PACMAN: Program-level Approximately Optimal Cache Management

Xiaoming Gu, Chen Ding
Department of Computer Science
University of Rochester
• What is PACMAN?
  • A famous video game

• An ongoing compiler study to reduce cache misses using hardware bypassing supports
The Framework

Analysis → feedback

Transformation →

- identify references for bypassing
- tag references with bypassing
Outline

• Motivation
• Hardware Bypassing Support
• An Example
• Details of Analysis and Transformation
• Summary
Outline

• Motivation
• Hardware Bypassing Support
• An Example
• Details of Analysis and Transformation
• Summary
Motivation

• Running a sequential program alone on a multi-core chip
  • hard to parallelize the program
  • running alone for no interference
    • reduce cache misses

PACMAN: focus on reducing cache misses for sequential programs running alone
Outline

• Motivation
• Hardware Bypassing Support
• An Example
• Details of Analysis and Transformation
• Summary
Normal LRU Inst.

(a) Normal LRU at hit: $w$, assuming at entry $S_3$, is moved to the top of the stack

(b) Normal LRU at a miss: $w$ is placed at the top of the stack, evicting $S_m$

---

**Figure** The Normal LRU memory access instruction
Bypass LRU Inst.

(a) Bypass LRU at a hit: the bypass moves $S_3(w)$ to the bottom of the stack
(b) Bypass LRU at a miss: the bypass posits $w$ at the bottom of the stack, evicting $S_m$

Figure The Bypass LRU memory access instruction
Outline

- Motivation
- Hardware Bypassing Support
- An Example
- Details of Analysis and Transformation
- Summary
SOR

- Jacobi Successive Over-relaxation
- from NIST SciMark 2.0
- a classical stencil computation
- compiled by LLVM 2.7 using gold plugin with -O4

Algorithm
The original kernel of SOR

Require: G is a 2-dimensional double array with the size M*N
1: for p = 1; p < NUM_ITERATIONS; p++ do
2:   for i = 1; i < M-1; i++ do
3:     Gi = G[i];
4:     Gim1 = G[i-1];
5:     Gip1 = G[i+1];
6:     for j = 1; j < N-1; j++ do
7:       Gi[j] = 0.3125*(Gim1[j]+Gip1[j]+Gi[j-1]+Gi[j+1])-0.25*Gi[j];
8:     end for
9:   end for
10: end for
The Gap between LRU and OPT

- Two gaps
- The working sets (knees) of OPT are much smoother than LRU’s

**Figure**  The miss curves of SOR on fully-associative cache (cache line size = 64B, mc = miss curve, mrc = miss reduction curve)

NUM_ITERATIONS=10
M=N=512
The Transformation

**Algorithm** The kernel of SOR in SSA form with bypassing

**Require:** G is a 2-dimensional double array with the size M*N

1: for p = 1; p < NUM_ITERATIONS; p++ do
2:     for i = 1; i < M-1; i++ do
3:         Gi = G[i];
4:         Gilm1 = G[i-1];
5:         Gip1 = G[i+1];
6:         Gijm1 = Gi[0];
7:         Gij = Gi[1];
8:     for j = 1; j < N-1; j++ do
9:         Gilm1j = Gilm1[j];
10:        Gip1j = Gip1[j];
11:        Gijp1 = Gij[1];
12:        tmp1 = Gilm1j + Gip1j;
13:        tmp1 += Gijm1;
14:        tmp1 += Gijp1;
15:        tmp1 *= 0.3125;
16:        tmp2 = -0.25 * Gij;
17:        tmp1 += tmp2;
18:        Gi[j] = tmp1;
19:        Gijm1 = tmp1;
20:        Gij = Gijp1;
21:     end for
22: end for
23: end for

Normal and bypass LRU instructions mixed ==></LRU Bypassing
The Improvement of PACMAN

- The gap at the second working set disappears!

**Figure**  The miss curves of SOR on fully-associative cache (cache line size = 64B, mc = miss curve)

NUM_ITERATIONS=10
M=N=512
Outline

• Motivation
• Hardware Bypassing Support
• An Example
• Details of Analysis and Transformation
• Summary
The Analysis

- Simulate OPT
  - for a given cache configuration
  - each run-time access has three fields
    - data addr, static ref. ID, and bypassing flag (off by default)
  - when an eviction happens, set bypassing flag on for the previously last access to the victim
The Analysis (cont’d)

• Simulate OPT (cont’d)
  
  • calculate **bypassing ratios** for all memory references

  bypassing ratio of a reference =

  \[
  \frac{\text{#accesses generated by the reference and with bypassing flag on}}{\text{#accesses generated by the reference}}
  \]

  • the references with high bypassing ratios are the candidates for bypassing
The Transformation

• Loop unrolling

• find out the target references in IR using the candidates’ ref. IDs

• figure out the last touch to a cache line in the innermost loop body
  • the cache line size
  • the array element size
  • the loop step stride
  • the array indexing

• separate the last touch using loop unrolling

• tag the last touch with bypass LRU
SOR by PACMAN

NUM_ITERATIONS=10, M=N=512
fully-associative, 512KB, line size=64B

<table>
<thead>
<tr>
<th>memory reference</th>
<th>percentage of total accesses</th>
<th>byassing ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gim1[j]</td>
<td>24.3%</td>
<td>9.5%</td>
</tr>
<tr>
<td>Gip1[j]</td>
<td>24.3%</td>
<td>0%</td>
</tr>
<tr>
<td>Gi[j+1]</td>
<td>24.3%</td>
<td>0%</td>
</tr>
<tr>
<td>Gi[j] (def)</td>
<td>24.3%</td>
<td>0%</td>
</tr>
</tbody>
</table>

Table: The statistics of memory references

- **Gim1[j]** is the candidate
- only do bypassing for the last touch in the innermost loop body
- spatial locality retained
- use loop unrolling to do separation in practice
Why only the Second Working Set Improved?

- PACMAN simulation only at the second working set
- Reduce cache misses for the second working set

![Graph](image.png)

**Figure** The miss curves of SOR on fully-associative cache (cache line size = 64B, mc = miss curve)
Set-associative Cache

- Keep losing benefits on cache with lower associativities
- The improvement is still significant

**Figure** The miss curves of SOR on 16-way set-associative cache (cache line size = 64B, mc = miss curve, mrc = miss reduction curve)

**Figure** The miss curves of SOR on 4-way set-associative cache (cache line size = 64B, mc = miss curve, mrc = miss reduction curve)
With a Different Input

- The improvement is scalable with input sizes.

**Figure** The miss curves of SOR on fully-associative cache (cache line size = 64B, mc = miss curve, mrc = miss reduction curve)
Future Work

• extend PACMAN for general applications
• more realistic hardware environment
  • multi-level pseudo-LRU cache
  • the differences between loads and stores
  • the interaction with prefetching
Summary

- Use simple hardware bypassing supports
- Find out bypassing references by simulating OPT
- Cache misses are reduced very close to the optimal case
- Achieve significant improvement even on low associativity cache
- The training results can be used for a real run with a larger input size
Q & A

GAME NOT OVER