot S_i \leq S_{\text{spm}} - I_{\text{buffer}} \cdot P$$
Energy Consumption and Execution Time

ARM9E-S 200MHz

Normalized Energy Consumption

SDRAM 1K SDRAM 256K SDRAM 512K SDRAM 1MB

Core SDRAM Standby SDRAM Data SDRAM Inst. ICache SPM

0% 20% 40% 60% 80% 100%

synthetic ft
epic

Normalized Energy Consumption

SDRAM 1K SDRAM 256K SDRAM 512K SDRAM 1MB

Core SDRAM Standby SDRAM Data SDRAM Inst. ICache SPM

0% 20% 40% 60% 80% 100%

unepic mpeg4enc mpeg4dec

ACACES 2009
Comparison to Instruction Cache

The ICache size is about 20% of the executed code size
Summary of Performance

- The ICache size is about 20% of the executed code size
- Instruction cache vs. SPM with comparable die area
  - Energy consumption reduced by 21.6%
  - Execution time reduced by 20.2%

<table>
<thead>
<tr>
<th>Application</th>
<th>Executed code size</th>
<th>ICache size</th>
<th>Comparable SPM size</th>
<th>Exec. Time to ICache (%)</th>
<th>Exec. Time to Knap (%)</th>
<th>Total Energy to ICache (%)</th>
<th>Total Energy to Knap (%)</th>
<th>Number of SPM pinned library/user functions</th>
<th>Number of paged library/user functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>synthetic</td>
<td>12KB</td>
<td>2KB</td>
<td>6KB</td>
<td>64.7</td>
<td>97.5</td>
<td>62.6</td>
<td>97.3</td>
<td>11/35</td>
<td>0/3</td>
</tr>
<tr>
<td>fft</td>
<td>14KB</td>
<td>2KB</td>
<td>6KB</td>
<td>70.1</td>
<td>78.2</td>
<td>75.3</td>
<td>77.7</td>
<td>14/6</td>
<td>0/0</td>
</tr>
<tr>
<td>epic</td>
<td>21KB</td>
<td>4KB</td>
<td>8KB</td>
<td>90.5</td>
<td>88.3</td>
<td>85.1</td>
<td>87.3</td>
<td>14/15</td>
<td>0/5</td>
</tr>
<tr>
<td>unepic</td>
<td>20KB</td>
<td>4KB</td>
<td>8KB</td>
<td>87.6</td>
<td>38.2</td>
<td>83.4</td>
<td>36.8</td>
<td>15/41</td>
<td>0/6</td>
</tr>
<tr>
<td>mpeg4enc</td>
<td>49KB</td>
<td>8KB</td>
<td>12KB</td>
<td>74.9</td>
<td>95.9</td>
<td>73.7</td>
<td>95.7</td>
<td>8/45</td>
<td>0/15</td>
</tr>
<tr>
<td>mpeg4dec</td>
<td>43KB</td>
<td>8KB</td>
<td>12KB</td>
<td>96.1</td>
<td>85.2</td>
<td>94.8</td>
<td>84.7</td>
<td>15/16</td>
<td>0/12</td>
</tr>
<tr>
<td>Average (geometric mean)</td>
<td></td>
<td></td>
<td></td>
<td>79.8</td>
<td>77.1</td>
<td>78.4</td>
<td>76.3</td>
<td>11/35</td>
<td>0/3</td>
</tr>
</tbody>
</table>
Instruction Scratchpad Memory Management Techniques: with a Memory Management Unit
Dynamic Code Placement Technique with an MMU

- Hardware
  - The on-chip instruction cache is replaced by physically-addressed SPM plus a small mini instruction cache
- Based on demand paging and compiler postpass optimizations
- Whole program analysis including libraries
Dynamic Code Placement Technique with an MMU (contd.)

- Scratchpad memory management for contemporary portable devices running a full-fledged operating system with virtual memory
  - Reduce memory system energy consumption
  - Keep (at least) the same performance
  - SPM size unknown at compile-time
  - SPM-optimized binaries run on various configurations, even cache-only cores
  - SPM-unaware binaries run with acceptable performance on the proposed architecture
Related Work

- Ngyuen et al. [CASES’05]
  - Memory allocation for embedded systems with a compile-time unknown scratch-pad size
  - Static approach
  - Decision made when the application is loaded
- Shrivastava et al. [CASES’05]
  - Compilation techniques for energy reduction in horizontally partitioned cache architectures
    - XScale: big main data cache + 2KB mini-cache allocate data objects to one of the caches to save energy
SPM Access Latency

- ARM926EJ-S SPM: 2 cycles
  - VA to PA translation: 1 cycle
  - Compare PA to the SPM base address register
    - If in the range, access the SPM
- ARM11 core: 1 cycle
  - Consumes more energy by accessing the SPM and cache simultaneously
SPM Access Latency

