A Study of Memory Management for Web-based Applications on Multicore Processors

Hiroshi Inoue, Hideaki Komatsu and Toshio Nakatani
IBM Tokyo Research Laboratory
Our Goal

To develop an efficient memory management technique for Web-based applications to improve their performance on multicore processors.

Our main contributions include:

1. Showing that a good technique for a single core is not necessarily a good one on a multicore processor.
2. Proposing our new approach for multicore environments
Characteristics of Web-based applications

- In Web-based applications, most of the allocated memory blocks are *transaction scoped*, and live only during one transaction.

- We exploit their characteristics to reduce the costs of memory management.
An existing approach

- Region-based memory management
  - reduces the cost of memory management by discarding all blocks in a region at once (instead of freeing individual blocks)
  - is widely used (e.g. Apache pool allocator)

The region-based memory management looks ideal for managing transaction-scoped memory blocks efficiently
Experimental setup with the PHP runtime

Software stack of the Zend’s PHP runtime

- PHP Application
- Zend’s PHP Runtime
- Garbage Collector
  - for transaction-scoped allocations
- custom memory allocator
  - for persistent allocations
- Standard C Library
- Operating System
- hardware
Experimental setup with the PHP runtime

We replaced only the custom memory allocator of the PHP runtime.

We did not modify:
- PHP applications
- garbage collector
- memory allocator in libc

Software stack of the Zend’s PHP runtime

- PHP Application
  - Garbage Collector
    - for transaction-scoped allocations
    - region-based allocator (replacing default allocator)
  - for persistent allocations
- Standard C Library
- Operating System
- hardware
Experimental setup with the PHP runtime

System used
- Blade Center HS21
- 2x quad-core Xeon (Clovertown) 1.86 GHz
- 8 GB system memory
- Red Hat Enterprise Linux 5

Software
- PHP-5.2.1 (w/ APC-3.0.14)
- lighttpd-1.4.13
- mysql-4.1.20

Benchmark
- six server applications (ezPublish, MediaWiki, SugarCRM, phpBB, CakePHP, SPECweb2005)
Performance of the region-based allocator: Throughput

on one core

8% speedup
Performance of the region-based allocator:

**Throughput**

- **on one core**
- **on eight cores**

- **8% speedup**
- **13% slowdown!**
Performance of the region-based allocator:
Scalability

The region-based allocator was 12% **faster** on one core
The region-based allocator was 23% **slower** on eight cores

![Graph showing relative throughput vs. number of cores for region-based allocator and default allocator of PHP runtime.](graph)

For ezPublish
Performance of the region-based allocator:
Execution time breakdown

- 10x speedup in memory management
- 40% slowdown in other parts

For ezPublish
Performance of the region-based allocator:

Cache misses and bus traffic

![Graph showing performance metrics for different applications and memory events.](image)

- **eZ Publish**
- **MediaWiki (read only)**
- **MediaWiki (read/write)**
- **SugarCRM**
- **phpBB**
- **CakePHP**
- **SPECweb2005**
- **Geo. MEAN**

- L1I miss
- L1D miss
- DTLB miss
- L2 miss
- Bus traffic

**Change in number of events on eight cores**

- **115%** more cache misses than default allocator
- **103%** fewer cache misses than default allocator (better)
Performance of the region-based allocator:

Memory latency

![Bar chart showing average L1 cache miss latency for different numbers of cores.]

- Default allocator of PHP runtime
- Region-based allocator

average memory latency increased due to bus contention

average L1 cache miss latency: \((\frac{L1D\_PEND\_MISS}{L1D\_REPL})\)

- on one or few cores: reduced cost of memory management > increased bus traffic
- on more cores: reduced cost of memory management < increased bus traffic
Our Goal

✓ to reduce the cost of memory management
✓ without slowing down on multicore processors

![Bar chart showing relative execution time]

- default allocator of the PHP runtime
- region-based allocator
- Our Goal

- memory management
- others
Revisiting general-purpose memory allocators

- General-purpose memory allocators’ tasks
  - `malloc()`: allocate memory block
  - `free()`: reclaim memory block and prepare for reuse in future allocations
  - minimize the heap fragmentation (defragmentation)

For example,

- coalescing multiple small blocks into large blocks
- splitting large blocks into small blocks
- sorting unused blocks in the free lists

*typically consumes large amounts of CPU time*
Our approach: **Defrag-Dodging**

**Key observation**

*the transactions in Web-based applications are short enough to ignore heap fragmentation, and so the*

\[
\text{cost of defragmentation} > \text{benefits}
\]

- Our new approach: **Defrag-Dodging**
  - reduces the memory management cost by avoiding defragmentation activities in malloc and free
  - unlike the region-based memory management, support a free() function to enable fine-grained memory reuse
Comparing three approaches

