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

Hiroshi Inoue, Hideaki Komatsu, and Toshio Nakatani

IBM Tokyo Research Laboratory
1623-14, Shimo-tsuruma, Yamato-shi, Kanagawa-ken, 242-8502, Japan
{inouehrs, komatsu, nakatani}@jp.ibm.com

Abstract

More and more server workloads are becoming Web-based. In these Web-based workloads, most of the memory objects are used only during one transaction. We study the effect of the memory management approaches on the performance of such Web-based applications on two modern multicore processors. In particular, using six PHP applications, we compare a general-purpose allocator (the default allocator of the PHP runtime) and a region-based allocator, which can reduce the cost of memory management by not supporting per-object free. The region-based allocator achieves better performance for all workloads on one processor core due to its smaller memory management cost. However, when using eight cores, the region-based allocator suffers from hidden costs of increased bus traffics and the performance is reduced for many workloads by as much as 27.2% compared to the default allocator. This is because the memory bandwidth tends to become a bottleneck in systems with multicore processors.

We propose a new memory management approach, *defrag-dodging*, to maximize the performance of the Web-based workloads on multicore processors. In our approach, we reduce the memory management cost by avoiding defragmentation overhead in the malloc and free functions during a transaction. We found that the transactions in Web-based applications are short enough to ignore heap fragmentation, and hence the costs of the defragmentation activities in existing general-purpose allocators outweigh their benefits. By comparing our approach against the region-based approach, we show that a per-object free capability can reduce bus traffic and achieve higher performance on multicore processors. We demonstrate that our defrag-dodging approach improves the performance of all the evaluated applications on both processors by up to 11.4% and 51.5% over the default allocator and the region-based allocator, respectively.

Categories and Subject Descriptors D.3.3 [Programming Languages]: Language Constructs and Features – Dynamic storage management.

General Terms Performance, Languages.

Keywords Dynamic memory management, Region-based memory management, Scripting Language, Web-based applications.

1. Introduction

Emerging Web-based workloads are demanding much higher throughputs to support ever-increasing client requests. When a next-generation server system is designed, it is important to be able to run such Web-based applications efficiently on multicore processors. One important characteristic of those Web-based applications is that most memory objects allocated during a transaction are transaction scoped, living only during that transaction. Therefore the runtime systems can discard these objects at once when a transaction ends.

Region-based memory management [1-4] is a well-known technique to reduce the cost of memory management by discarding many objects at once. Typically, a region-based allocator does not provide a per-object free and hence does not reuse the memory area allocated for dead objects. Instead, it discards the whole region at once to reclaim all the objects allocated in that region. For example, the Apache HTTP server uses a region-based custom allocator [5] that deallocates all of the objects allocated to serve an HTTP connection when that connection terminates.

Although the region-based memory management looks ideal for managing transaction-scoped objects in Web-based applications, it does not improve, or in some cases even degrades, the performance of Web-based applications on systems with multicore processors compared to a general-purpose allocator. For example, our results showed that performance of Web-based applications written in PHP significantly degraded on a system with eight cores of Intel® Xeon® (Clovertown) processors, while it improved the performance of the same applications on one or few processor cores. Figure 1 shows an example of the degradation in throughput. Although the region-based allocator significantly

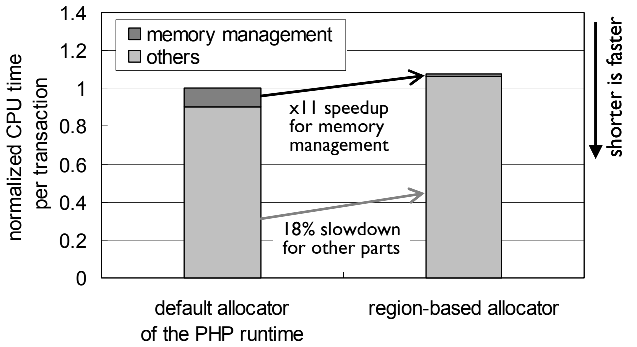


Figure 1. Performance of a region-based allocator on multi-core processors (for MediaWiki on 8 Xeon cores).

Table 1. Summary of three allocation approaches for transaction-scoped objects including our proposed defrag-dodging approach.

type of allocators	bulk free	per-object free	defrag mentation	cost of malloc / free	bandwidth requirement	examples of the approach
general purpose allocators supporting bulk freeing	Yes	Yes	Yes	high	low	default allocator of the PHP runtime [8], amalloc (arena malloc) in libc, Reaps [9]
region-based allocators	Yes	No	No	lowest	high	Apache pool allocator [5], GNU obstack [10]
our defrag-dodging allocator	**Yes**	**Yes**	**No**	**low**	**low**	our DDmalloc

speeds up the memory management functions, it degraded the performance of the rest of the program. This degradation is due to longer access latency for system memory caused by the increased bus traffics. The region-based allocator does not support per-object free and hence cannot reuse the memory locations of dead objects. Therefore many dead objects are left in cache memory and cache pressure is increased, resulting in increased cache misses and bus traffics. The system memory bandwidth tends to become a bottleneck in systems with multicore processors [6, 7], because the rate of improvement in processing power with such multicore processors exceeds the rate of improvement in system memory bandwidth. Hence the increase in bus traffic may limit the performance of the region-based allocator on multicore processors.

Based on our learning, we present a new memory management approach for transaction-scoped objects, called *defrag-dodging*. With this approach, we aim to find a sweet spot between the general-purpose memory management and the region-based memory management. As discussed above, the region-based memory management is not an optimal way to reduce the memory management cost on multicore processors.

Unlike the general-purpose memory allocators such as the default allocator of the PHP runtime, our approach reduces the memory management cost by skipping those defragmentation activities in malloc and free, which take a non-negligible CPU time. We found that the transactions of Web-based applications are short enough to ignore the harmful effects of gradual heap fragmentation even in large Web-based applications, and hence the cost of the defragmentation activities outweighs the benefit. For example, the default allocator of the PHP runtime supports both per-object and bulk freeing and it clean up the heap at the end of each transaction by bulk freeing. In spite of cleaning up the heap every transaction, the default allocator pays a cost for defragmentation activities in malloc and per-object free functions to avoid gradual performance degradation and the cost matters for the overall performance of the workloads.

Unlike the region-based memory management, our approach retains its per-object free capability even though all of the memory objects are freed at the end of a transaction. This per-object free capability, which enables fine-grained memory reuse, is important to avoid the performance degradation with multicore processors. Table 1 contrasts those three approaches. By avoiding defragmentation activities, we can use a very simple data structure for the heap and we do not have to add per-object header for each memory object in our new allocator. Such simpler data structure can give further improvements in the overall performance by reducing memory pressure in addition to the reduced allocator cost.

To evaluate our defrag-dodging approach, we implemented a new memory allocator, called *DDmalloc*, in the PHP runtime and compared the performance of three allocators, our DDmalloc, a region-based allocator, and the default allocator of the PHP runtime, using six Web-based applications. All three allocators support bulk freeing (with the function freeAll) for transaction-scoped objects at the end of each transaction. In addition to the freeAll, the default allocator and our DDmalloc support per-object free during a transaction. Our DDmalloc avoids defragmenting the heap in malloc and per-object free functions to reduce costs but still maintains a free list to keep track of freed objects and reuse them in future allocations. This configuration gives the highest performance on multicore processors. We conducted experiments on two systems, one with eight Intel Xeon cores and the other with eight Sun Niagara cores. By reducing the CPU time used for memory management our DDmalloc showed improvements in throughput by up to 11.1% on Xeon and 11.4% on Niagara compared to the default allocator of the PHP runtime using eight cores. Meanwhile, due to the increased bus traffic the region-based allocator did not improve the throughput when using eight cores.

