Utilization of Separate Caches to Eliminate Cache Pollution Caused by Memory Management Functions

Mehran Rezaei

Computer Science and Engineering Dept.
University of North Texas
{mehran,kavi}@cs.unt.edu
Introduction and Motivation

• Modern Programming Style Requires Sophisticated Dynamic Memory Management
• Object Oriented and Linked Data Structured Applications do not Exhibit Spatial Localities
• The Performance Gap Between CPU and Memory Led to New Memory Management techniques such as:
  - Prefetching and Jump Pointers
  - Memory Forwarding and High Performance Allocators
Introduction and Motivation (Cont’d)

- **Is It Worth Offloading All Memory Management functions to a Separate Processor**
  - A separate processing logic embedded in DRAM chip
  - A processor in the memory controller
  - A separate Processor in the CPU chip
  - Use a dedicated thread on a multithreaded architecture
  - Or caching the references made by memory management functions using separate cache (memory manager’s cache)
Introduction and Motivation (Cont’d)

• What are the advantages?
  - Improve performance *(maybe not)*
  - Eliminate Cache pollution caused by the memory management functions *(addressed in this work)*
  - If you have a designated processor for memory management functions, you can implement more intelligent techniques such as:
    • Jump Pointers
    • Memory Relocation
    • Profile memory usage of an application
• In This Paper We Have Two Specific Goals
  - How many cache misses will be eliminated if memory management functions (allocations and de-allocations) are offloaded to a separate processor
  - How do various memory allocators perform in terms of cache misses they cause
Outline

• Introduction and Motivation

• Memory Management Functions
  A cause for cache pollution

• Separate Caches With/Without a Separate Processing Engine
  Eliminate the cache pollution caused by Memory Management Functions

• Framework and Empirical Results

• Conclusions
Memory Management Functions in CPU

```
main()

mmf()

load a;
load c;
mmf();

load k;
load m;
mmf();

load a;
load c;
mmf();

load k;
load m;
mmf();
```

Instruction cache:
- `main_i1`
- `main_i2`
- `main_i3`
- `...`

Program counter:
- `main_i1`
- `main_i2`
- `main_i3`
- `...`

Data cache:
- `a, b`

Memory:
- Global
- `a, b, c, d, ...
- `k, l, m, n, ...
- `...`

Data:
- `...`

Heap:
- `...`

Stack:
- `...`
Memory Management Functions in CPU

main()

mmf()

{ load a;
  load c;
  mmf();
}

{ load k;
  load m;
}

main()

mmf()

{ load a;
  load c;
  mmf();
}

mmf()

{ load k;
  load m;
}

main()

mmf()

{ load a;
  load c;
  mmf();
}

mmf()

{ load k;
  load m;
}

program counter

instruction cache

data cache

memory

data

heap

stack

global

a, b, c, d,

k, l, m, n,

main_ra

$main_fp

$main_arg0

main_i1

main_i2

main_i3

mmf_i1

mmf_i2

mmf_i3

a, b

k, l

c, d
Memory Management Functions in CPU

This image illustrates the memory management functions in a CPU. The code snippet is as follows:

```c
main()
{
    mmf();
}

mmf()
{
    ...
}
```

The diagram shows the memory management with components such as the instruction cache, program counter, instruction cache, and memory. The variables `k, l, m, n, ...` and `a, b, c, d, ...` are also depicted.
Separate Caches With/Without Processor
## Framework

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>Description</th>
<th>Inputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>boxed-sim</td>
<td>balls in box simulator</td>
<td>-n 10 –s 1</td>
</tr>
<tr>
<td>Cfrac</td>
<td>it factors numbers.</td>
<td>a 36-digit number</td>
</tr>
<tr>
<td>Ptc</td>
<td>Pascal to C converter</td>
<td>mf.p</td>
</tr>
<tr>
<td>espresso</td>
<td>PLA optimizer</td>
<td>largest.espresso</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Allocator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABT</td>
<td>Address Order Binary Tree</td>
</tr>
<tr>
<td>Doug Lea</td>
<td>A Hybrid Allocator used in gnu compilers</td>
</tr>
</tbody>
</table>
Framework (Cont’d)
Empirical Results

Total number of cache misses for benchmarks with different allocators (8 Kbytes direct mapped cache with 32 bytes blocks)
Empirical Results (Cont’d)

Cache Miss Rates – 8 Kbyte Cache with 32 Bytes cache line size
Empirical Results (Cont’d)

Cache Miss Rates – 8 Kbyte Cache with 64 Bytes cache line size
Empirical Results (Cont’d)

Cache Miss Rates – 16 Kbyte Cache with 32 Bytes cache line size
Empirical Results (Cont’d)

Memory Management Cache Misses
1 Kbyte Cache with 32 Bytes cache line size

<table>
<thead>
<tr>
<th></th>
<th>boxed</th>
<th>cfraction</th>
<th>espresso</th>
<th>ptc</th>
<th>average</th>
</tr>
</thead>
<tbody>
<tr>
<td>abt</td>
<td>334</td>
<td>353</td>
<td>353</td>
<td>353</td>
<td>348.25</td>
</tr>
<tr>
<td>lea</td>
<td>339</td>
<td>358</td>
<td>358</td>
<td>358</td>
<td>353.25</td>
</tr>
</tbody>
</table>
Empirical Results (Cont’d)

Memory Management Cache Miss Rates
1 Kbyte Cache with 32 Bytes cache line size
Conclusions

• Object Oriented and Linked Data Structured Applications Exhibit Poor Locality
  
  Cache pollution caused by Memory Management functions

• Utilizing Separate Caches to Cache the References of Memory Management Functions
  
  On average, about 40% of cache misses eliminated

• A Small Cache, 1 Kbytes or even 512 Bytes, is large enough as Memory Management cache
Backups
New Memory Management Algorithms

Address Ordered Binary Tree (ABT)

- Head
  - Size:
    - 50
    - 65
  - Address:
    - 300
    - 95

Available chunk

- Size:
  - 65
  - 95
- Address:
  - 110
  - 0

- 20 malloc(70 bytes)
- 21 free(object x)
- Size 25 and address 175

24 bytes Memory overhead
Address Ordered Binary Tree

- head
- size
  - maximum size (left subtree)
  - 50 300
  - 65 95
- available chunk
- address
  - maximum size (right subtree)
  - 95 700
- 20 malloc(70 bytes)
- 65 110
  - 0 0
- 0 0
- 0 0
Address Ordered Binary Tree

head

<table>
<thead>
<tr>
<th>Size</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>300</td>
</tr>
<tr>
<td>65</td>
<td>25</td>
</tr>
</tbody>
</table>

maximum size (left subtree)

available chunk

65 110
0 0

25 700
0 0

maximum size (right subtree)

70 725
0 0

20 malloc
(70 bytes)

request is responded
Address Ordered Binary Tree

- **head**
- **size maximum size (left subtree)**
- **size maximum size (right subtree)**
- **available chunk**
- **address free(object x)**

- $110 + 65 = 175$
Intelligent Memory Manager

System Bus

BIU

Processor

Instruction Cache

Data Cache

W/D Completion

R/A Ready

Data

Control

Interface

DRAM

eDRAM

25
Intelligent Memory Manager (Cont’d)