Managing Distributed, Shared L2 Caches through OS-Level Page Allocation

January 21, 2020
In the last episode...
Problem: CPUs are cheap and fast.
Memories are cheap, fast, capacious (choose two out of three).
Fast CPU with slow memory doesn’t make sense.

CPU looks for data in the cache; if it finds it, it gets it from there and continues to work; if it doesn’t find it, it looks for it in main memory.
Smaller, faster, and costlier (per byte) storage devices

Larger, slower, and cheaper (per byte) storage devices

L0: CPU registers hold words retrieved from cache memory.

L1: L1 cache (SRAM) holds cache lines retrieved from L2 cache.

L2: L2 cache (SRAM) holds cache lines retrieved from L3 cache.

L3: L3 cache (SRAM) holds cache lines retrieved from memory.

L4: Main memory (DRAM) holds disk blocks retrieved from local disks.

L5: Local secondary storage (local disks) holds files retrieved from disks on remote network servers.

L6: Remote secondary storage (distributed file systems, Web servers)
<table>
<thead>
<tr>
<th>memory</th>
<th>latency</th>
<th>size</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1</td>
<td>4 cycles</td>
<td>32KiB</td>
</tr>
<tr>
<td>L2</td>
<td>10 cycles</td>
<td>256KiB</td>
</tr>
<tr>
<td>L3</td>
<td>40-75 cycles</td>
<td>8MiB</td>
</tr>
<tr>
<td>DRAM</td>
<td>600 cycles</td>
<td>8GiB</td>
</tr>
<tr>
<td>HDD</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>
It was working
It was working; but now...
How to live?
How to live?

Private cache?
How to live?

Private cache? Shared cache?
Private cache
Private cache
Each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them.
Private cache
Each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them.

Pros: low hit latency
Private cache
Each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them.

Pros: low hit latency
Cons: capacity misses
Private cache
Each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them.

Pros: low hit latency
Cons: capacity misses

Shared cache
Private cache
Each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them.

Pros: low hit latency
Cons: capacity misses

Shared cache
Single cache where each cache slice accepts only an exclusive subset of all memory blocks.
Private cache
Each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them.

Pros: low hit latency
Cons: capacity misses

Shared cache
Single cache where each cache slice accepts only an exclusive subset of all memory blocks.

Pros: better overall utilization of on-chip caching capacity, enforcing cache co-herence becomes simpler
Private cache
Each cache slice is associated with a specific processor core and replicates data freely as the processor accesses them.

Pros: low hit latency
Cons: capacity misses

Shared cache
Single cache where each cache slice accepts only an exclusive subset of all memory blocks.

Pros: better overall utilization of on-chip caching capacity, enforcing cache co-herence becomes simpler
Cons: cache hit latency will be longer
Figure 1. An example 16-core tiled processor chip and its tile (core).
Idea: give OS control over cache.
Idea: give OS control over cache.
Using simple shared cache hardware we can implement a private caching policy, a shared cache policy, or a hybrid of the two.
Idea: give OS control over cache.
Using simple shared cache hardware we can implement a private caching policy, a shared cache policy, or a hybrid of the two without any hardware support.
IBM Power 5!
\[ S = A \mod N \]
\[ S = A \mod N \]

Where \( S \) stands for the cache slice number, \( A \) for the memory block address, and \( N \) for the number of cache slices.
Problem: contiguous memory blocks are hosted by different cache slices
Problem: contiguous memory blocks are hosted by different cache slices
Solution: replacing A with physical page number (PPN)
Figure 2. (a) Physical memory partitioning and mapping to cache slice at the cache line granularity. (b) Physical memory partitioning and mapping at the memory page granularity. (c) Virtual to physical page mapping (P0, P1: process 0 and 1, VM: virtual memory).
\[ CG_i \ (0 < i < N - 1) \] – congruence group

\[ CG_i = \{ \text{physical page (PPN = j)} \mid \text{pmap}(j) = i \} \]

OS has control on pmap.
\( CG_i \ (0 < i < N - 1) \) – congruence group

\[ CG_i = \{ \text{physical page (PPN = } j \text{)} \mid \text{pmap}(j) = i \} \]

OS has control on pmap.

Private caching: for a page requested by a program running on core \( i \), allocate a free page from \( CG_i \)
\[ CG_i \ (0 < i < N - 1) \] – congruence group
\[ CG_i = \{ \text{physical page (PPN} = j) \mid \text{pmap}(j) = i \} \]

OS has control on pmap.