To compare our DDmalloc against well known general-purpose allocators that do not support bulk freeing, we also examined the performance of the Ruby runtime for a Web-based application. We used Hoard [11] and TCmalloc [12] for the comparisons. The results showed that the cost of the defragmentation activities exceeds the benefit even in those sophisticated memory allocators. Our DDmalloc achieved higher performance than these high-performance allocators.

In this paper, we examined our defrag-dodging approach for transaction-scoped objects in Web-based applications, since multicore processors have already become so common for servers that efficient execution of Web-based server applications on multicore processors is important. However our defrag-dodging approach is applicable to other applications that deallocate many objects in groups and where the lifetimes of those objects are short enough to ignore gradual fragmentation. Although it has received little study to date, our results show that the limited bandwidth in multicore environments is another important problem for memory allocators, in addition to such problems as lock contention and false sharing.

The main contribution of this paper is two-fold. 1) We demonstrate that the advance of multicore processors is significantly changing the requirements for memory allocators. For example, our experiments show that region-based memory management may degrade the performance of applications on multicore environments, while it improves the performance on systems with only one or a few cores. 2) We propose and evaluate a new memory management approach for transaction-scoped objects, called *defrag-dodging*, based on our observation that the cost of the defragmentation activities outweighs the benefit in Web-based workloads. The defrag-dodging approach can achieve higher performance in Web-based applications on multicore processors by reducing the cost of memory management.

The rest of the paper is organized as follows. Section 2 gives an overview of existing memory management techniques. Section 3 describes our new defragment-dodging approach and our implementation of the new approach. Section 4 shows the experimental environment and gives a summary of our results. Section 5 discusses how our observations affect runtime systems of GC-based languages. Section 6 covers related work. Finally, Section 7 reports our conclusions.

2. Existing Memory Management Approaches

This section gives an overview of existing memory management approaches as a background of our proposed one.

2.1 Region-based memory management

Region-based memory management [1-4] is a well-known technique to reduce the cost of memory management using knowledge of the object lifetimes. The region-based allocators obtain a large block of memory from an underlying allocator, such as operating systems, and the allocation is done by simply incrementing a pointer that tracks the current location to allocate in the block. The programmer can delete all the allocated objects merely by discarding the entire block. Thus the costs of both allocation and deallocation are almost negligible. The region-based memory management, however, may suffer from performance degradation with Web-based applications on multicore processors due to increased bus traffic, as shown in Figure 1.

2.2 General-purpose malloc-free

General-purpose memory allocators have to perform many activities to avoid excess fragmentation of the free memory chunks in the heap. How to avoid fragmentation depends on the implementation of the memory allocator and is a major area of innovation. One typical approach is used in a well known memory allocator developed by Doug Lea [13], which sorts all of the objects in the free lists in order of their size to easily find the best object to allocate for a request, coalesces multiple small objects into large objects, and splits large objects into small objects in response to requests. The default allocator of the current PHP runtime developed by Zend Technologies [8] also does coalescing and splitting of objects. Memory allocators often spent a large amount of CPU time for such defragmentation activities, which are necessary for general-purpose allocators to avoid gradual performance degradations of the applications, both in execution time and memory consumption.

3. Our Defrag-dodging Approach

3.1 Basic concept

Our defrag-dodging approach eliminates the overhead of the costly defragmentation activities in malloc and free by introducing a freeAll function for initializing the heap. The freeAll function is called from an application when all of the objects in the heap can be deallocated, such as at the end of each transaction in Web-based applications. In the defrag-dodging approach, allocators still retain the per-object free capability. Our approach replaces the defragmentation activities in malloc and free by a freeAll that cleans up the heap including the metadata, such as the free lists of unallocated objects. Therefore applications need to call freeAll even if all of the objects in the heap have already been freed by per-object free. The defrag-dodging approach can reduce the overhead of memory management because the cost to initialize the metadata in the heap is much smaller than the cost of maintenance for the heap with many live objects. We can skip defragmentation activities in malloc and free because each transaction in most Web-based applications is short enough to ignore the gradual performance degradations due to the fragmentation of the heap.

Both our defrag-dodging approach and the region-based memory management can reduce the cost of memory management compared to general-purpose malloc-free. Comparing the defrag-dodging approach to the region-based memory management, the cost is slightly higher for the defrag-dodging approach, because it still needs to maintain the free lists to support per-object free semantics. This per-object free capability is the key not only to reduce the amount of memory used but also to improve the overall performance of the applications by avoiding increases in bus traffic. Without reusing the freed objects, many already-dead objects remain on the cache memory and hence waste the cache capacity.

As with the region-based memory management, the defrag-dodging approach requires modifications to applications written with general malloc-free semantics, because our defrag-dodging allocator depends on invocations of freeAll to avoid fragmentation in the heap. Also, it requires the allocations to be aware of the objects' lifetimes. For example, the current PHP runtime uses its custom memory allocator to allocate transaction-scoped objects and an allocator in the standard library to allocate persistent objects. The PHP runtime calls freeAll for the heap of its custom allocator at the end of the lifetime of transaction-scoped objects. Thus it is easy to apply the defrag-dodging approach or the region-based memory management to the PHP runtime by replacing only the memory allocator. The modification to apply the defrag-dodging approach for other applications written for general malloc-free semantics is similar to that for the region-based memory management [3]. The steps for modification are:

1) identify memory objects that can be discarded at once (such as transaction-scoped objects in the Web-based workloads),
2) replace malloc and (per-object) free for those identified objects with the allocation function for our allocator, and
3) add freeAll function calls in appropriate places (such as at the ends of transactions in the PHP runtime).

In the modification for the region-based management, all invocations of per-object free for the objects identified in Step 1 have to be removed in addition to these three steps, but those free calls must be retained for ours.

3.2 Implementation of DDmalloc

In this section, we describe the details of an implementation of our defrag-dodging allocator, which we call *DDmalloc*. DDmalloc is based on a segregated storage [14] similar to other high-performance allocators. We carefully implemented the malloc and free functions to be as efficient as possible by avoiding activities other than maintenance of the free lists. Though we describe the

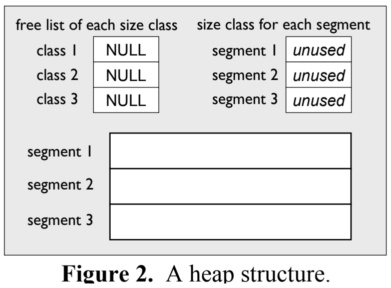


Figure 2. A heap structure.

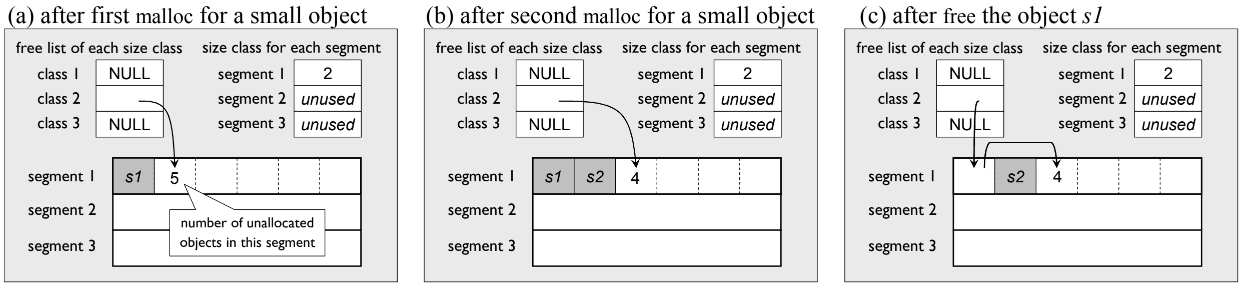

(a) after first malloc for a small object
(b) after second malloc for a small object
(c) after free the object s1
free list of each size class
size class for each segment
class 1
class 2
class 3
NULL
segment 1
segment 2
segment 3
2
unused
s1
s2
5
4
number of unallocated objects in this segment

Figure 3. Steps of malloc and free operations for small objects.

details of DDmalloc in this section, the method of free list managements in DDmalloc is not new. The novelty of our DDmalloc is that we totally eliminate, not just delay, the defragmentation activities. TCmalloc [12], for example, reduces the overhead by delaying the defragmentation activities until the total size of the memory objects in the free lists exceeds a threshold. However TCmalloc still has costs for the delayed defragmentation activities and the costs matter for the overall performance, as shown in Section 4.4.

