Spring Buddy: A Self-Adaptive Elastic Memory Management Scheme for Efficient Concurrent Allocation/Deallocation in Cloud Computing Systems

Yihui Lu[1], Weidong Liu[1], Chentao Wu[1,2*], Jia Wang[3], Xiaoming Gao[3], Jie Li[1], Minyi Guo[1]
[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China
[2]Sichuan Research Institute, Shanghai Jiao Tong University, Sichuan, China
[3]Cloud Architecture & Platform, TEG, Tencent Inc., Shenzhen & Beijing, China
*Corresponding Author: wuct@cs.sjtu.edu.cn

***Abstract*—Within the cloud computing scenario, each server usually carries multiple service processes, which intensifies the concurrency pressure of the system. As a result, the process of memory management during page allocation and deallocation becomes a significant bottleneck. Although several methods such as Buddy System and Inverse Buddy System (iBuddy) have been proposed to improve the performance of memory management, they cannot adapt to the highly concurrent environment of cloud computing, because they either force the memory allocation/deallocation requests to be serialized or bring extra fragmentation.**

To address the above problem, we propose Spring Buddy, which improves the concurrency of both memory allocation and deallocation and avoids unnecessary fragmentation. It can detect the changes of system- and process-level memory request patterns and dynamically adjust the organization of page frames. Inventively, Spring Buddy uses the spring core layer to provide both concurrent response and resource aggregation capability which is adapted to the system's concurrency pressure, and also uses the spring lazy layer to further mitigate the system resource contention through process behavior prediction. To demonstrate the effectiveness of Spring Buddy, we implement it in the Linux kernel. The results demonstrate that Spring Buddy can reduce memory allocation latency by 71.47% and deallocation latency by 93.20% on average compared to the existing methods.

***Index Terms*—memory management, buddy system, resource contention, memory allocation/deallocation, scalability**

I. Introduction

As cloud computing technology evolves, multiple services are usually aggregated on the same server to achieve higher hardware resources utilization [1][2][3]. However, inter-process contention for resources is exacerbated in such systems, causing severe performance degradation under the concurrent requests [4][5]. Therefore, improving the parallelism among processes has become a hot topic for making full use of hardware resources and enhancing service capabilities [6][7].

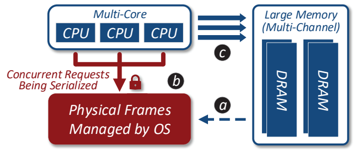

Multi-Core
CPU
CPU
CPU
Large Memory (Multi-Channel)
DRAM
DRAM
c
Concurrent Requests Being Serialized
b
a
Physical Frames Managed by OS

Fig. 1: Three steps of a memory access: a) collect information at system boot. b) serialized memory request for physical frames. c) parallelized memory access.

Generally, with the increasing number of CPU cores and memory size, the scalability issues with memory management become prominent because of the existing workflow of memory accessing [8]. As shown in Fig. 1, a memory access consists of three main steps. During bootup, the operating system collects hardware data and divides physical memory into a large number of page frames (Step a). When a page fault occurs, the OS allocates a free page frame and modifies its state (Step b). Finally, processes access memory via the MMU (Memory Management Unit) and the memory bus in parallel [9] (Step c).

When multiple memory allocation/deallocation requests are generated concurrently, they have to be serialized in Step b, resulting in a sharp degradation of system performance [10]. To highlight this problem, we run several workloads (detailed in Section IV-A) with various numbers of threads and analyze the average latency of the memory allocation/deallocation process, as shown in Fig. 2, where a significant latency penalty occurs when waiting for spin locks, especially when the concurrency increases in the whole system.

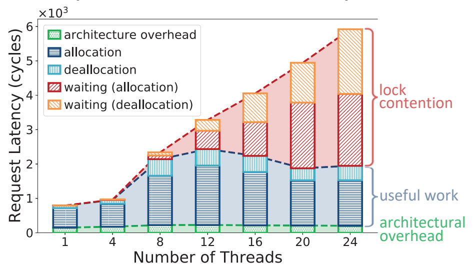


Fig. 2: Increased latency of memory requests in case of concurrency, caused by serialized memory management.

