Advanced cache memory optimizations

Computer Architecture

J. Daniel García Sánchez (coordinator)
David Expósito Singh
Francisco Javier García Blas

ARCOS Group
Computer Science and Engineering Department
University Carlos III of Madrid
1 Introduction

2 Advanced optimizations

3 Conclusion
Why do we use caching?

- To overcome the *memory wall*.

- To reduce both data and instructions access times.
  - Make memory access time nearer to cache access time.
  - Offer the illusion of a cache size approaching to main memory size.
  - Based on the *principle of locality*. 
Memory average access time

- 1 level.

\[ t = t_h(L1) + m_{L1} \times t_p(L1) \]

- 2 levels.

\[ t = t_h(L1) + m_{L1} \times (t_h(L2) + m_{L2} \times t_p(L2)) \]

- 3 levels.

\[ t = t_h(L1) + m_{L1} \times (t_h(L2) + m_{L2} \times (t_h(L3) + m_{L3} \times t_p(L3))) \]

- ...
Basic optimizations

1. Increase block size.
2. Increase cache size.
3. Increase associativity.
4. Introduce multi-level caches.
5. Give priority to read misses.
6. Avoid address translation during indexing.
Advanced optimizations

- **Metrics to be decreased:**
  - Hit time.
  - Miss rate.
  - Miss penalty.

- **Metrics to be increased:**
  - Cache bandwidth.

- **Observation:** All advanced optimizations aim to improve some of those metrics.
1 Introduction

2 Advanced optimizations

3 Conclusion
Advanced cache memory optimizations

- Advanced optimizations
- Small and simple caches

2. Advanced optimizations

- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Small caches

- **Lookup** procedures:
  - Select a line using the *index*.
  - Read line *tag*.
  - Compare to *address tag*.

- Lookup time is **increased** as cache size grows.

- A **smaller** cache allows:
  - Simpler lookup hardware.
  - Cache can better fit into processor chip.

- A **small cache improves lookup time**.
Simple caches

- **Cache simplification.**
  - Use mapping mechanisms as *simple* as possible.
  - **Direct mapping:**
    - Allows to *parallelize* tag comparison and data transfers.

- **Observation:** Most modern processors focus more on using small caches than on simplifying them.
Advanced cache memory optimizations

- Advanced optimizations
- Small and simple caches

**Intel Core i7**

- **L1 cache (1 per core)**
  - 32 KB instructions.
  - 32 KB data.
  - Latency: 3 cycles.
  - Associative 4(i), 8(d) ways.

- **L2 cache (1 per core)**
  - 256 KB
  - Latency: 9 cycles.
  - Associative 8 ways.

- **L3 cache (shared)**
  - 8 MB
  - Latency: 39 cycles.
  - Associative 16 ways.
2 Advanced optimizations

- Small and simple caches
- **Way prediction**
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Way prediction

**Problem:**
- **Direct mapping** → fast but many misses.
- **Set associative mapping** → less misses but more sets (slower).

**Way prediction**
- Additional bits stored for predicting the way to be selected in the next access.
- Block prefetching and compare to single tag.
  - If there is a miss, it is compared with other tags.
Advanced cache memory optimizations

- Advanced optimizations
- Pipelined access to cache

2 Advanced optimizations

- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Pipelined access to cache

- **Goal**: Improve cache bandwidth.

- **Solution**: Pipelined access to the cache in multiple clock cycles.

- **Effects**:
  - Clock cycle can be shortened.
  - A new access can be initiated every clock cycle.
  - Cache bandwidth is increased.
  - Latency is increased.

- Positive effect in **superscalar processors**.
2 Advanced optimizations

- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Non-blocking caches

- **Problem**: Cache miss leads to a **stall** until a block is obtained.

- **Solution**: Out-of-order execution.
  - **But**: How is memory accessed while a miss is resolved?

- **Hit during miss**
  - Allow accesses with hit while waiting.
  - Reduces miss penalty.

- **Hit during several misses / Miss during miss**:
  - Allow overlapped misses.
  - Needs multi-channel memory.
  - Highly complex.
2 Advanced optimizations

- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Multi-bank caches

- **Goal**: Allow simultaneous accesses to different cache locations.

- **Solution**: Divide memory into independent banks.

- **Effect**: Bandwidth is increased.