Figure 2 shows the heap structure with DDmalloc. A heap consists of fixed-size memory chunks, each called a segment, and metadata. Each segment must start from an address of a multiple of the size of a segment. Due to this alignment restriction, DDmalloc can efficiently determine to which segment an object belongs from the address of the object. DDmalloc divides each segment into multiple objects of the same size and uses the segment as an array of those objects. There is no per-object metadata between objects. This results in both high space efficiency and better cache utilization. Like many other high-performance memory allocators, DDmalloc maintains a free list for each size-class. It maps all allocation requests into the corresponding size-class and allocates memory from the free list for the size-class. The metadata includes an array of pointers for the head of a single-linked free list for each size-class and an array of 1-byte integers that track the size-classes for each segment.

DDmalloc classifies objects into two categories, large objects (larger than half the size of a segment) and small objects. Figure 3 presents examples of malloc and free calls for a small object corresponding to the size-class 2. To handle the malloc request, DDmalloc first determines the size-class for the requested size. Then it checks a free list for that size-class. In this example, it finds that the free list is empty because it is the first invocation of malloc. DDmalloc generates new free objects for this size-class by obtaining an unused segment and dividing the segments into fixed-sized objects corresponding to the size-class. It returns the top of newly generated objects to the caller and uses a pointer to the second object as the head of the free list. The size-class is recorded in the metadata. In order to track the number of unallocated objects within the segments, DDmalloc stores the number of unallocated objects (5 in this example) at the top of the unallocated objects. Figure 3(a) shows a snapshot of the heap after the first call to malloc. Figure 3(b) shows an example of a subsequent call to malloc with the same size request. Now the free list for the size-class 2 is not empty and DDmalloc immediately returns the object at the top of the free list and updates the free list. Figure 3(c) illustrates an example of a call for free for the object allocated by the first malloc. DDmalloc chains the freed object to the top of the corresponding free list. Thus the freed objects are reused in LIFO order.

For large objects requests, DDmalloc directly allocates and reclaims the segments without using the free lists. It marks the segments as used for the large object in the size-class array when it allocates the segments. To free the large objects, it simply marks the segment as unused.

When freeAll is called, DDmalloc clears only the metadata in the heap. The metadata is much smaller than the entire heap. Hence the overhead of freeAll is almost negligible. As a result of the freeAll call, the heap returns to the initial state shown in Figure 2.

How to map the requested sizes of small objects onto each size-class is an important tunable parameter. Our current implementation 1) rounds up the requested size to a multiple of 8 bytes if the size is smaller than 128 bytes, 2) rounds up to a multiple of 32 bytes if the size is smaller than 512 bytes, and 3) rounds up to the nearest power of two for larger sizes. The size of a segment is another important parameter, which affects both the amount of memory consumed and the allocation speed. We used 32 KB as the size of a segment. We chose these parameters based on our measurements. For example, using larger segment size tended to increase memory footprint and cache misses while it reduced the number of instructions to manage each segment. We chose the parameters based on such tradeoffs to provide the best throughput in PHP applications

3.3 Optimizations

In addition to the basic memory management mechanisms, we tuned our implementation for the following traits:

1. DDmalloc frequently accesses the metadata in the heap. Thus accesses to the metadata may often incur cache misses due to associativity overflows if they are located at the same location in the heap. We change the position of the metadata in the heaps using the process ids to cut down on cache misses. The effect of this optimization is significant on Niagara where multiple hardware threads share a small L1 cache.
2. DDmalloc uses large page memory for the heap to reduce the overhead of TLB handling. Using larger size page for the heap results in notable performance improvements on some processors, such as Niagara, because of the high overhead of TLB handling in software. However we disabled this large-page optimization in our evaluations on Xeon to allow fair comparisons to the default allocator of the PHP runtime.
3. The default configuration of the current PHP runtime is a single-threaded application. However it can be configured as a multi-threaded application to run as plug-ins for multi-threaded HTTP servers. Even in the multi-threaded configuration, the threads serve independent transactions and they do not communicate with each other. Thus DDmalloc provides a separate heap for each thread and does not need locks for each heap. For our evaluations, we configured the PHP runtime as a single-threaded application and hence each runtime process owns only one heap.

4. Experimental Results

This section focuses on the performance comparisons of the Web-based applications written in PHP using the default memory allocator of the PHP runtime, a region-based allocator, and our DDmalloc for managing transaction-scoped objects.

4.1 Platforms and implementations

We implemented DDmalloc and a region-based allocator for two platforms, Intel's quad-core Xeon (Clovertown) processors running Linux and Sun's UltraSPARC T1 (Niagara) processors running Solaris. The Xeon system used for our evaluation was equipped with two 1.86 GHz quad-core Xeon E5320 processors for a total of eight cores and 8 GB of system memory. Red Hat Enterprise Linux 5 was running on the system. The PHP runtime was compiled as a 64-bit binary with gcc-4.1.2. The Niagara system used for our evaluation was equipped with a 1.2 GHz eight-core Niagara processor and 16 GB of system memory. Solaris 10 was running on the system. The PHP runtime was compiled with the C compiler included in Sun Studio 11. None of the experiments used the large pages on Xeon, but they did use the 4-MB pages on Niagara. This was because Linux could not use the large pages without modifying the applications, while Solaris supports them. Although both systems have a total of eight cores, the Niagara system provides 32 hardware threads with 4-way multi-threading for each core, while the Xeon system provides only 8 hardware threads.

We selected these two platforms to study the performance of the allocators on multicore processors with different focuses. The Xeon processor focuses on fast single-thread performance. Thus the processor has a higher frequency, larger cache memories, a hardware memory prefetcher, and out-of-order cores. In contrast, the Niagara processor focuses on total throughput by aggregating many simple cores in one chip. Thus the processor has a lower frequency, smaller cache memories, no hardware memory prefetcher, and in-order cores with 4-way multi-threading.

To modify the memory allocator for transaction-scoped objects in the PHP runtime, we simply replaced the custom memory allocator of the PHP runtime with our DDmalloc or the region-based allocator. We did not need to modify other parts of the runtime for our experiment. Also, we did not need to modify the PHP applications.

Our region-based allocator obtains a 256 MB chunk of memory from the operating system at startup time and allocates memory objects from the top of the chunk by simply incrementing a pointer showing the next position to allocate. It rounds up the requested size to a multiple of 8 bytes when it allocates an object. When the pointer reaches the end of the chunk, the allocator obtains the next 256 MB chunk. One 256 MB chunk was large enough for most of the PHP transactions and additional chunks were rarely needed, so the overhead of system calls to obtain additional chunks was negligible here. We also evaluated the GNU obstack [10] as another region-based allocator. However our own region-based allocator outperformed the obstack for the PHP applications. Therefore we used only our own region-based allocator in this paper.