The key to mitigate such latency of concurrent requests is to improve the concurrency of memory management. Some methods reduce the waiting time of processes by simplifying the page frame maintenance process [11][12][13]. Other methods reduce lock contention by region segmentation or pre-allocation [14][15][16]. However, there are still several drawbacks encountered as the memory capacity or pressure increases due to the following reasons. First, they ignore the concurrency of memory deallocation requests. Second, they fail to dynamically aggregate contiguous free frames, resulting in unnecessary tremendous fragmentation in the system. Finally, they treat successive memory requests as independent events, ignoring the correlations among them.

To address the above issues, we propose Spring Buddy, a memory management scheme that dynamically adapts to the memory request pressure in the system and enables spring-like elastic resource management to improve the responsiveness to concurrent memory allocation/deallocation requests in cloud computing systems with our innovatively proposed spring core layer and spring lazy layer.

In this paper, we have the following contributions.

- We design an elastic memory management scheme that can automatically adapt to the concurrency of the system. At high concurrency, it satisfies responsiveness requirements, and at low concurrency, it satisfies resource aggregation requirements.
- We design elastic per-CPU private buffers that can analyze and exploit memory request correlation to improve buffer utilization.
- Several experiments have been conducted to demonstrate that the proposed scheme (Spring Buddy) can reduce the latency of both memory allocation and deallocation requests in highly concurrent environments.

The rest of paper is organized as follows. Section II introduces the related work and our motivations. The design of Spring Buddy is illustrated in detail in Section III. The evaluation of Spring Buddy and other methods are presented in Section IV. Finally, we draw our conclusion in Section V.

II. Background and Motivation

In this section, the background and related works are illustrated in detail, and our motivation is given at the end of this section.

A. Overview of Memory Management Schemes

Memory management scheme is used to organize physical page frames, which is critical for operating systems that support memory paging techniques [17]. Furthermore, in cloud computing systems, responsiveness to concurrent memory requests becomes another important performance indicator for memory management. Currently, there are two representative schemes, called standard buddy system (the most widely used) and iBuddy (the state-of-the-art).

1) Standard Buddy System: standard buddy system (or binary buddy system) [18] is the most widely used memory management scheme, especially in Unix-like systems [19]. It is first proposed by Knowlton [20][21] and has been continuously improved [22][23]. When a process requests physical memory, the buddy system allocates the corresponding free physical frames to it. And when a process releases the memory, the corresponding physical frames are deallocated and merged with their adjacent free frames (buddy frames) for special requests requiring contiguous physical memory. Similarly, high-order[1] contiguous pages can be split at allocation time to satisfy the low-order requests. Besides, all requests must be serialized in the standard buddy system.

[1]In the buddy system, "order" indicates the length of contiguous physical frames. For example, an order-4 physical frame consists of 2^4 single frames

2) iBuddy: iBuddy (inverse buddy system) [15] is the state-of-the-art variant based on the standard buddy system. On the one hand, it designs a special bitmap to record page frame states and uses a worst-fit mechanism for allocation. As a result, it reduces the allocation overhead for a single frame to $O(1)$ at the cost of fragmentation. On the other hand, it pre-divides the memory into 4MB blocks and binds each page to the unique memory block where it belongs. When handling memory allocation requests, iBuddy first allocates a private memory block for each CPU and then allocates pages from that block. If the pages in that block are exhausted, a new non-empty memory block will be fetched from the global resource pool. However, when handling memory deallocation requests, iBuddy needs to place memory pages into the appropriate memory blocks. This operation should be serialized if these memory blocks are being used (both allocation or deallocation) by other CPU cores.