- **Example**: Sun Niagara
  - L2: 4 banks.
Advanced cache memory optimizations

- Advanced optimizations
- Multi-bank caches

## Bandwidth

- For increasing the bandwidth, it is necessary to distribute accesses across banks.

- **Simple approach**: Sequential interleaving  
  - Round-robin of blocks across banks.

<table>
<thead>
<tr>
<th>Block addr.</th>
<th>Bank 0</th>
<th>Block addr.</th>
<th>Bank 1</th>
<th>Block addr.</th>
<th>Bank 2</th>
<th>Block addr.</th>
<th>Bank 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td>1</td>
<td></td>
<td>2</td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>5</td>
<td></td>
<td>6</td>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td>9</td>
<td></td>
<td>10</td>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td>13</td>
<td></td>
<td>14</td>
<td></td>
<td>15</td>
<td></td>
</tr>
</tbody>
</table>
2 Advanced optimizations

- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Critical word first and early restart

- **Observation**: Usually processors need a single word to proceed.

- **Solution**: Do not wait until the whole block from memory has been transferred.

- **Alternatives**:
  - **Critical word first**: Reorder blocks so that first word is the word needed by the processor.
  - **Early restart**: Block received without reordering.
    - As soon as the selected word is received, the processor proceeds.

- **Effects**: Depends on block size → the larger the better.
2 Advanced optimizations
- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Write buffer

- A write buffer allows to decrease miss penalty.
  - When processor writes on buffer, it considers write is completed.
  - Simultaneous writes on memory are more efficient than a single write.

- Uses:
  - **Write-through**: On every write.
  - **Write-back**: When block is replaced.
Merges in write buffer

- If buffer contains modified blocks, addresses are checked and, if it is possible, processor performs overwrite.

**Effects:**
- Decrease number of *memory writes*.
- Decrease amount of *stalls* due to full buffer.
Merges in write buffer

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>M[100]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>108</td>
<td>1</td>
<td>M[108]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>116</td>
<td>1</td>
<td>M[116]</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>124</td>
<td>1</td>
<td>M[124]</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Write address</th>
<th>V</th>
<th>V</th>
<th>V</th>
<th>V</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>1</td>
<td>M[100]</td>
<td>0</td>
<td>M[108]</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>100</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Advanced optimizations

- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge

Compiler optimizations

- Hardware prefetching
Compiler optimizations

- **Goal**: Generate code with a reduced number of instructions and data misses.

- **Instructions**:
  1. Procedure reordering.
  2. Align code blocks to cache line start.
  3. Branch linearization.

- **Data**:
  1. Array merge.
  2. Loop interchange.
  3. Loop merge.
  4. Blocked access.
Procedure reordering

- **Goal**: Decrease *conflict misses* due to two concurrent procedures are mapped to the *same cache line*.
- **Technique**: Reorder procedures in memory.
Basic block alignment

- **Definition**: A basic block is a set of instructions sequentially executed (contains no branches).

- **Goal**: Decrease the cache misses possibility for sequential codes.

- **Technique**: Align the first instruction in a basic block with the first word in cache line.
Branch linearization

- **Goal**: Decrease cache misses due to conditional branches.

- **Technique**: If the compiler detects a branch is likely to be taken, it may invert condition and interchanges basic blocks in both alternatives.
  - Some compilers define extensions to hint the compiler.
  - **Example**: gcc (`__likely__`).
Array merge

Parallel arrays

vector<int> key;  
vector<int> val;  

for (int i=0; i<max; ++i) {  
    cout << key[i] << "," << val[i] << endl;  
}

Merged array

struct entry {  
    int key;  
    int val;  
};  
vector<entry> v;  

for (int i=0; i<max; ++i) {  
    cout << v[i].key << "," << v[i].val << endl;  
}

- Decrease conflicts.
- Improve spatial locality.
**Advanced cache memory optimizations**

- Advanced optimizations
- Compiler optimizations

---

**Loop interchange**

---

**Striped accesses**

```c
for (int j=0; j<100; ++j) {
    for (int i=0; i<5000; ++i) {
        v[i][j] = k * v[i][j];
    }
}
```

**Sequential accesses**

```c
for (int i=0; i<5000; ++i) {
    for (int j=0; j<100; ++j) {
        v[i][j] = k * v[i][j];
    }
}
```