- ARM926EJ-S SPM: 2 cycles
  - VA to PA translation: 1 cycle
  - Compare PA to the SPM base address register
    - If in the range, access the SPM
- ARM11 core: 1 cycle
  - Consumes more energy by accessing the SPM and cache simultaneously
SPM Access Latency

- ARM926EJ-S SPM: 2 cycles
  - VA to PA translation: 1 cycle
  - Compare PA to the SPM base address register
    - If in the range, access the SPM
- ARM11 core: 1 cycle
  - Consumes more energy by accessing the SPM and cache simultaneously
Proposed Memory Architecture

- Horizontally-partitioned on-chip memory subsystem
  - $\mu$ TLB
  - Scratchpad memory and direct-mapped mini-cache
  - 1-cycle latency (up to 1GHz)
  - SPM flag: an additional bit in each TLB entry
    - Determine whether the datum is to be loaded from the SPM or the mini-cache
Scratchpad Memory Management

- At compile time
  - Code is classified based on a trace analysis
    - Uncached, cached, and paged regions
  - Paged code is clustered into pages based on temporal locality
- Run-time SPM manager (SPMM)
  - Part of the runtime
  - SPMM loader
  - SPMM handler
SPMM Loader

- Detects SPM-optimized binaries
- Sets up the page tables accordingly
  - Pages in the paged region are unmapped initially

```c
spmm_loader() {
    load SPM-optimized binary;
    setup page tables;
    start process;
}
```
SPMM Handler

- Manages the SPM while the application is running
  - Page replacement policy: round robin
- Copies code to the SPM on demand
  - Intercepts page fault exceptions generated by the MMU

```c
int main(void) {
    return foobar(bar());
}
int foo() { ... }
int bar() { ... }
int foobar() { ... }

int spmm_pgflt(uint adr) {
    ...
    copy page to SPM;
    modify page table entry;
    resume aborted inst;
}
```
int main(void) {
    return foobar(bar());
}
int foo() { ... }
int bar() { ... }
int foobar() { ... }

int spmm_pgflt(uint adr) {
    disable PTE of loaded page;
    copy page to SPM;
    modify page table entry;
    resume aborted inst;
}
Code Classification

- For each basic block $b_i$, the code region is determined by

$$\text{Loc}_i = \begin{cases} 
\text{uncached} & \text{if the code is executed less than once} \\
\text{cached} & \text{if } E_{cached}(b_i) < E_{paged}(b_i) \\
\text{paged} & \text{otherwise}
\end{cases}$$

with

$$E_{paged}(b_i) = A_i E_{spm} + M S_i (E_{ext} + E_{spm})$$
$$E_{cached}(b_i) = A_i (E_{cache} + m_{cache} E_{miss})$$

where

- $A_i$: number of instructions fetched
- $S_i$: size of block $i$
- $M$: average number of page misses
- $m_{cache}$: cache miss ratio
- $E_{spm/\text{ext/cache/miss}}$: SPM/external memory/cache access/cache miss energy
Function Splitting

- Depending on the location determined to each basic block

```
foo()
B1
B2
B3
B4
B5

foo_paged()
B2
B4

foo_uncached()
B5

foo_cached()
B1
B3
```
Code Placement

- To achieve maximum performance, paged functions must be arranged in a particular way
  - Cluster temporally local functions together into as few pages as possible
- Loop detection
  - Using dynamic call graph with edges weighted by the number of calls
  - $|a \rightarrow b|$: the weight of $a \rightarrow b$
  - A function $f$ is a loop header if there exists some $g$ s.t. $|g \rightarrow f| \geq c \cdot |* \rightarrow g|$
    - $|* \rightarrow g|$: the number of incoming calls to $g$
    - $c$: a threshold value
  - The members of in the loop with header $h$
    - All functions $f$ that are reachable from $h$ and are called at least as many times as $h$ itself
- Place functions in the same loop together
  - Place functions in the same innermost loop first
Performance and Energy Consumption

- Comparing SPM+minicache to ICache (20%-30% of the executed code size) with comparable die area requirements, we achieve
  - 12% improvement in runtime performance
  - 33% reduction in energy consumption
Summary

- Based on post-pass optimizations and demand paging
  - Automated dynamic code placement technique
  - Need a good code clustering technique
  - Without MMU
    - At run-time, function calls/returns to paged functions are intercepted by the page manager
  - With MMU,
    - On-demand paging using the MMU’s page fault exception
- Can replace the instruction cache
- Effective to reduce energy consumption without loosing performance
Further Readings

Further Readings

Software-Managed Caches for Multicores with Local Memory
The Target Architecture

- General-purpose processor element (PE)
  - System management tasks (OS)
- Accelerator PEs
  - Compute intensive workloads
  - Local memory
  - DMAs for transferring data

E.g., Cell BE processors and GPGPUs
The Runtime

- A runtime can provide a coherent and consistent view of the main memory
- E.g., COMIC - a software shared virtual memory \( \text{[Lee et al., PACT’08]} \)
- Software-Managed Cache (SMC) implemented in the local memory is a critical component of this runtime
Software-Managed Caches

- Fast memory access on a hit
  - Similar to hardware caches
- Major differences from hardware caches
  - Flexibility
    - Adjust cache parameters both on-line and off-line
  - Transparency
    - Not transparent to the user, need user API functions
- Performance
  - Depend on the number of misses and the cache implementation cost
Software-Managed Caches (contd.)