3) Other Memory Management Schemes: Several studies are conducted to accelerate the page frame maintenance process. Leite and Rocha [11] create a lock-free region splitting and merging mechanism based on boundary flags. Brodal et al. [12] propose a superblock-based approach in which lazy splitting is enabled by superblocks that contain allocatable blocks of varying sizes. Barkley and Lee [13] propose a lazy buddy system that uses the DELAY-2 coalescence policy to limit coalescence delays to two per class. Herter et al. [24] propose a dynamic memory allocation algorithm for hard real-time setups where worst-case execution time analysis is paramount.

Other studies have found that artificially dividing the memory space or privatizing a portion of the memory can reduce the lock contention. Through process-level pre-allocation and page reuse within a thread group, a series of user-mode memory pools reduce requests to the underlying memory management system [16][25][26]. Marotta et al. [14] design a page management method based on a complete binary tree called NBBS. It provides independent management of page frames belongs to different sub-trees.

Besides, there are also studies on specialized scenarios, such as high-performance computing [27][28] and in-memory data processing [29][30][31][32]. They take advantage of the memory access patterns or hardware characteristics specific to these scenarios to optimize the memory management process.

B. Our Motivation

TABLE I: Summary on Representative Schemes

	Allocation Concurrency	Deallocation Concurrency	Resource Aggregation Ability	Requests Prediction
std-Buddy	serialized	serialized	high	no
iBuddy	high	low	low	no
Spring Buddy	high	high	high	yes

We summarize the problems of existing representative schemes in Table I. The traditional schemes, such as the standard buddy system, are designed for small-size single-core systems and thus do not consider concurrent service capabilities. The state-of-the-art schemes, such as iBuddy, ignore the need for concurrency in the deallocation process.

Moreover, it sacrifices the ability to aggregate contiguous memory pages, exacerbating the fragmentation problem. Besides, they both treat successive memory requests as the independent events, ignoring the correlation of requests brought by process behavior. To this end, we propose the Spring Buddy, which dynamically responds to the concurrency of memory requests in the system and predicts potential requests to achieve elastic memory management, meeting the concurrent access and resource aggregation requirements in cloud computing systems.

III. Design of Spring Buddy

This section provides an overview of Spring Buddy and describes each component in detail.

A. Spring Buddy Overview

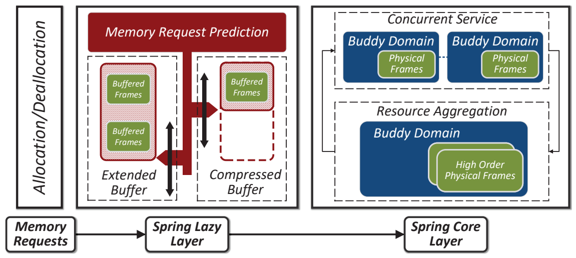


Fig. 3: Overview of the Spring Buddy.

Fig. 3 shows the overview of Spring Buddy which consists of two main layers, including the spring core layer and the spring lazy layer.

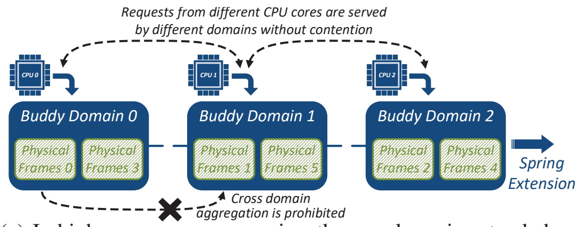
- **Spring core layer:** Spring Buddy dynamically adapts to the global concurrency pressure in the system. The spring core layer is expanded to satisfy requests from different CPU cores in the case of high concurrency, and is compressed in the case of low concurrency for appropriate resource aggregation and fragmentation reduction.
- **Spring lazy layer:** Spring Buddy uses a dynamically adjusted lazy mechanism for requests buffering. The spring lazy layer adjusts the per-CPU buffer size based on the prediction of the subsequent potential memory requests. As a result, it increases the buffer hit rate while reducing the wasted resources.