4.2 Workloads and configurations

We used PHP-5.2.1 for the runtime, lighttpd-1.4.13 for the HTTP server, and mysql-4.1.20 for the database server. We also installed the APC-3.0.14 extension to the PHP runtime, which enhances the performance of PHP applications by caching the compiled intermediate code. Figure 4 is an overview of the system configuration for the measurements. The database server and the client emulator ran on separate machines. The number of PHP runtime processes was 16 for Xeon and 48 for Niagara. We selected a variety of Web applications written in PHP and evaluated their performances. Table 2 summarizes these workloads.

MediaWiki [15] is a wiki server program developed by the WikiMedia foundation for the Wikipedia online encyclopedia. We imported 1,000 articles from a Wikipedia database dump (http://download.wikimedia.org/). We configured MediaWiki to use memcached-1.2.1 running on the same machine to cache the results of the database queries. We measured two user scenarios with the MediaWiki: a read-only scenario and a read-write scenario. For the read-only scenario, we measured the throughput for reading randomly selected articles. In the read-write scenario, the clients opened randomly selected articles for editing and then updated the article instead of just reading it in 20% of the total transactions. The emulated clients introduced three seconds of think time between each request. The number of emulated clients was set for the highest throughput that allowed the average response time to stay under 2.0 seconds.

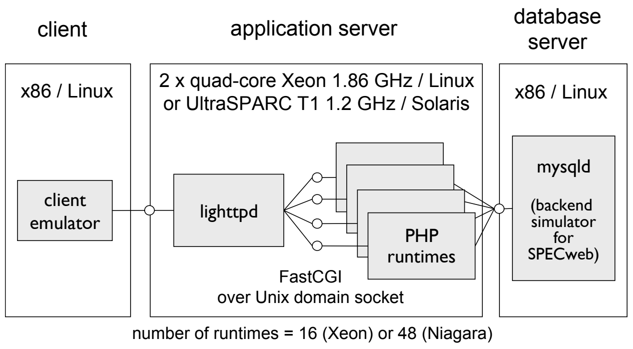

client
application server
database server
x86 / Linux
2 x quad-core Xeon 1.86 GHz / Linux or UltraSPARC T1 1.2 GHz / Solaris
x86 / Linux
client emulator
lighttpd
PHP runtimes
mysqld
(backend simulator for SPECweb)
FastCGI over Unix domain socket
number of runtimes = 16 (Xeon) or 48 (Niagara)

Figure 4. System configuration for the measurements.

Table 2. Workloads used in our measurements

workload	version	descriptions of the workload
MediaWiki	1.9.3	A wiki server program
SugarCRM	4.5.1	A customer relationship management system
eZ Publish	4.0.0	A content management system
phpBB	3.0.1	A Web-based forum program
CakePHP	1.2.0.7296	A Web application framework
SPECweb 2005	1.10	An industry-standard benchmark for Web servers with dynamic pages

Table 3. Statistics on average number of malloc and free calls per transaction and average size of memory allocation per malloc

workload	numbers of calls per transaction			allocation size (byte)
	malloc	free	realloc	
MediaWiki (read only)	151770	129141	6147	62.1
MediaWiki (read/write)	404983	354775	22371	66.7
SugarCRM	276853	225800	3120	49.3
eZ Publish	123019	109856	4646	78.6
phpBB	46965	43267	1003	56.3
CakePHP	99195	82645	3574	68.6
SPECweb	3277	2383	106	175.6

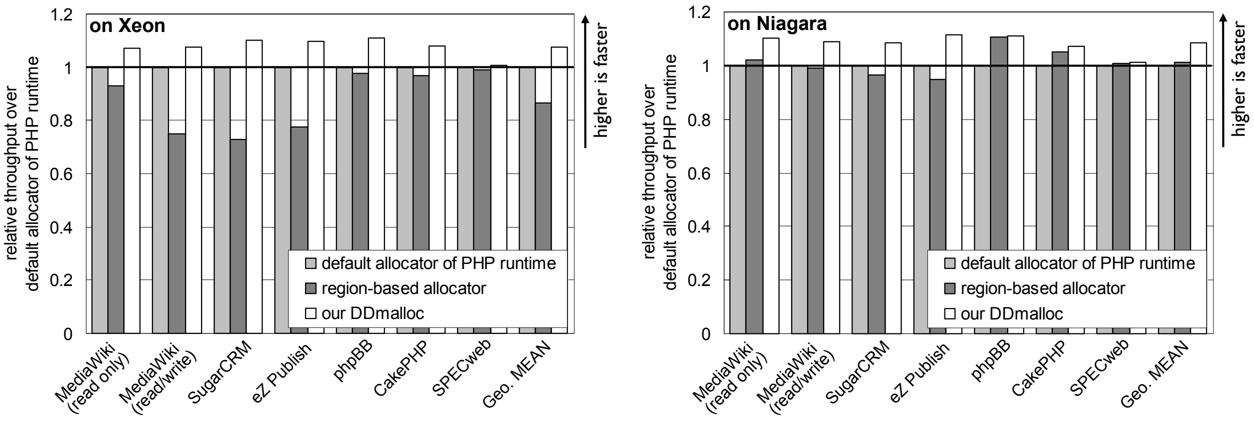


Figure 5. Relative throughput over the default allocator of the PHP runtime on 8 cores of Xeon and Niagara.

For SugarCRM [16], we set up a database with 512 user accounts and 10 customers for each user. Each emulated client logged into SugarCRM using a unique user-id and then repeatedly issued AJAX-style requests to obtain information on a customer assigned to that user.

For eZ Publish [17], we built a website using the setup wizard included in the distribution. We imported 1,000 articles from an actual blog server as blog posts into the website. Each emulated client kept its own session and read randomly selected articles.

The phpBB [18] is a popular Internet forum software package. We generated 1,000 posts in a database and measured the throughput to read randomly selected articles.

CakePHP [19] is a framework for developing Web applications. We built a simple telephone-directory application on top of the framework to focus on the performance of the framework. We generated 100 records in the database and measured the throughput for the following scenario: reading a table of all of the records, selecting one record randomly, and then updating that record.

SPECweb2005 [20] is an industry-standard benchmark for Web servers with dynamic pages. We selected the eCommerce scenario of the benchmark, because the other two scenarios emphasize SSL performance (in the Banking scenario) or large file transfers (in the Support scenario) and thus were not suitable for evaluating the performance of the PHP runtime itself. Even in the eCommerce scenario, the PHP program is much simpler than the other test applications and large amount of CPU time were consumed in static file serving.

Table 3 shows the average sizes of memory allocations per malloc call and average numbers of calls for malloc, per-object free, and realloc per transaction for each workload. We include the number of calls for calloc, a variation of malloc function, in the number of malloc calls. The number of free calls is 7.9% to 27.3% (15.3% on average) less than that of malloc. This means some allocated memory objects are not deallocated by per-object free and remain alive until the end of the transaction. Those objects are freed by the freeAll at the end of the transaction. More than 80% of the total objects are deallocated by per-object free and thus the performance of the per-object free is still very important, even with the runtime calls to freeAll at the ends of the transactions. The statistics for SPECweb2005 are slightly different from the other applications. The number of malloc and free calls are significantly smaller than in other workloads and the average size of memory allocation is larger than any other workload. As a result, the performance of SPECweb2005 was not sensitive to the memory allocator, as shown later.