- The design and implementation of a high performance SMC requires a cache management algorithm with
  - Low implementation complexity and
  - A low cache miss rate
- The Extended Set-Index Cache meets all of the above requirements
- A single cache line size or a single replacement policy does not yield the best performance
  - Adaptive execution strategy!
- Applicable to all cores with access to both local and global memory in multicore architectures
Extended Set-Index Cache (ESC)

- Based on the 4-way set-associative cache (4WC)
- The TEs and the data blocks are decoupled
  - Mapping is determined at run time
  - # of TEs can be greater than # of cache lines (4 x S ≥ N)
- Tag comparison using a SIMD instruction
- Hash function: h(x) = x modulo S

![Diagram of ESC cache](image)
Extended Set-Index Cache (contd.)

- Determining hit or miss
  - Same as the 4-way set-associative cache

```
4 x S ≥ N
```
Extended Set-Index Cache (contd.)

- A global address is requested through the cache API
- The hash function computes the set index

```
4 \times S \geq N
```

![Diagram showing Tag Entry Array, Cache Lines, and Line Table](image)

ACACES 2009
Extended Set-Index Cache (contd.)

- Compare the tag
- Use a SIMD instruction
Extended Set-Index Cache (contd.)

- **Cache hit**
- There is a valid matching tag

```
4 × S ≥ N
```

```
Tag Entry Array
```

```
Cache Lines
```

```
Line Table
```

**Fig. 1** Cache hit

```
tag  offset
```

```
tag  line index
```

**TE**

**hash**

**a hit**

```
line 0
```

```
line 1
```

```
line 2
```

```
tag 0
```

```
tag 1
```

```
tag 2
```

```
tag N-1
```

ACACES 2009
Extended Set-Index Cache (contd.)

- **Cache miss**
  - There is no matching tag in the set

![Diagram showing Tag Entry Array, Cache Lines, and Line Table with no matching tag, indicating a miss.]
Extended Set-Index Cache (contd.)

- When there is no empty slot in the set
  - Select a victim in the set
  - The same as the process of the 4-way set-associative cache
Extended Set-Index Cache (contd.)

- When there is an empty slot in the set
  - Select a victim from all the cache lines using the Line Table

An empty slot
Extended Set-Index Cache (contd.)

- Evict the victim to the main memory if it is dirty
- Otherwise, just invalidate it

\[ 4 \times S \geq N \]
Extended Set-Index Cache (contd.)

- Fetch the requested line to the location of the victim
- Map the empty TE to the fetched line

- Tag Entry Array
- Cache Lines
  - line 0
  - line 1
  - line 2
  - line N-1
- Line Table
  - V D tag 0
  - V D tag 1
  - V D tag 2
  - V D tag N-1

4 x S ≥ N
Replacement Policies

- FIFO, Clock, and LRU
- For fully associative SMCs
  - Replacement policy is applicable to all cache lines
- For set-associative SMCs
  - To the lines in a set
- For the ESC
  - If there is no empty TE in the set, to the lines in a set
  - Otherwise, to all cache lines
Implementation of Replacement Policies

- FIFO
  - A cache-line pointer to the last victim
  - The pointer is maintained in a round-robin manner
- Clock
  - An approximation of LRU
  - A reference bit for each cache line
  - A pointer to the cache lines (line pointer)
- LRU
  - A time stamp for each cache line
  - Linear search of a cache line with the smallest time stamp
V-Way Cache

- A hardware cache proposed by Qureshi et al. [ISCA’05]
- Essentially the same decoupled cache structure
- Differences
  - The ESC is more flexible
    - V-Way cache is a hardware cache
  - The reuse replacement policy used in the V-Way cache is not appropriate for SMCs
    - Due to the cost of software implementation
  - The ESC can fully utilize the available local memory space
    - Depending on the application needs
Indirect Index Cache

- A hardware cache proposed by Hallnor et al. [ISCA’00]
- Based on the 4-way set-associative cache
- The mapping between TEs and cache lines is dynamic
  - Conflicting TEs are stored in the Chain Storage
  - Essentially the same as a fully associative cache
- Large overhead in the software implementation due to the sequential search of TEs in the Chain Storage
Benefits of the ESC