<table>
<thead>
<tr>
<th></th>
<th>general-purpose memory management</th>
<th>our Defrag-Dodging</th>
<th>region-based memory management</th>
</tr>
</thead>
<tbody>
<tr>
<td>malloc() allocate new memory block</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
<tr>
<td>free() reclaim and reuse memory block</td>
<td>✔</td>
<td>✔</td>
<td>-</td>
</tr>
<tr>
<td>defragmentation</td>
<td>✔</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
# Comparing three approaches

<table>
<thead>
<tr>
<th></th>
<th>general-purpose memory management</th>
<th>our Defrag-Dodging</th>
<th>region-based memory management</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>malloc()</code> allocate new memory block</td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td><code>free()</code> reclaim and reuse memory block</td>
<td>✔️</td>
<td>✔️</td>
<td>-</td>
</tr>
<tr>
<td>defragmentation</td>
<td>✔️</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

- reduced memory management cost
  - simpler allocator code
  - simpler heap structure (e.g. no per-object metadata)
Comparing three approaches

<table>
<thead>
<tr>
<th></th>
<th>general-purpose memory management</th>
<th>our Defrag-Dodging</th>
<th>region-based memory management</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>malloc()</code></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td><code>allocate new memory block</code></td>
<td>✔️</td>
<td>✔️</td>
<td>✔️</td>
</tr>
<tr>
<td><code>free()</code></td>
<td>✔️</td>
<td>✔️</td>
<td>-</td>
</tr>
<tr>
<td><code>reclaim and reuse memory block</code></td>
<td>✔️</td>
<td>✔️</td>
<td>-</td>
</tr>
<tr>
<td>defragmentation</td>
<td>✔️</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

better scalability on multicore processors by avoiding increases in bus traffic
DDmalloc: Implementation of Defrag-Dodging

- based on a segregated heap allocator
  - maintains free lists for each size of blocks to keep track of freed blocks and reuse them in future allocations
- reduced cost by keeping malloc and free as simple as possible
  - for example, free() function only chains the freed blocks to the corresponding free list (and does nothing else!)
  - the allocator code is less than 500 lines of C code
- clears all metadata to refresh the heap at the end of each transaction
- see the paper for the implementation details
Experimental setup with the PHP runtime

Software stack of the Zend’s PHP runtime

<table>
<thead>
<tr>
<th>PHP Application</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zend’s PHP Runtime</td>
</tr>
<tr>
<td>Garbage Collector</td>
</tr>
<tr>
<td>custom memory allocator</td>
</tr>
<tr>
<td>Standard C Library</td>
</tr>
<tr>
<td>Operating System</td>
</tr>
</tbody>
</table>

for transaction-scoped allocations
for persistent allocations
Experimental setup with the PHP runtime

Again, we replaced only the custom memory allocator with DDmalloc.

We did not modify:
- PHP applications
- Garbage collector
- Memory allocator in libc

<table>
<thead>
<tr>
<th>Software stack of the Zend’s PHP runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PHP Application</strong></td>
</tr>
<tr>
<td><strong>Zend’s PHP Runtime</strong></td>
</tr>
<tr>
<td><strong>Garbage Collector</strong></td>
</tr>
<tr>
<td><strong>DDmalloc</strong> (replacing default allocator)</td>
</tr>
<tr>
<td><strong>Standard C Library</strong></td>
</tr>
<tr>
<td><strong>Operating System</strong></td>
</tr>
<tr>
<td><strong>hardware</strong></td>
</tr>
</tbody>
</table>
Performance of DDmalloc: Throughput

on one core

on eight cores

higher is faster

0.0 0.2 0.4 0.6 0.8 1.0 1.2

relative throughput

eZ Publish  MediaWiki (read only)  MediaWiki (read/write)  SugarCRM  phpBB  CakePHP  SPECweb  Geo. MEAN

default allocator of PHP runtime
region-based allocator
our DDmalloc
Performance of DDmalloc: Scalability

- our DDmalloc
- region-based allocator
- default allocator of PHP runtime

for ezPublish
Performance of DDmalloc:
Execution time breakdown

- 10x speedup
- 2x speedup in memory management
- no slowdown! (2% improved)

For ezPublish
Performance of DDmalloc:
Cache misses and bus traffic

- eZ Publish
- MediaWiki (read only)
- MediaWiki (read/write)
- SugarCRM
- phpBB
- CakePHP
- SPECweb2005
- Geo. MEAN

more cache misses than default allocator
fewer cache misses than default allocator (better)
on eight cores
Summary

- We studied the effects of memory management approaches on the performance of Web-based applications on multicore processors
  - region-based allocator: fast on a single core, but slow on multicore processors due to increased bus traffic
  - general-purpose allocator: not cost-effective in avoiding heap fragmentation
- We proposed the new approach of *Defrag-Dodging* to reduce memory management costs

More data on the paper
- evaluation on Niagara
- evaluation with Ruby runtime
- comparisons with TCmalloc, Hoard
- discussions on the GC-based languages