B. Spring Core Layer

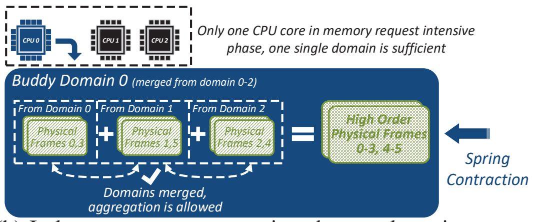
The core layer manages free page frames on a system-wide basis which needs to balance concurrent response with resource aggregation capabilities. Therefore, we propose Spring Core Layer with self-adaptive elastic memory management to satisfy the system requirements in different scenarios.

1) Buddy Domains as Spring: Spring Buddy is used to divide memory into multiple buddy domains in a self-adaptive manner. Moreover, the merging and splitting of the contiguous pages is restricted to a single domain. Here, the domain boundaries are elastically adjusted (hence the reason of name Spring) according to the changes in concurrency in the system.

When the highly concurrent requests for memory is generated in the system, as shown in Fig. 4(a), meeting the concurrency of the system becomes a priority. Therefore, the memory is dynamically split into multiple domains and the whole spring is extended. As a result, requests from different CPU cores can be served by different domains without contention, achieving a high concurrency in memory management. When the concurrency demand in the system decreases, as shown in Fig. 4(b), avoiding fragmentation becomes a priority again. It is no longer necessary to divide multiple domains, so the buddy domains are merged and the whole spring is compressed. As a result, the merged domains expand the management boundaries of the resource pool so that contiguous pages distributed across multiple domains can be merged into higher-order pages. This makes full use of the opportunity of free page aggregation that had been overlooked.



(a) In high concurrency scenarios, the core layer is extended so that multiple domains can respond to requests from different CPU cores at the same time.


CPU 0
CPU 1
CPU 2
Only one CPU core in memory request intensive phase, one single domain is sufficient
Buddy Domain 0 (merged from domain 0-2)
From Domain 0
Physical Frames 0,3
From Domain 1
Physical Frames 1,5
From Domain 2
Physical Frames 2,4
High Order Physical Frames 0-3, 4-5
Spring Contraction
Domains merged, aggregation is allowed

(b) In low concurrency scenarios, the core layer is compressed so that larger domains can better exploit page merging opportunities.

Fig. 4: Spring core layer with adaptive resource extension or compression capabilities.

As concurrency changes in the system, spring core layer needs to dynamically adjust the number of domains to satisfy the system's trade-off for concurrency and defragmentation under different situations. Therefore, the extension and compression of the spring do not depend on pre-defined parameters but are automatically managed by Spring Buddy based on the system state. To better illustrate how the spring core layer identifies system concurrency and adjusts the number of domains, the extension and compression process are described in Section III-B2 and Section III-B3 respectively.

2) Spring Extension: To ensure system concurrency, Spring Buddy needs to split the domain immediately when the system is under the high concurrent pressure. As shown in Fig. 5, this process can be divided into three phases.

The first phase detects the increase of concurrent requests in the system. Initially, there is only one buddy domain, and it is used by CPU 0 (Step a). When CPU 1 makes a memory request, there is no buddy domain left (Step b). Thus, a ghost domain is created, and CPU 1 is forced to wait (Step c).

The second phase performs the splitting of domain: when CPU 0 finishes the requests and releases Domain 0, it detects the appearance of the ghost domain, and passes a part of its pages to the waiting ghost domain (Step d). Then, this ghost domain is activated and transformed into a new buddy domain to serve CPU 1 (Step e). Finally, the remaining part of Domain 0 is put back into the global spring (Step f).

The third phase completes the spring extension: when CPU 1 finishes the request, Domain 1 is put back into the global spring (Step g). Currently, the number of buddy domains is increased. As a result, subsequent concurrent requests can be handled without the above extension operations.

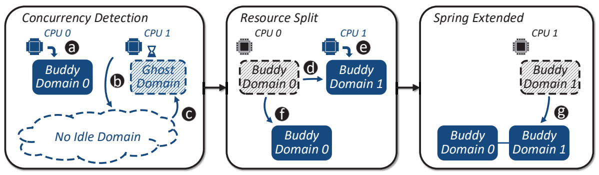


Fig. 5: Extension of spring core layer. Triggered when the system is in a high concurrency phase to generate new buddy domains and improve system concurrency.