• Fast tag search
  • Comparable to the 4-way set-associative cache
• Reduced conflict misses
  • Address collisions are distributed over the extended sets
  • With a sufficiently large number of sets
    • # of misses of the ESC = # of misses of the fully associative cache with the same number of cache lines
• Improved utilization of the local memory space
  • # of cache lines ≤ # of TEs
• Fast set index computation
  • Can always make the # of sets = a power of 2
  • A low-cost bit-wise AND operation can be used
Using a Fetch Buffer

- On a cache miss

Select a victim → Flush the victim → Fetch the block
Using a Fetch Buffer

- On a cache miss

Select a victim → Flush the victim → Fetch the block

These happen sequentially
Using a Fetch Buffer

- On a cache miss

1. Select a victim
2. Flush the victim
3. Fetch the block

These happen sequentially

Overlap using a fetch buffer
Using a Fetch Buffer

- On a cache miss

Select a victim → Flush the victim → Fetch the block

These happen sequentially

Overlap using a fetch buffer

Fetch the block into the fetch buffer
Using a Fetch Buffer

- On a cache miss

Select a victim → Flush the victim → Fetch the block

These happen sequentially

Overlap using a fetch buffer

Fetch the block into the fetch buffer

Select a victim → Flush the victim
Using a Fetch Buffer

- On a cache miss

Select a victim → Flush the victim → Fetch the block

These happen sequentially

Overlap using a fetch buffer

Fetch the block into the fetch buffer

Select a victim → Flush the victim

overlap
Using a Fetch Buffer

- On a cache miss

Select a victim → Flush the victim → Fetch the block

These happen sequentially

Overlap using a fetch buffer

Fetch the block into the fetch buffer

Select a victim → Flush the victim

overlap

Victim line → Next fetch buffer
Using a Fetch Buffer

- On a cache miss

Select a victim → Flush the victim → Fetch the block

These happen sequentially

Overlap using a fetch buffer

Fetch the block into the fetch buffer

Victim line → Next fetch buffer

Select a victim → Flush the victim

The fetch buffer reduces execution time (up to 26%)

Overlap
Evaluation

- Environments
  - Cell Blade Server
    - Dual 3.2GHz Cell BE, 8 SPEs each
      - only one processor is used
    - 2GB main memory, RedHat Linux ES 5.1
    - PPE: 512KB L2 cache
    - SPE: 256KB local memory, asynchronous DMA
  - Applications
    - 8 OpenMP applications
      - CG, FT, IS, MG (NAS Parallel Benchmarks)
      - equake, swim (SPEC OMP), jacobi, md (www.openmp.org)
    - Iterations in each parallel loop are distributed over 8 SPEs
    - SPE uses the SMCs to access the main memory
  - Runtime
    - COMIC: a software shared virtual memory [Lee et al. PACT’08]
    - Guarantees coherence between the SMCs in local memory and the main memory
SMC Configurations

- 160KB local memory space for cache lines
- To fully utilize the 256KB local memory
- For FAC, 4WC, and IIC
  - # of sets is not a power of 2 → need a modulo operation
- For the ESC
  - Can choose any # of sets
  - # of sets is set to the power of 2 → can use a bitwise AND operation
- A fetch buffer is used
- Vary the cache line size from 1KB to 8KB
- Replacement policies: FIFO, Clock, LRU
Number of Misses

Normalized Number of Misses

Normalized Number of Misses
Number of Misses

Normalized Number of Misses

Normalized Number of Misses

FAC = IIC <= ESC < 4WC

ACACES 2009
## Effect of the Number of Sets

<table>
<thead>
<tr>
<th>FAC</th>
<th>ESC</th>
<th>8</th>
<th>10</th>
<th>12</th>
<th>14</th>
<th>16</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>CG</td>
<td>0.45</td>
<td>2.19</td>
<td>0.60</td>
<td>0.50</td>
<td>0.51</td>
<td>0.45</td>
<td>0.45</td>
</tr>
<tr>
<td>equake</td>
<td>0.20</td>
<td>0.22</td>
<td>0.21</td>
<td>0.21</td>
<td><strong>0.20</strong></td>
<td>0.20</td>
<td>0.20</td>
</tr>
</tbody>
</table>

Miss rate according to the # of sets

- 128KB local memory space for cache lines
- 4KB cache line
- LRU replacement policy
- Vary the # of sets from 8 to 32
### Effect of the Number of Sets

<table>
<thead>
<tr>
<th></th>
<th>FAC</th>
<th>ESC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>CG</td>
<td>2.19</td>
<td>0.60</td>
</tr>
<tr>
<td>equake</td>
<td>0.22</td>
<td>0.21</td>
</tr>
</tbody>
</table>

Miss rate according to the # of sets

As the # of sets increases, the miss rate of the ESC decreases and approaches to the miss rate of the FAC

- 128KB local memory space for cache lines
- 4KB cache line
- LRU replacement policy
- Vary the # of sets from 8 to 32
Execution Time
Execution Time