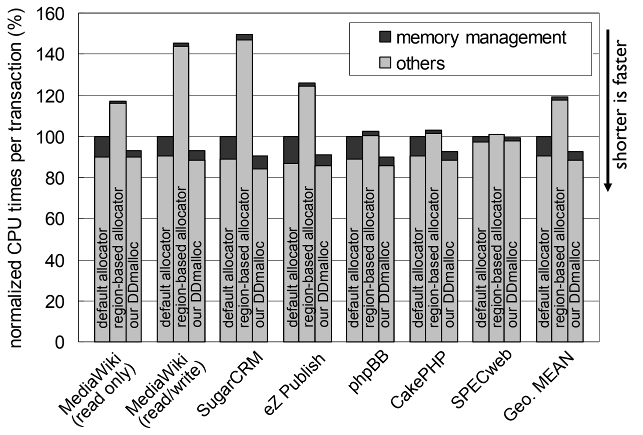


Figure 6. Breakdown of CPU time per transaction on 8 Xeon cores.

4.3 Performance results

Figure 5 shows comparisons of the throughput of the applications on Xeon and Niagara. Our DDmalloc had the best performance for all applications on both platforms. The maximum performance advantages over the default allocator of the PHP runtime were 11.1% (7.7% on average) on Xeon and 11.4% (8.3% on average) on Niagara. When we enabled the optimization using large pages on Xeon, the improvement increased to 11.7% (9.0% on average). In contrast, the region-based allocator significantly degraded the average performance by as much as 27.2% on Xeon. On Niagara, the region-based allocator improved the performance of four workloads, while degrading the other three. As a result, the region-based allocator average almost matched the default allocator. The maximum performance advantages of our DDmalloc over the region-based memory allocator were up to 51.5% (24.2% on average) on Xeon and 17.4% (7.0% on average) on Niagara.

To examine the differences due to the memory allocators, Figure 6 compares breakdowns of the system-wide CPU time per transaction as measured on Xeon. In the figure, memory operations include only those for transaction-scoped objects in the PHP runtime and do not include other memory operations, such as memory management functions in the operating system. We used a sampling profiler (Oprofile [21]), which uses a hardware performance monitor, and did not insert instrumentation code in the memory management functions to obtain the execution time of each call to the allocator. As discussed in Section 1, the region-based allocator reduced the CPU time used for memory opera-

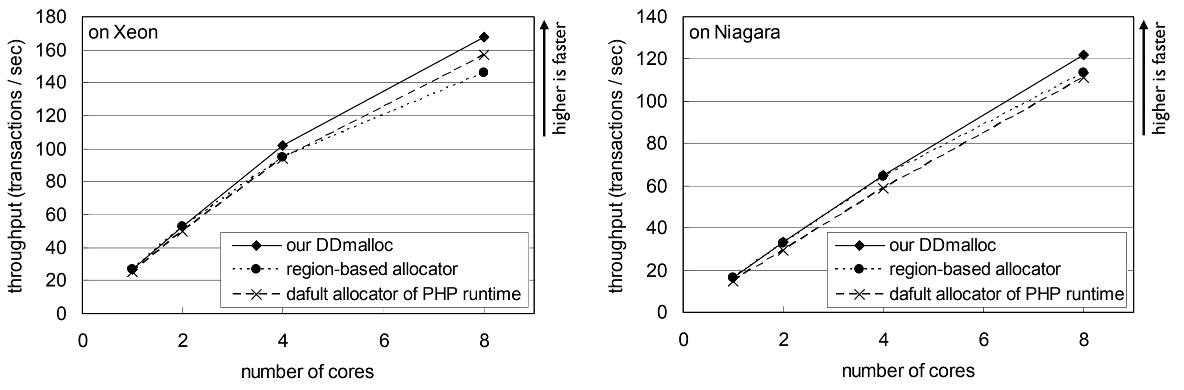


Figure 7. Throughput of MediaWiki (read-only scenario) with increasing numbers of cores on Xeon and Niagara.

Table 4. Speedups with 8 cores for each workload

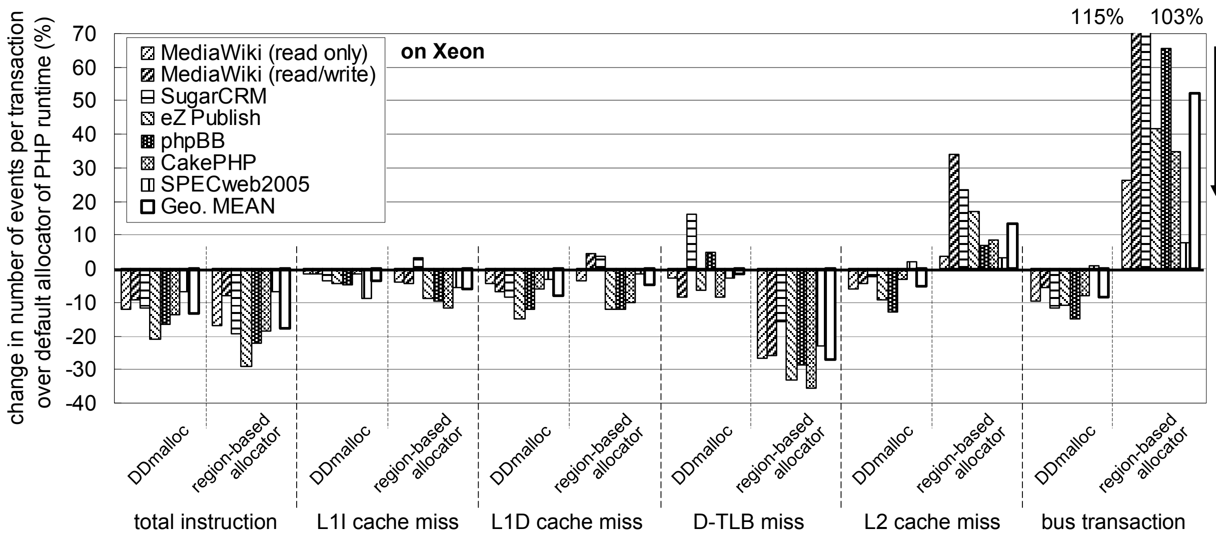
workload	**memory allocator**	**on Xeon**			**on Niagara**		
		throughput with 1 core (transactions/sec)	**throughput with 8 cores** (transactions/sec)	**speedup with 8 cores**	**throughput with 1 core** (transactions/sec)	**throughput with 8 cores** (transactions/sec)	**speedup with 8 cores**
MediaWiki (read only)	default	25.3	156.6	6.2x	14.9	111.0	7.5x
	region-based	26.4 (+4.0%)	145.7 (-7.0%)	5.5x	16.5 (+11.3%)	113.3 (+2.1%)	6.9x
	DDmalloc	26.4 (+4.1%)	167.9 (+7.2%)	6.4x	16.5 (+11.3%)	122.2 (+10.1%)	7.4x
MediaWiki (read/write)	default	11.7	79.6	6.8x	5.2	40.0	7.7x
	region-based	12.5 (+6.6%)	59.7 (-24.9%)	4.8x	5.5 (+5.4%)	39.6 (-1.1%)	7.2x
	DDmalloc	12.7 (+7.9%)	85.5 (+7.4%)	6.8x	5.6 (+7.0%)	43.5 (+8.8%)	7.8x
SugarCRM	default	19.4	134.6	6.9x	8.1	64.4	7.9x
	region-based	20.8 (+7.2%)	98.0 (-27.2%)	4.7x	9.2 (+13.4%)	62.3 (-3.4%)	6.8x
	DDmalloc	21.1 (+8.9%)	148.4 (+10.2%)	7.0x	8.8 (+8.3%)	69.7 (+8.3%)	7.9x
eZ Publish	default	28.5	178.6	6.3x	13.6	99.4	7.3x
	region-based	31.8 (+11.6%)	138.3 (-22.6%)	4.3x	16.5 (+21.1%)	94.4 (-5.1%)	5.7x
	DDmalloc	32.2 (+12.9%)	196.3 (+9.9%)	6.1x	15.8 (+15.9%)	110.8 (+11.4%)	7.0x
phpBB	default	62.6	402.4	6.4x	30.5	234.0	7.7x
	region-based	69.2 (+10.6%)	393.5 (-2.2%)	5.7x	35.9 (+17.7%)	259.1 (+10.8%)	7.2x
	DDmalloc	69.5 (+11.0%)	447.2 (+11.1%)	6.4x	34.0 (+11.2%)	259.8 (+11.0%)	7.7x
CakePHP	default	28.3	191.6	6.8x	12.6	96.7	7.7x
	region-based	31.6 (+11.4%)	185.7 (-3.1%)	5.9x	13.8 (+9.3%)	101.6 (+5.1%)	7.4x
	DDmalloc	30.8 (+8.8%)	206.6 (+7.8%)	6.7x	13.6 (+7.7%)	103.8 (+7.3%)	7.7x
SPECweb 2005	default	188.6	970.0	5.1x	115.5	699.3	6.1x
	region-based	197.3 (+4.6%)	960.4 (-1.0%)	4.9x	118.3 (+2.4%)	705.4 (+0.9%)	6.0x
	DDmalloc	194.3 (+3.0%)	977.3 (+0.8%)	5.0x	118.4 (+2.5%)	709.2 (+1.4%)	6.0x