---

**Goal**: Improve spatial locality.

- Depends on the storage model defined by the programming language.
  - FOTRAN versus C.
### Independent loops

```c
for (int i=0; i<rows; ++i) {
    for (int j=0; j<cols; ++j) {
        a[i][j] = b[i][j] * c[i][j];
    }
}
```

```c
for (int i=0; i<rows; ++i) {
    for (int j=0; j<cols; ++j) {
        d[i][j] = a[i][j] + c[i][j];
    }
}
```

### Merged loop

```c
for (int i=0; i<rows; ++i) {
    for (int j=0; j<cols; ++j) {
        a[i][j] = b[i][j] * c[i][j];
        d[i][j] = a[i][j] + c[i][j];
    }
}
```

- **Goal**: Improve temporal locality.
- **Beware**: It may decrease spatial locality.
Advanced cache memory optimizations

- Advanced optimizations
- Compiler optimizations

Blocked access

Original product

```c
for (int i=0; i<size; ++i) {
    for (int j=0; j<size; ++j) {
        r=0;
        for (int k=0; k<size; ++k) {
            r += b[i][k] * c[k][j];
        }
        a[i][j] = r;
    }
}
```

Blocked product

```c
for (bj=0; bj<size; bj+=bsize) {
    for (bk=0; bk<size; bk +=bs) {
        for (i=0; i<size; ++i) {
            for (j=bj; j<min(bj+bsize,size); ++j) {
                r=0;
                for (k=bk;k<min(bk+bsize,size); ++k) {
                    r += b[i][k] * c[k][j];
                }
                a[i][j] += r;
            }
        }
    }
}
```

- **bsize**: Block factor
2 Advanced optimizations

- Small and simple caches
- Way prediction
- Pipelined access to cache
- Non-blocking caches
- Multi-bank caches
- Critical word first and early restart
- Write buffer merge
- Compiler optimizations
- Hardware prefetching
Observation: Instructions exhibit high spatial locality.

Instruction prefetching:
- Read two consecutive blocks on miss.
  - Block causing the miss.
  - Next block.

Location:
- Block causing the miss → instruction cache.
- Next block → instruction buffer.
Data prefetching

- **Example**: Pentium 4.

- **Data prefetching**: Allows to prefetch a 4KB page to L2 cache.

  - Prefetching is invoked if:
    - 2 misses in L2 due to the same page.
    - Distance between misses lower than 256 bytes.
1 Introduction

2 Advanced optimizations

3 Conclusion
Summary (I)

- **Smaller and simpler caches**
  - **Improves**: Hit time.
  - **Worsens**: Miss rate.
  - **Complexity**: Very low.
  - **Observation**: Widely used.

- **Way prediction**:
  - **Improves**: Hit time.
  - **Complexity**: Low.
  - **Observation**: Used in Pentium 4.

- **Pipelined access to cache**:
  - **Improves**: Hit time.
  - **Worsens**: Bandwidth.
  - **Complexity**: Low.
  - **Observation**: Widely used.
Summary (II)

- **Non blocking access to cache:**
  - **Improves:** Bandwidth and miss penalty.
  - **Complexity:** Very high.
  - **Observation:** Widely used.

- **Multi-bank access to cache:**
  - **Improves:** Bandwidth.
  - **Complexity:** Low.
  - **Observation:** Used at L2 in Intel i7 L2 and Cortex A8.

- **Critical word first and early restart:**
  - **Improves:** Miss penalty.
  - **Complexity:** High.
  - **Observation:** Widely used.
Summary (III)

- **Write buffer merge**:
  - **Improves**: Miss penalty.
  - **Complexity**: Low.
  - **Observation**: Widely used.

- **Compiler optimizations**:
  - **Improves**: Miss rate.
  - **Complexity**: Low for HW.
  - **Observation**: Challenge is software.

- **Hardware prefetching**:
  - **Improves**: Miss penalty and miss rate.
  - **Complexity**: Very high.
  - **Observation**: More common for instructions than data accesses.
References

- **Computer Architecture. A Quantitative Approach**
  5th Ed.
  Hennessy and Patterson.
  **Sections**: 2.1, 2.2.

- **Recommended exercises**:
  - 2.1, 2.2, 2.3, 2.8, 2.9, 2.10, 2.11, 2.12