ESC is the fastest
ESC is the fastest

Miss rate comparable to the FAC and
the smallest computational overhead
Adaptive Execution Strategy

- For each parallel loop, we identify the optimal line size and replacement policy adaptively at run time
- Compare the execution time per iteration (TPI) of the current invocation to that of the previous invocation
- Across invocations, vary the line size and replacement policy until we find an optimal
Adaptive Execution Strategy

- In the first stage, we identify an optimal line size
- A hill climbing algorithm with 5 levels of different line sizes

1st invocation

4KB lines (TPI4)

2KB lines (TPI2)

1KB lines (TPI1)

TPI2 < TPI4

TPI1 < TPI2

Yes

No

8KB lines (TPI8)

TPI4 < TPI8

Yes

No

TPI8 < TPI16

16KB lines (TPI16)

No

Yes

16KB is an optimal

8KB is an optimal

2KB is an optimal

4KB is an optimal
Adaptive Execution Strategy

- In the first stage, we identify an optimal line size
- A hill climbing algorithm with 5 levels of different line sizes
Adaptive Execution Strategy

- In the first stage, we identify an optimal line size
- A hill climbing algorithm with 5 levels of different line sizes
Adaptive Execution Strategy

- In the first stage, we identify an optimal line size.
- A hill climbing algorithm with 5 levels of different line sizes.
Adaptive Execution Strategy

- In the first stage, we identify an optimal line size
- A hill climbing algorithm with 5 levels of different line sizes

1\textsuperscript{st} invocation

- 4KB lines (TPI\textsubscript{4})
- 2KB lines (TPI\textsubscript{2})

2\textsuperscript{nd} invocation

3\textsuperscript{rd} invocation

- 1KB lines (TPI\textsubscript{1})
- TPI\textsubscript{1} < TPI\textsubscript{2}
- Yes
- TPI\textsubscript{2} < TPI\textsubscript{4}
- No
- 8KB lines (TPI\textsubscript{8})
- TPI\textsubscript{4} < TPI\textsubscript{8}
- No
- TPI\textsubscript{8} < TPI\textsubscript{16}
- Yes
- 16KB is an optimal
- 4KB is an optimal

- 1KB is an optimal
- 2KB is an optimal
- Yes
- 16KB lines (TPI\textsubscript{16})
- 8KB is an optimal
Result of Adaptive Execution

- Line size
  - On average, 20% slower than the optimal
- Line size + replacement policy
  - On average, 14% slower than the optimal
- Without the adaptive execution strategy
  - In some cases, significant performance degradation, up to 3.5 times
Further Readings

COMIC: an SVM for Multicores with Local Memory
Target Architecture

- PPE has two levels of cache hierarchy
  - Coherent to the main memory
  - Running an OS
- SPEs have no caches
  - Non-coherent local stores (256KB each)
  - Explicit DMA transfers between local stores or between local stores and the main memory
The Cell BE Architecture

- Hard to program
  - The programmers need to consider,
    - Explicit communication between processing elements (PEs) through mailboxes and signals
  - DMA transfers
  - Small local store
  - Coherence
- Need a good parallel programming model
  - Important to balance between delivering performance and ease of programming
COherent shared Memory Interface for Cell BE
Software shared virtual memory (SVM)
  A single, globally shared address space between PEs
  Centralized release consistency
  Software-managed cache in the local store
    Page-level caching
Runtime system (library)
SPMD-style parallel programming model
Open-source software
Downloadable at http://aces.snu.ac.kr
On Sony PS3, IBM Cell Blade, and PCIe Cell BE accelerator board
Many SVM proposals

- For loosely-coupled distributed memory multiprocessors
- Munin [SOSP’91], Midway [CMU-TR 1991], TreadMarks [ISCA’92, USENIX’94], Cashmere [SOSP’97], Shasta [ASPLOS’96, HPCA’98], etc.

Not good for Cell BE

Because,

- No MMU in an SPE for detecting and handling page faults
- The local store is too small
  - To support storing coherence information and page-level caching
  - Need to handle local store overflow
- Processing asynchronous messages in the SPE is expensive
- SPE initiates coherence actions and the PPE handles them
Generating a Cell BE Executable

app.c → programmer or compiler

Spe compiler → app_spe.o → SPE linker → SPE executable

COMIC_lib_SPE.so

PPE compiler → app_ppe.o → PPE linker → Cell BE executable

app_ppe.c

app_spe.c
COMIC Thread Model

- Threads cooperate in a single shared address space
- PPE thread
  - Assigns tasks and provides SPEs with OS services
  - Manages synchronization and coherence actions
  - May participate in the computation
- SPE threads
  - Perform tasks assigned
An OpenMP program

```c
#include <stdio.h>
#include <omp.h>
#define SIZE 24
int a[ SIZE ];

int main()
{
    int i;

#pragma omp parallel for
    for( i=0; i < SIZE; i++ )
        a[i] = i;

    for( i=0; i < SIZE; i++ )
        printf( "%d\n", a[i] );

    return 0;
}
```

i=0 i=1 ... i=SIZE-1
time
#include <comic_spe.h>
define SIZE 24

COMIC_shared ( int a[ SIZE ]; )

void spe_main_case0()
{
  int i, tid, chunk, low, high;
  tid = COMIC_get_spe_thread_id();
  chunk = ...;
  low = ...;
  high = ...;

  for ( i = low; i < high; i++ )
    COMIC_spe_write_int( &COMIC_access(a[i]), i );