✓ The ratios in parenthesis show the relative throughputs over the default allocator.

tions by 85% on average compared to the default allocator. Other parts of the programs, however, were slowed down. Our DDmalloc also reduced the overhead for memory management functions by 56% on average and up to 65%. The performance of other parts was unchanged or slightly improved. Comparing the breakdown for all workloads, SPECweb2005 consumed the least time for memory management. As a result, the effect of memory allocators was also least significant.

To see the effects of memory allocators on the scalability with increasing number of cores, Figure 7 compares the throughput of MediaWiki for the read-only scenario with various numbers of cores. DDmalloc consistently outperformed the default allocator. It almost tied the region-based allocator for small numbers of cores (up to 2 cores on Xeon and 4 cores on Niagara). However, DDmalloc demonstrated much better scalability with increased numbers of cores compared to the region-based allocator, and hence it achieved the best performance among the three allocators when using eight cores on both platforms, as shown in Figure 5. Table 4 summarizes the speedups using eight cores for all workloads. Both DDmalloc and the region-based allocator improved the performance of every workload when using only one core on both platforms. However the region-based allocator showed much poorer scalability compared to the other two allocators while DDmalloc and the default allocator of the PHP runtime achieved almost comparable speedups for all workloads.

Figures 8 shows the changes in the numbers of instructions, cache misses, and bus transactions per Web transaction from the default allocator. On both platforms, DDmalloc reduced both L1 and L2 cache misses and bus transactions while the region-based allocator significantly increased the numbers of L2 cache misses and bus transactions. The reduction in instructions and L1 instruction cache misses for DDmalloc and the region-based allocator were because of the smaller size of the allocator code. The reduction in L1 data cache misses was mainly because they do not use



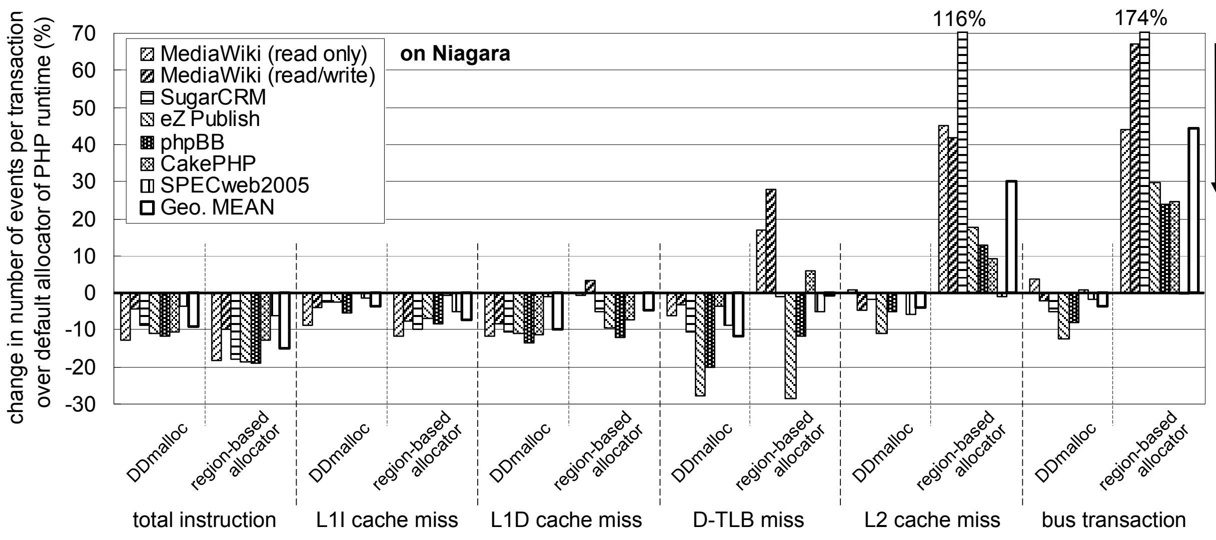


Figure 8. Comparisons of changes in the numbers of instructions, cache misses, and bus transactions per web transaction with our DDmalloc and the region-based allocator on 8 cores of Xeon and Niagara.

the per-object metadata that increases the size of the objects and reduces the cache locality. Both DDmalloc and the region-based allocator increased the D-TLB misses for two or three workloads and reduced them for the others. The number of TLB misses was sensitive to the allocator and its parameters, such as the size of a segment in DDmalloc, because the number of TLB entries is very small. For DDmalloc, we chose the parameter values that produced the highest throughput rather than the fewest TLB or cache misses. If we enable the large page optimization on Xeon, the TLB misses were reduced by more than 60% compared to the default allocator. The increase in L2 cache misses and subsequent bus transactions for the region-based allocator was because it does not support per-object free and thus cannot reuse the dead objects residing in the L2 cache. The increased cache pressure did not significantly increase the L1 data cache misses, because only the most frequently used data objects were residing in the small L1 cache and the dead objects were quickly spilled out of it. On Xeon, the increases in bus transactions were much larger than the increases in the L2 cache misses. This difference mainly came from the hardware memory prefetcher. We observed that the difference was reduced by disabling the prefetcher. The inferior scalability of the region-based allocator was unaffected, even without the prefetcher.

These cache miss trends were consistent when we changed the numbers of cores used. Our DDmalloc achieved better performance with the PHP runtime because of these improvements in the cache misses in addition to the smaller memory management overhead. In contrast, the region-based allocator was slowed down by the increases in bus transactions when using large number of cores as shown in Figure 7 and Table 4. Those performance degradations for the region-based allocator were less significant on Niagara because Niagara provides relatively higher memory bandwidth than Xeon.

Figure 9 shows the average memory consumptions during transactions. We defined memory consumption for each allocator as follows: the amount of memory allocated from the underlying memory allocator for the default allocator, the total amount of memory used for allocated segments and the metadata for DDmalloc, and the total amount of memory allocated during a transaction for the region-based allocator. DDmalloc consumed

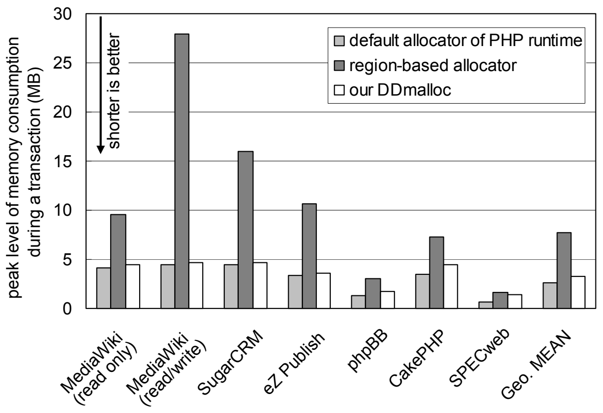


Figure 9. Comparison of the amount of memory consumed.

