Exploring DMA-assisted Prefetching Strategies for Software Caches on Multicore Clusters

Christian Pinto and Luca Benini
Introduction
Introduction

- SPM used to mitigate DRAM latency: explicit management
- **Software caches[1]** to hide DRAM-to-SPM copies

STM STHORM:
- Multiple clusters
- 16 cores x cluster
- Per cluster shared SPM
- Per cluster DMA
- No data caches

Software caches are great, but…

```c
void main()
{
    /* initialization */

    int a = cache_lookup(address, &x);  // Cache miss! Blocking!

    int a = cache_lookup(address + K, &y);
    ...
}
```

Software caches are inherently synchronous: processors have to wait line refills to be completed

Cache lines can be prefetched asynchronously using the DMA engines available in the platform

Related work

**HW prefetch**

- Mitigate DRAM latency
- Improve performance available caching systems

**SW prefetch**

- Smart prefetch strategies

**Software prefetching for irregular memory references:**
  - Based on compiler hints

**Regular memory references:**
  - resolved using direct buffering

Our approach relies on programmer’s hint for irregular references, and still uses the SW cache for the regular ones. Direct buffering waste memory!!!

**Per cluster shared SW cache:**

- Internal data structures allocated in shared SPM (Tags table, Lines table)
  - Exploiting multi-bank design

- Fast Lookup function
  - Hash function: simple bitwise operations
  - 18 clock cycles Lookup & Hit

- Cache lines can be accessed in parallel
  - One lock associated to each cache line for thread safety

- Operation mode
  - Default mode: standard cache behavior
  - Object mode: reduce overhead minimizing number of cache accesses

“A Highly Efficient, Thread-Safe Software Cache Implementation for Tightly-Coupled Multicore Clusters”. Christian Pinto and Luca Benini
Cache lines software prefetch

Main goal: transform the target Software Cache into a pro-active entity

```c
void main(){
    /*intialization*/
    int a = cache_lookup(0x20000, &x);
    do_smthg(a);
    /*some more computation on the same line, further lookups - only hits*/
    int a = cache_lookup(0x20020, &y);
    ...
}
```

- Overlap of computation and next line prefetch
- HIT!!!! Processor saving clock cycles
Software prefetch schemes

Prefetch is sensitive to application’s memory access pattern

Regular access pattern:
  • Easy to predict
  • Benefits from automatic prefetch

Irregular access pattern:
  • Hard to predict
  • Benefits from static hints given by the programmer
    • Which line to prefetch next

We support both automatic and programmer assisted prefetch schemes
Many-core accelerators often used for computer vision applications:

```
foreach (subframe) do
    foreach (p in subframe) do
        for (N:=0; N<K) do
            execute function(p+N)
        done
    done
done
```

Nested Loops:
- Images processed in subframes
- Subframes (usually) scanned with fixed spatial pattern
  - Easily predictable
- Innermost loop accessing several adjacent pixels
  - Overlaps with the prefetch of the next line

Prefetch triggered only at first reference to each line:
- Minimize possibility of waiting for prefetch completion
**Automatic prefetch**

**Possible access patterns:**

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>9</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>13</td>
<td>14</td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>

We defined two automatic prefetch schemes:

- **Horizontal prefetching**
- **Vertical prefetching**

**Requires:**

- Extension of lookup routine
- Definition of extra data structures
- Definition of a prefetch routine
Applications with irregular memory patterns may benefit from programmer's hint!!!

```c
void prefetch_line ( void * ext_mem_address, cache_parameters * cache );
```

Explicitly programs the prefetch of a specific cache line

Some applications have long prologue (initialization phase):
- It might be exploited to pre-load data into the cache

```c
dma_req_t * prefetch_cache (void * emt_mem_address, uint32_t from, cache_parameters * cache );

void wait_cache_prefetch (dma_req_t * dma_events);
```
- 1 DMA event (32 bytes) per line prefetch:
  - 16 in total: considering at most 1 prefetch per core at same time
  - Allocated in a common shared buffer in SPM (events_buffer)
    - Circular buffer
    - Requires one extra lock (1 byte)

- 1 byte per cache line:
  - index of prefetch DMA event in events_buffer

- Enhanced line status register (1 byte per line):
  - DIRTY flag (already present in original impl, embedded in tag)
  - PREFETCHING flag (true if a line is prefetching)
  - FIRST_ACCESS flag (true only at the first access to the line)

Total memory overhead: 513 bytes + 2 bytes per cache line
Lookup routine extension

DMA event id retrieved by looking at the field associated to the line

- PREFETCHING
  - Y: Retrieve DMA event id
  - N: Wait for DMA transfer completion

- Lookup
  - Prefetch Next
    - Y: FIRST_ACCESS
      - Y: Next line prefetch invoked only at first access to the line
    - N: Prefetch Next