  COMIC_spe_barrier();
}

COMIC_spe_main
{
  case 0: spe_main_case0();
  break;
}

#pragma omp parallel for
for( i=0; i < SIZE; i++ )
a[i] = i;
```c
#include <stdio.h>
#include <comic_ppe.h>
#define SIZE 24

COMIC_shared ( int a[ SIZE ]; )

int main()
{
    int i;

    COMIC_ppe_init();

    COMIC_ppe_run( 0 ); /* assigning task 0 to the SPEs */
    for( i=0; i < SIZE; i++ )
        printf( "%d\n", COMIC_access( a[i] ) );

    COMIC_ppe_exit();

    return 0;
}

#pragma omp parallel for
    for( i=0; i < SIZE; i++ )
        a[i] = i;
```
Software-Managed Cache (Page Buffer)

- The Extended Set-Index Cache (ESC) in each SPE’s local store
Centralized Release Consistency (CRC)

- Key component in the parallel programming model
- Similar to the home-based lazy release consistency (HLRC \{OSDI’96\})
  - Each page has a home node
- A single centralized home for all pages
  - The PPE (main memory) is the home node
  - All synchronization (locks and barriers) and coherence requests are handled by the PPE in a round-robin manner
- Modified pages are sent to the main memory (the PPE) when
  - SPE performs a release
  - SPE’s cache overflows
CRC (contd.)

SPE0

PPE

SPE1
CRC (contd.)

SPE0

PPE

SPE1

page 36

a copy of page 36
CRC (contd.)

SPE0
acquire

PPE
page 36

SPE1
a copy of page 36
CRC (contd.)

SPE0

acquire

PPE

page 36

SPE1

a copy of page 36

grant
CRC (contd.)

SPE0

acquire

W(u)

PPE

page 36

grant

SPE1

a copy of page 36
CRC (contd.)

SPE0 → PPE

acquire

W(u)

PPE → SPEI

page 36

a copy of page 36
CRC (contd.)

SPE0  
acquire  
W(u)  

PPE  
page 36  

SPE1  
a copy of page 36
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

grant

page 36

SPE1

a copy of page 36
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

page 36

grant

SPE1

a copy of page 36
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

page 36

SPE1

a copy of page 36
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

page 36

SPE1

acquire

a copy of page 36
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

page 36

SPE1

a copy of page 36

acquire

grant & inval

grant & inval
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

page 36

acquire

grant

grant & inval

R(v)

SPE1

a copy of page 36
CRC (contd.)

SPE0

acquire

W(u)

release

grant

PPE

page 36

SPE1

a copy of page 36

acquire

R(v)

grant & inval
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

page 36

acquire

grant

R(v)

grant & inval
CRC (contd.)

SPE0

acquire

W(u)

release

PPE

page 36

grant & inval

R(v)

SPE1

a copy of page 36

grant

release
Multiple Writer Protocol

- Multiple SPEs can write to different locations of the same page at the same time without synchronization
- Reduces the communication overhead due to false sharing (i.e., avoids the ping pong effect)
### Multiple-Writer Protocol (contd.)

<table>
<thead>
<tr>
<th>SPE0</th>
<th>SPE1</th>
<th>SPE2</th>
<th>PPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>write</td>
<td>twins</td>
<td>invalidate</td>
</tr>
<tr>
<td>{    }</td>
<td>{    }</td>
<td>o</td>
<td>{    }</td>
</tr>
<tr>
<td>{    }</td>
<td>{    }</td>
<td>p</td>
<td>{    }</td>
</tr>
<tr>
<td>{    }</td>
<td>{    }</td>
<td>q</td>
<td>{    }</td>
</tr>
<tr>
<td>{    }</td>
<td>{    }</td>
<td>r</td>
<td>{    }</td>
</tr>
<tr>
<td>{    }</td>
<td>{    }</td>
<td>s</td>
<td>{    }</td>
</tr>
</tbody>
</table>

- **The PPE maintains,**
  - A directory entry for each page
  - read, write, and twins fields
  - An invalidate list for each SPE
  - The IDs of pages that need to be invalidated when the SPE performs an acquire

- **Whenever a write fault occurs, a twin of the faulting page is created**
Multiple-Writer Protocol (contd.)