Private caching: for a page requested by a program running on core \( i \), allocate a free page from \( CG_i \).

Shared caching: for a requested page, allocate a free page from all the congruence groups using random selection, round-robin etc...
$CG_i$ \((0 < i < N - 1)\) – congruence group

$$CG_i = \{ \text{physical page (PPN = j)} \mid \text{pmap}(j) = i \}$$

OS has control on pmap.

Private caching: for a page requested by a program running on core \(i\), allocate a free page from \(CG_i\).

Shared caching: for a requested page, allocate a free page from all the congruence groups using random selection, round-robin etc…

Hybrid caching: partition \(\{ CG_i \}\) into \(K\) groups \((K < N)\); then define a mapping from a core to a group; for page requested by program running on core \(i\), allocate a free page from the group that core \(i\) maps to.
Figure 3. Program (“P”) and data locations determine the minimum distance to bridge them.
Figure 4. A Bloom filter based monitor mechanism to count actively accessed pages.
Figure 5. A virtual multicore (VM) example.
<table>
<thead>
<tr>
<th></th>
<th><strong>PRIVATE</strong></th>
<th><strong>SHARED (w/ line interleaving)</strong></th>
<th><strong>OS-BASED</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Hardware (tile)</strong></td>
<td>Similar to a conventional uniprocessor core with two-level caches; coherence mechanism (<em>e.g.</em>, directory) to cover L1 and L2 caches</td>
<td>Coherence enforcement is simpler and is mainly for L1 because L2 is shared by all cores</td>
<td>(Same as <strong>SHARED</strong>); simple hardware-based performance monitoring mechanism will help reduce monitoring overhead</td>
</tr>
<tr>
<td><strong>Software</strong></td>
<td>(NA)</td>
<td>(NA)</td>
<td>OS-level support, esp. in the page allocation algorithm</td>
</tr>
<tr>
<td><strong>Data Proximity</strong></td>
<td>Data items are attracted to local cache slices through active replication; limited caching space can result in performance degradation due to capacity misses</td>
<td>Fine-grained cache line interleaving results in non-optimal data distribution; there is no explicit control over data mapping</td>
<td>Judicious data mapping though page allocation can improve data proximity</td>
</tr>
<tr>
<td><strong>Network Traffic</strong></td>
<td>High coherence traffic (<em>e.g.</em>, directory look-up and invalidation) due to data replication [23]; increased off-chip traffic due to high on-chip miss rate</td>
<td>High inter-tile data traffic due to remote L2 cache accesses, often 2× to 10× higher than <strong>PRIVATE</strong> [30]; lower off-chip traffic due to larger caching capacity</td>
<td>Low off-chip traffic like <strong>SHARED</strong>; improved program-data proximity through page allocation and process scheduling leads to lower inter-tile traffic than <strong>SHARED</strong></td>
</tr>
</tbody>
</table>

**Table 1.** Comparing private caching, shared caching, and OS-based cache management approaches.
Test setup

CPU: 16 tiles (4x4); 16kB L1 I/D caches, 512kB L2 cache slice

L1 caches are four-way set associative with a 32-byte line size

Each 8-cycle L2 cache slice is 8-way set associative with 128-byte lines, two-cycle latency per each hop

2-GB off-chip main memory with 300 cycles latency
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Input</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>gcc compiler</td>
<td>reference (integrate.i)</td>
</tr>
<tr>
<td>parser</td>
<td>English parser</td>
<td>reference</td>
</tr>
<tr>
<td>eon</td>
<td>probabilistic ray tracer</td>
<td>reference (chair)</td>
</tr>
<tr>
<td>twolf</td>
<td>place &amp; route simulator</td>
<td>reference</td>
</tr>
<tr>
<td>wupwise</td>
<td>quantum chromodynamics solver</td>
<td>reference</td>
</tr>
<tr>
<td>galgel</td>
<td>computational fluid dynamics</td>
<td>reference</td>
</tr>
<tr>
<td>ammp</td>
<td>ODE solver for molecular dynamics</td>
<td>reference</td>
</tr>
<tr>
<td>sixtrack</td>
<td>particle tracking for accelerator design</td>
<td>reference</td>
</tr>
<tr>
<td>ffft</td>
<td>fast Fourier transform</td>
<td>4M complex numbers</td>
</tr>
<tr>
<td>lu</td>
<td>dense matrix factorization</td>
<td>$512 \times 512$ matrix</td>
</tr>
<tr>
<td>radix</td>
<td>parallel radix sort</td>
<td>3M integers</td>
</tr>
<tr>
<td>ocean</td>
<td>ocean simulator</td>
<td>$258 \times 258$ grid</td>
</tr>
</tbody>
</table>