24% more memory on average compared to the default allocator. This was because the segregated heap management, which is the basis of DDmalloc, tends to consume more memory area as a trade-off for faster allocation and deallocation. Using a smaller size for segments can reduce the memory consumptions while it often increases the CPU time used for malloc and free. The region-based allocator consumed almost 3 times as much memory on average and more than 7 times more in the worst case.

4.4 Performance comparisons with high-performance general-purpose allocators

This section compares the performance of our DDmalloc against the well known general-purpose allocators, hoard-3.7 [11] and TCmalloc [12] included in the google-perf-tools-0.9.1. Those general-purpose allocators support only the malloc-free interface, and thus they are not applicable to the PHP runtime. Therefore we used the Ruby runtime for this comparison. The Ruby runtime does not call freeAll at the end of each Web transaction and does not distinguish between the allocations for transaction-scoped objects and other objects. To compare the performance of DDmalloc to the general-purpose allocators, we did not modify the runtime to call freeAll. Instead, we configured the runtime to restart periodically (once per 500 transactions) to clean up the entire heap without calling freeAll. This allows direct comparisons with these general-purpose allocators supporting only malloc and free, though this is not the best way to clean up the heap for DDmalloc because restarting process incurs more overhead than the freeAll function and once per 500 transactions is not short enough to totally ignore the gradual degradations due to heap fragmentation. Restarting the processes of scripting language runtimes is a common practice to avoid performance degradations even with standard memory allocators. Thereby we configured the runtime to restart every 500 transactions for all of the allocators because it was beneficial for all of the allocators.

We selected Ruby on Rails-1.2.3 [22] as a workload for the evaluation. Ruby on Rails is a framework for developing Web applications. We followed the evaluation for CakePHP, building a similar application on top of Ruby on Rails and evaluating it using the same scenario. We used the prebuilt binary of the ruby-1.8.5 included in the OS distribution and dynamically linked each memory allocator at runtime. The other server configurations were unchanged from the measurements for PHP. Figure 10 shows the comparisons for the throughput with various memory allocators on all eight cores of Xeon. Our DDmalloc again achieved the best performance. The performance advantage over the default allocator, glibc-2.5, was 13.6%. Hoard and TCmalloc also outperformed glibc. DDmalloc outperformed the next best allocator, TCmalloc, by 5.3%.

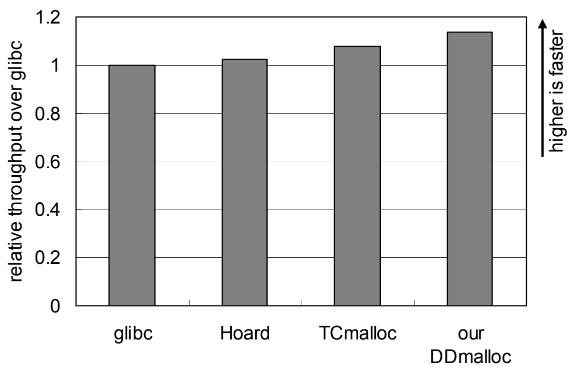


Figure 10. Relative throughput of Ruby on Rails over glibc-2.5 on 8 Xeon cores.

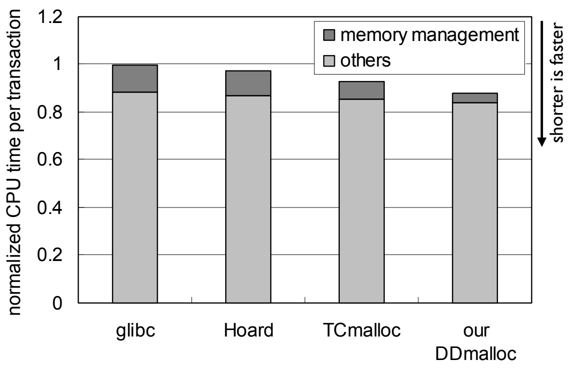


Figure 11. Breakdown of CPU time per transaction for Ruby on Rails on 8 Xeon cores with various allocators.

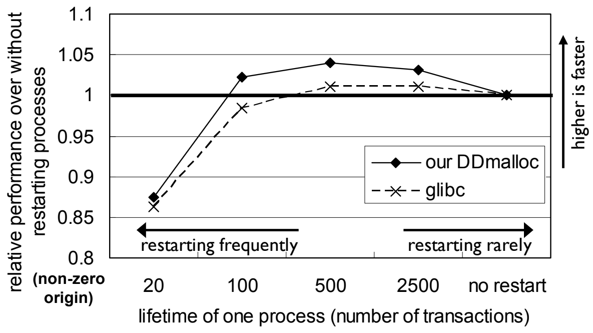


Figure 12. Performance improvements by restarting Ruby processes for various periods of restart on 8 Xeon cores.

Figure 11 compares the breakdowns of CPU cycles per transaction. All of the results are normalized against the results for glibc, the default allocator for the platform. DDmalloc obviously spent the least time on memory operations among the tested allocators by avoiding the costs for defragmentation activities in malloc and free. The results show that the costs of the defragmentation activities exceed the benefits in Web-based applications even in those sophisticated memory allocators, and the costs matter for the overall performance of the workloads.

In our DDmalloc, objects in the free lists are chained in random order after a long run without defragmentation activities. Hence DDmalloc starts to allocating separated objects for successive allocation requests. Such allocations tend to decrease the cache locality. To see this degradation more quantitatively, Figure 12 depicts the performance improvements from restarting the processes of the Ruby runtime to clean up the heap. From this figure, restarting a process after every 500 transactions improved throughput by 4.0% compared to no restarts for our DDmalloc while the improvement was only 1.1% for glibc. We observed that restarting a process after every 500 transactions reduced the number of L2 cache misses per transaction by 8.0% for DDmalloc while it was 2.5% for glibc. The reductions in the L1 cache misses by restarting the process were only 0.3% and 0.8% for DDmalloc and glibc respectively.

These results show that, even without restarting, DDmalloc was almost as fast as the TCmalloc and still faster than glibc or Hoard. This was because the benefit of efficient malloc and free of DDmalloc due to no defragmentation outweighed the cost of increased L2 cache misses due to fragmentations for this particular workload. We have evaluated only one application in this section and more extensive study on the benefit and cost of defragmentation is an interesting future work.

5. Discussions

In this section, we consider how important our observations are for systems other than Web application servers.

The language runtime systems supporting garbage collection for memory management, such as Java™ VMs and .net runtimes, are widely used today. Many of these virtual machines, especially those using copying garbage collectors, allocate heap memory for newly created objects in a similar way to the region-based allocators, in which the allocation is done by simply incrementing a pointer that tracks the current location to allocate. This allocation mechanism is widely used because of the virtue of small allocation overhead. In those virtual machines, however, allocated objects are not freed until the heap becomes full and the virtual machines execute garbage collection. Hence the virtual machines may suffer from the increased bus traffic on multicore processors, just as the region-based allocator suffers in the PHP runtime, because they cannot reuse the memory locations used by already-dead objects.