SPE0    SPE1    SPE2
R(p)    W(p)    W(p)
R(r)    R(q)    R(s)
acquire(A)
W(p)    release(B)

read           write
{ }            { }    {0,1,2}    {1,2 }
{1}            { }    {0}     { }
{2}            { }    { }     { }

PPE
SPE0          SPE1          SPE2
twins
p             p
q             r
s

invalidate
{ }    { }    { }

(a)             (b)         (c)

(d)             (e)
Multiple-Writer Protocol (contd.)

SPE0       SPE1       SPE2
R(p)       W(p)       W(p)  read   | write   |
R(r)       R(q)       R(s)  {0,1,2} | {1,2}   |
{0}        {1}        {2}   SPE0   SPE1   SPE2
SPE0       SPE1       SPE2
acquire(A) | acquire(B) | release(B)   twins
            |            | (a)           p
            |            | (b)           p
W(p)       W(p)       W(p)  {0,1,2} | {1,2}   |
R(s)       R(r)       R(q)  {0}        {1}        {2}   SPE0   SPE1   SPE2
invalidates

SPE1 performs a release for the location B (page p in SPE1 needs to be flushed to the main memory)
Multiple-Writer Protocol (contd.)

SPE0  SPE1  SPE2
\begin{align*}
R(p) & \quad W(p) & \quad W(p) \\
R(r) & \quad R(q) & \quad R(s) \\
\end{align*}

PPE
\begin{align*}
\text{read} & \quad \text{write} \\
\{0,1,2\} & \quad \{1,2\} \\
\{1\} & \quad \{\} \\
\{0\} & \quad \{\} \\
\{2\} & \quad \{\} \\
\end{align*}

SPE0  SPE1  SPE2
\begin{align*}
\text{twins} & \\
p & \quad p \\
q & \quad r \\
s & \quad s \\
\end{align*}

SPE1’s local copy of p is compared to its twin
Page p in the main memory is updated with the difference.
Multiple-Writer Protocol (contd.)

1. Page p is inserted in the invalidate list of SPE0 and SPE2
Multiple-Writer Protocol (contd.)

1. Page p is inserted in the invalidate list of SPE0 and SPE2

2. W → R
Multiple-Writer Protocol (contd.)

The twin is discarded
Multiple-Writer Protocol (contd.)

SPE0  SPE1  SPE2

R p  R p  W p
R r  R q  R s

read  write
{ }  { }  {0,1,2}  {2}
{1}  { }  {0}  {2}
{0}  { }  {1}  {2}
{2}  { }  {2}  {0}

PPE

SPE0  SPE1  SPE2

write
{0,1,2}  {2}  {p}
{1}  { }  {q}
{0}  { }  {r}
{2}  { }  {s}

invalidation
{p}  { }  {p}

SPE0 tries to write to page p
(write fault, R → W)
Multiple-Writer Protocol (contd.)

1. A twin is created
Multiple-Writer Protocol (contd.)

1. A twin is created

2. Page p is fetched
Multiple-Writer Protocol (contd.)

1. A twin is created

2. Page p is fetched

3. p is removed from the list
Multiple-Writer Protocol (contd.)

SPE0 writes to page p
Lazy Twin Creation

- Creating a twin is an expensive operation
  - Main memory allocation
  - Copying overhead
- The PPE postpones the twin creation of a page until a dirty copy updates the page in the main memory
- Similar to the copy-on-write mechanism in operating systems
No Asynchronous Messages Sent to SPEs

- Asynchronous messages are detrimental to the performance of SPEs
- All coherence actions are initiated by SPEs, but handled by the PPE
- Three-way handshakes

On a miss, to fetch a page

- Lock the page p
- If write, create a twin
- Send the address of p
- DMA receive for the page p
- DMA completion notification
- Unlock the page p

On an overflow or release, to flush a page

- Flush request of a page p
- Send the address of the space for the modified copy of the page p
- DMA send for the page p
- DMA completion notification
- Lock the page p
- Update the page p
- Unlock the page p
Optimization Techniques for COMIC

- Variable page size
  - The optimal page size for the software-managed cache varies for different parallel regions

- Access localization in a page
  - No check for cache miss or hit except the first access to a page

- Read-only accesses
  - If a page is only read (not written) by the SPEs, no three-way handshaking for fetching is necessary

- Single writers
  - If a page is written by only one SPE, no three-way handshaking is necessary

- Assigning tasks to the PPE
  - If we can guarantee that a parallel region does not need any coherence action
## Benchmark Applications