**Table 2. Benchmark programs.**
Figure 6. Single program performance ($time^{-1}$) of different policies, relative to PRV.
<table>
<thead>
<tr>
<th></th>
<th>PRV</th>
<th>SL</th>
<th>SP-RR</th>
<th>SP40</th>
<th>SP60</th>
<th>SP80</th>
<th>PRV8</th>
</tr>
</thead>
<tbody>
<tr>
<td>gcc</td>
<td>2.9</td>
<td>0.1</td>
<td>0.5</td>
<td>1.8</td>
<td>2.1</td>
<td>2.8</td>
<td>0.1</td>
</tr>
<tr>
<td>parser</td>
<td>6.6</td>
<td>0.5</td>
<td>0.6</td>
<td>2.6</td>
<td>3.7</td>
<td>5.8</td>
<td>0.4</td>
</tr>
<tr>
<td>eon</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>twolf</td>
<td>16.3</td>
<td>0.1</td>
<td>0.1</td>
<td>1.6</td>
<td>7.3</td>
<td>13.1</td>
<td>0.0</td>
</tr>
<tr>
<td>wupwise</td>
<td>25.0</td>
<td>25.0</td>
<td>25.0</td>
<td>25.0</td>
<td>25.0</td>
<td>25.0</td>
<td>25.0</td>
</tr>
<tr>
<td>galgel</td>
<td>6.3</td>
<td>0.1</td>
<td>0.1</td>
<td>0.9</td>
<td>3.4</td>
<td>5.0</td>
<td>0.1</td>
</tr>
<tr>
<td>ammp</td>
<td>46.6</td>
<td>0.1</td>
<td>0.4</td>
<td>18.9</td>
<td>26.4</td>
<td>34.9</td>
<td>0.1</td>
</tr>
<tr>
<td>sixtrack</td>
<td>13.5</td>
<td>0.5</td>
<td>0.5</td>
<td>1.4</td>
<td>3.2</td>
<td>10.4</td>
<td>0.5</td>
</tr>
<tr>
<td>gcc</td>
<td>10.8</td>
<td>270.4</td>
<td>261.7</td>
<td>135.9</td>
<td>76.0</td>
<td>55.6</td>
<td>0.4</td>
</tr>
<tr>
<td>parser</td>
<td>8.7</td>
<td>96.8</td>
<td>96.5</td>
<td>40.4</td>
<td>18.9</td>
<td>18.2</td>
<td>0.5</td>
</tr>
<tr>
<td>eon</td>
<td>0.0</td>
<td>86.9</td>
<td>90.2</td>
<td>23.7</td>
<td>20.4</td>
<td>17.7</td>
<td>0.0</td>
</tr>
<tr>
<td>twolf</td>
<td>35.0</td>
<td>138.2</td>
<td>150.1</td>
<td>67.8</td>
<td>48.4</td>
<td>37.8</td>
<td>0.1</td>
</tr>
<tr>
<td>wupwise</td>
<td>35.1</td>
<td>39.4</td>
<td>39.9</td>
<td>20.3</td>
<td>15.6</td>
<td>10.1</td>
<td>0.1</td>
</tr>
<tr>
<td>galgel</td>
<td>38.0</td>
<td>412.0</td>
<td>406.6</td>
<td>185.8</td>
<td>132.3</td>
<td>76.2</td>
<td>0.6</td>
</tr>
<tr>
<td>ammp</td>
<td>441.7</td>
<td>810.9</td>
<td>803.4</td>
<td>424.6</td>
<td>361.9</td>
<td>306.9</td>
<td>0.5</td>
</tr>
<tr>
<td>sixtrack</td>
<td>9.6</td>
<td>57.2</td>
<td>60.9</td>
<td>22.0</td>
<td>18.9</td>
<td>15.8</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 3. L2 cache load miss rate and on-chip network traffic (message-hops) per 1k instructions.
Figure 7. Performance sensitivity to network traffic. Performance relative to $PRV$, no traffic case.
Figure 8. Performance of parallel workloads, relative to $PRV$. 
(standing ovation)