In the GC-based languages, programmers do not free objects explicitly and hence reusing the memory locations quickly is not a trivial task. However techniques to quickly reclaim short-lived objects are quite important to reduce the hidden costs of the increased bus traffic and to achieve high performance on multicore processors. For example, the advanced escape analysis of Shankar *et al.* [23] is a good way to quickly reuse the short-lived objects by allocating them on a stack. For another example, MicroPhase of Xian *et al.* [24] can improve the memory locality by aggressively invoking a garbage collection before the Java heap becomes full.

6. Related Work

Memory management consumes a considerable fraction of CPU time in applications [25]. Therefore many designs of dynamic memory management have been proposed in the past [26]. For general purpose use, an allocator by Doug Lea [13] is known as one of the most advanced designs of dynamic memory management, which balances several goals, including speed, space, and portability. Recent advances in multicore processors require good scalability for every application. Hence currently many memory allocators focus on scalability for multi-threaded applications. Hoard [11], TCmalloc (included in google-perf-tools) [12] are such examples. Many of them achieve good scalability of multi-threaded applications by avoiding lock contentions for heap accesses and false sharing. In this paper, we focus on poor scalability caused by the limited bus bandwidth on multicore processors as another reason for poor scalability.

There have been many efforts to exploit knowledge about the characteristics of the applications, such as the lifetimes or sizes of the objects, to improve the performance of memory management [1-4, 27, 28]. Region-based memory management [1-4], which allows fast allocation by simply incrementing a pointer and freeing multiple objects at once, is one of the most attractive ways to exploit the knowledge of the lifetimes of the objects. As already discussed in this paper, the region-based memory management is very effective in reducing the overhead of memory management. However, it may suffer from poor scalability on multicore processors. Our proposed defrag-dodging approach can also exploit the knowledge of the object lifetimes to reduce the overhead of memory management, but it yields higher scalability than the region-based memory management.

Berger *et al.* [9] proposed an allocator, called Reaps, that combines the conventional malloc/free and the region-based memory management. Like our defrag-dodging approach or the custom allocator in the PHP runtime, it supports both per-object free and bulk free for all of the objects in a region. In contrast to ours, their allocator acts in almost the same way as Doug Lea's allocator [13] for per-object free and does not focus on improving the performances of the per-object free. Thus the Reaps also pays cost of the defragmentation activities, which is excessive for short-lived transactions in Web-based applications, like the default allocator of the PHP runtime.

7. Conclusions

In this paper, we examine the performance of a general-purpose allocator and a non-freeing region-based allocator using Web-based workloads on two platforms with multicore processors. Our results show that the region-based allocator achieves much better performance for all workloads on one or a few processor cores due to its smaller memory management costs. However the region-based allocators suffer from hidden costs of increased bus traffic on multicore environments and the performance is reduced by as much as 27.2% compared to the default allocator when using eight cores. This is because the system memory bandwidth tends to become a bottleneck in systems with multicore processors.

This paper describes a new memory management approach, called defrag-dodging, for transaction-scoped objects in Web-based applications. The approach can improve the performance of Web-based applications on multicore processors by reducing the total costs of memory management without increasing the bus traffic. The key to the reduced costs for memory management in the defrag-dodging allocator is that it avoids the defragmentation activities in the malloc and free invocations during transactions. In Web-based applications, the costs of the defragmentation activities in existing general-purpose allocators outweigh their benefits. We show that the throughput with our new approach is higher than with the other two allocators for all of the applications when using eight cores. The improvements are up to 11.4% and 51.5% over the default allocator and the region-based allocator, respectively. Our results show that the increasing use of multicore processors is significantly changing the requirements for memory allocators to fully benefit from the large amounts of computing resources provided by such multicore processors.

Acknowledgments

We are grateful to the anonymous reviewers for their thoughtful comments and suggestions. We thank Tamiya Onodera, Rei Odaira, and Kazunori Ogata for their useful feedback on earlier drafts on this work. We also thank Shannon Jacobs for his editorial assistance and valuable suggestions including the name *de-frag-dodging*.

References

[1] D. R. Hanson. Fast allocation and deallocation of memory based on object lifetimes. *Software—Practice & Experience*, 20(1), pp. 5–12, 1990.

[2] D. A. Barrett and B. G. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 187–196, 1993.

[3] D. Gay and A Aiken. Memory management with explicit regions. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 313–323, 1998

[4] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-Based Memory Management in Cyclone. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pp. 282–293, 2002.

[5] Apache Software Foundation. *The Apache Portable Runtime Project*. http://apr.apache.org/ .

[6] R. Iyer, M. Bhat, L. Zhao, R. Illikkal, S. Makineni, M. Jones, K. Shiv, and D. Newell. Exploring Small-Scale and Large-Scale CMP Architectures for Commercial Java Servers. In *Proceedings of the IEEE International Symposium on Workload Characterization*, pp. 191–200, 2006.

[7] Y. Chen, E. Li, W. Li, T. Wang, J. Li, X. Tong, P. P. Wang, W. Hu, Y. Zhang, Y. Chen. Media mining – emerging tera-scale computing applications. *Intel Technology Journal*, 11(3), pp 239–250, 2007.

[8] The PHP Group. *PHP: Hypertext Preprocessor*. http://www.php.net/ .

[9] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. In *Proceedings of the ACM Conference on Object Oriented Programming, Systems, Languages, and Applications*, pp. 1–12, 2002.

[10] Free Software Foundation, Inc. *GNU C Library obstack*. http://www.gnu.org/software/libc/manual/html_node/Obstacks.html.

[11] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 117–128, 2000.

[12] S. Ghemawat and P. Menage. TCMalloc : Thread-Caching Malloc. http://goog-perftools.sourceforge.net/doc/tcmalloc.html .

[13] Doug Lea. *A Memory Allocator*. http://g.oswego.edu/dl/html/malloc.html .

[14] M. S. Johnstone and P. R. Wilson. The memory fragmentation problem: Solved. In *Proceedings of the International Symposium on Memory Management*, pp. 26–36, 1998.

[15] Wikimedia Foundation. *MediaWiki*. http://www.mediawiki.org .

[16] SugarCRM Inc. *SugarCRM*. http://www.sugarcrm.com .

[17] eZ Systems. *eZ Publish*. http://ez.no .

[18] The phpBB Group. *phpBB*. http://www.phpbb.com/ .

[19] Cake Software Foundation, Inc. *CakePHP*. http://www.cakephp.org/ .

[20] Standard Performance Evaluation Corporation. *SPECweb2005*. http://www.spec.org/web2005/ .

[21] OProfile - A System Profiler for Linux. http://oprofile.sourceforge.net/news/ .

[22] D. H. Hansson. *Ruby on Rails*. http://www.rubyonrails.org .

[23] A. Shankar, M. Arnold, and R. Bodik. Jolt: lightweight dynamic analysis and removal of object churn. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pp. 127–142, 2008.

[24] F. Xian, W. Srisa-an, and, H. Jiang. Microphase: an approach to proactively invoking garbage collection for improved performance. In *Proceedings of the ACM Conference on Object Oriented Programming Systems Languages and Applications*, pp. 77–96, 2007.

[25] D. Detlefs, A. Dosser, and B. Zorn. Memory allocation costs in large C and C++ programs. *Software—Practice & Experience*, 24(6), pp. 527–542, 1994.

[26] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proceedings of the International Workshop on Memory Management*, pp. 1–116, 1995.

[27] M. L. Seidl, and B. G. Zorn. Segregating heap objects by reference behavior and lifetime. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 12–23, 1998.

[28] Y. Shuf, M. Gupta, R. Bordawekar, and J. P. Singh. Exploiting prolific types for memory management and optimizations. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pp. 295–306, 2002.