<table>
<thead>
<tr>
<th>Application</th>
<th>Source</th>
<th>Input</th>
<th>Shared Data Size</th>
<th>Page Buffer Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>SPEC2K</td>
<td>reference</td>
<td>3.9MB + malloc</td>
<td>128KB</td>
</tr>
<tr>
<td>CG</td>
<td>NAS</td>
<td>class A</td>
<td>26.1MB</td>
<td>192KB</td>
</tr>
<tr>
<td>EP</td>
<td>NAS</td>
<td>class A</td>
<td>16.2KB</td>
<td>192KB</td>
</tr>
<tr>
<td>equake</td>
<td>SPEC2K</td>
<td>reference</td>
<td>28.5MB</td>
<td>192KB</td>
</tr>
<tr>
<td>FT</td>
<td>NAS</td>
<td>class A</td>
<td>417.7MB</td>
<td>168KB</td>
</tr>
<tr>
<td>IS</td>
<td>NAS</td>
<td>class A</td>
<td>64MB + malloc</td>
<td>192KB</td>
</tr>
<tr>
<td>jacobii</td>
<td>OpenMP</td>
<td>2000*2000</td>
<td>91.6MB</td>
<td>192KB</td>
</tr>
<tr>
<td>md</td>
<td>OpenMP</td>
<td>8192 particles</td>
<td>768KB</td>
<td>192KB</td>
</tr>
<tr>
<td>MG</td>
<td>NAS</td>
<td>class A</td>
<td>464B + malloc</td>
<td>176KB</td>
</tr>
<tr>
<td>SP</td>
<td>NAS</td>
<td>class A</td>
<td>79.6MB</td>
<td>192KB</td>
</tr>
<tr>
<td>STREAM</td>
<td>HPCC</td>
<td>N=1, Ns=1000</td>
<td>1.9MB</td>
<td>192KB</td>
</tr>
<tr>
<td>SWIM</td>
<td>SPEC2K</td>
<td>reference</td>
<td>190.6MB</td>
<td>192KB</td>
</tr>
</tbody>
</table>
Ease of Programming

- The most difficult task in the manual porting is identifying all shared memory accesses
- For OpenMP programs, it is easy
- The 12 OpenMP applications were ported by 4 first year graduate students
- Did not have any parallel programming experience
- It took approximately one month
  - to learn OpenMP semantics
  - to port the applications to COMIC runtime
System Configurations

- **Cell Blade Server (QS21)**
  - Dual 3.2GHz Cell BE
  - 8 SPEs, 512KB L2 for each processor
  - 2GB main memory
  - RedHat Linux ES 5.1
  - Only a single Cell processor is used

- **Intel Xeon Server**
  - Dual 1.6GHz Xeon quadcore (Clovertown, E5310)
  - A shared 4MB L2 cache for each pair of cores
  - 12GB main memory
  - SUSE Linux ES 10.0
Sensitivity to Page Sizes

- The optimal page size is different from application to application (from program region to program region)
- Adaptive execution techniques
Speedup over a single PPE

- With an optimal page size for each application
- COMIC performs as well as or better than CBEXLC for all but IS
- COMIC(opt) performs the best for all but ammp
Comparison to Homogeneous Multicores

- GCC and ICC: speedup over a single Xeon core
- COMIC(opt): speedup over a single PowerPC core
Transistor Cost/Performance Ratio

- Not an apples-to-apples comparison
  - But, to give you an idea
- ICC is on average 1.89 times faster than COMIC
- COMIC is 4.98 times cheaper than ICC in the number of transistors
  - Two quadcore Xeon chips: 1,166 M
  - A Cell BE chip: 234 M
- With the same transistor cost,
  - COMIC achieves 2.63 times better performance than ICC
## Effectiveness of the Optimizations

<table>
<thead>
<tr>
<th>Application</th>
<th>Access Localization</th>
<th>Variable Page Size</th>
<th>Read-Only Accesses</th>
<th>Single Writer</th>
<th>Tasks to the PPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ammp</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CG</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>EP</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>equake</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>FT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>IS</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>jacobi</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>md</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>MG</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SP</td>
<td>X</td>
<td></td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>STREAM</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>SWIM</td>
<td>X</td>
<td></td>
<td>X</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>
Further Readings


- Sequoia: Programming the Memory Hierarchy. Kayvon Fatahalian, Daniel Reiter Horn, Timothy J. Knight, Larkhooon Leem, Mike Houston, Ji Young Park, Mattan Erez, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Proceedings of the 2006 Supercomputing Conference (SC’06), November 2006


Future Directions for Explicitly-Managed Memory Hierarchies

- Realtime guarantee with scratchpad memory
  - Predictable scratchpad memory management
  - WCET analysis and optimization
- Data overlaying techniques
  - Without profiling information, it is hard
- Locality enhancement techniques
  - Prefetching techniques
- Intelligent memory management using machine learning techniques
- Transactional memory
- Memory management techniques for emerging multicore architectures
  - Cluster of (heterogeneous) multicores
  - GPGPUs (e.g., Intel Larrabee)
  - Embedded multicores (e.g., TI OMAP4)
- Parallel programming models