- FIRST_ACCESS
  - N: Prefetch Next
Line Prefetch Routine

1. Identify the prefetch candidate
2. Acquire its line lock
3. Sanity checks
   a) The line selected as victim is already prefetching
   b) The line to prefetch is already in cache
4. Acquire DMA event index (by finding the first free entry in event_buffer)
5. Compute source and destination addresses
6. Trigger the DMA transfer
7. Save the dma event into the event_buffer

a) and b) may overlap:
some other processor has programmed the prefetch of the same line
Experiments performed on an evaluation board of STHORM:

- STHORM acceleration fabric in 28 nm CMOS.
  - four computing clusters
    - 16 STxP70 pe cluster
    - processors working at a clock frequency of 430 MHz
    - shared data scratchpad of 256 KBytes.
    - memory bandwidth towards the external memory is 300 MB/sec.

- Xilinx ZYNQ ZC-702 chip (Host):
  - dual Core ARM Cortex A9

- Benchmarks implemented in OpenCL

- Software cache plain C code linked as external library to applications

- Experiments:
  - Prefetch overhead characterization
  - Real applications: Normalized Cross-Correlation (NCC), Viola-Jones face Detection, color conversion algorithm.
Single core runs: avoid any unwanted interference
32 bytes cache line size

<table>
<thead>
<tr>
<th>Inst/clk</th>
<th>Hit</th>
<th>Miss</th>
<th>Hit&amp;Prefetch</th>
<th>Miss&amp;Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prefetch</td>
<td>11/18</td>
<td>118/893</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Prefetch</td>
<td>21/46</td>
<td>-</td>
<td>127/197</td>
<td>215/1073</td>
</tr>
</tbody>
</table>

Two pipeline flushes added with prefetching code:
- Line not prefetching
- Not First access

**HIT CASE!!!**
Overhead characterization (2)

Single core runs: avoid any unwanted interference

~ same # instr
much less clk cycles

<table>
<thead>
<tr>
<th></th>
<th>Miss</th>
<th>Hit&amp;Prefetch</th>
<th>Miss&amp;Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prefetch</td>
<td>11/18</td>
<td>118/893</td>
<td>-</td>
</tr>
<tr>
<td>Prefetch</td>
<td>21/46</td>
<td>-</td>
<td>127/197</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>215/1073</td>
</tr>
</tbody>
</table>

Hit&Prefetch:
- 1st access to prefetched line
- Processor accessing the line gets hit instead of miss

893 – 197 = 696 clock cycles saved!!!
Overhead characterization (3)

Single core runs: avoid any unwanted interference
32 bytes cache line size

<table>
<thead>
<tr>
<th>Inst/clk</th>
<th>Hit</th>
<th>Miss</th>
<th>Hit&amp;Prefetch</th>
<th>Miss&amp;Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>No prefetch</td>
<td>11/18</td>
<td>118/893</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Prefetch</td>
<td>21/46</td>
<td>-</td>
<td>127/197</td>
<td>215/1073</td>
</tr>
</tbody>
</table>

Case less suffering from code increase due to prefetch!!!
Case study 1: Normalized Cross Correlation (NCC)

2 Input images:
  • Consecutive lines in the same column are accessed
  • One sw cache per input image, 32 KB each

Case Study 2: Face Detection

• Cascade classifier accessed through sw cache
• Object caching to reduce the number of cache accesses
• 64 KB sw cache, 32 bytes objects

Case Study 3: Color Conversion:

• Low re-usage of input data: image rows are swiped from first to last pixel
• 32 KB sw cache with 512 byte lines
Baseline software cache 40% slowdown wrt DMA impl:
• Tight innermost loop, only 3 arith operation per cache access

Vertical prefetch was already supposed to work better!!!
• Only 5% overhead w.r.t. hand-tuned DMA
Case study 1: NCC (2)

- # executed instructions increase from 19% to 24%
- Overall benefit of w.r.t. baseline sw cache up to ~ 40%
Case study 2: Face Detection

Initiated sw cache prefetching for all image (56x72)
- 7.5% Miss rate
- Slowdown reduced to 2.5x from 8.6x
- Due to slight reduction of # cache misses (86%)
- from 2483 to 435 misses over 33117 total cache accesses

Not enough work to hide sw cache overhead (due to high # misses)
Case study 3: Color Coversion

Even if data is not re-used line prefetching is still beneficial for the overall perf:

- **Slowdown w.r.t. DMA is reduced down to 13%**
Conclusion

- SW caches are a reactive component:
  - Processors wait for cache refills to be completed
  - Clock cycles waste

- DMA engines can be used to prefetch cache lines ahead of time

- Both automatic and programmer assisted prefetch studied

- **Thanks to prefetching sw cache performance much closer to hand tuned DMA: only 4% slowdown in the best case.**

- High reduction of miss rate: **up to 86%**

- Makes sw cache suitable for applications with low data re-usage
THANK YOU!!

ANY QUESTION??