3) Spring Compression: When the system enters the stage of low concurrent requests, Spring Buddy needs to efficiently merge domains, compress the spring and perform defragmentation operations. As shown in Fig. 6, this process of compression can be divided into three phases.

The first phase detects the presence of redundant resources in the system. When Domain 0 completes its service and is put back into the global spring, it triggers the detection operation (Step a). Then two redundant domains are detected in the system that have been idle for a long time (Step b).

The second phase merges the redundant domain to better aggregate resources. The resource manager picks two suitable domains and performs the merge (Step c). They are merged from separate small fragmented domains into a single large domain with more high-order free frames (Step d).

The third phase completes the compression of the spring. When the resource aggregation is complete, the sequentially generated domain is put back into the global spring (Step e). At this moment the spring core layer is intuitively compressed and the fragmentation rate is decreased (Step f).

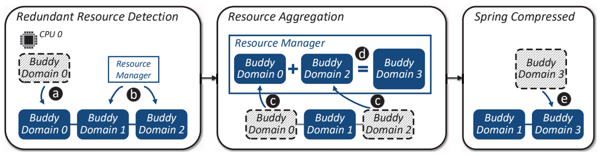


Fig. 6: Compression of spring core layer. Triggered when the system is in a low concurrency phase to aggregate idle domains and reduce fragmentation.

C. Spring Lazy Layer

To further reduce the pressure of contention for global system resources under the intensive memory requests, spring lazy layer is introduced in Spring Buddy to dynamically adjust the length of per-CPU private buffer to aggregate memory requests. The design of the spring lazy layer is based on our observation of process memory request patterns and is used to reduce contention over global resources.

1) Pattern of Memory Requests: The design of the spring lazy layer is based on the pattern of process memory requests. Memory request pattern refers to the fact that a process may generate memory requests continuously over a short time. Since this series of memory requests often comes from the same address space segment within the process's virtual address space, which is highly predictable. The reason for this pattern is that the page frames mapped within the same address space segment have the same type, e.g. they are mapped to the same file, or have the same read/write permissions. Therefore, they tend to have a similar life cycle. As a result, these frames are usually allocated in batches as the process initializes its data or deallocated in batches as the process releases its memory.

To prove this hypothesis, the number of such successive memory requests from the same address space is counted under different workloads, as shown in Table II. Two characteristics can be noticed: 1) memory requests from the same process address space are highly continuous and predictable. 2) the patterns of each workload vary greatly, so the batch size of request aggregation needs to be self-adaptive.

TABLE II: Average number of successive memory requests from the same virtual address space under different workloads

Workloads	Allocation	Deallocation
Ramspeed	2265.03	16929.51
Stream	122.01	975.93
Linux Scalability	2500.36	10010.51
Threadtest	6242.04	74248.67
Sysbench	4835.61	68491.47
Average of five workloads	**3193.01**	**34131.22**

Based on the above features, we designed Spring Lazy Layer, which obtains both process behavior and address space information, to predict subsequent memory requests, and to dynamically aggregates request that occur in a short time.

2) Lazy Layer as Spring: The design of the Spring Lazy Layer is shown in Fig. 7. Take the memory deallocation request process as an example: when a process releases an allocated page frame, it is first placed in the lazy buffer and then released in bulk when the buffer limit is reached (Step a). At this moment, Spring Buddy collects information about the process's memory request behavior and the mapping of its virtual address space (Step b). With the information collected, Spring Buddy predicts the probability of subsequent page deallocation requests (Step c). Suppose there are a large number of pages of the same type in the address space and multiple deallocation requests have already occurred, there is a high probability that a batch of deallocation requests will continue to be generated in a short time. Based on the above prediction, the buffer boundaries are adjusted to accommodate more pages (Step d). The extended buffer avoids global interactions during subsequent page deallocation

(Step e). When the prediction module determines that this batch request has ended, the pages in the buffer are released into the global resource pool at the end (Step f).

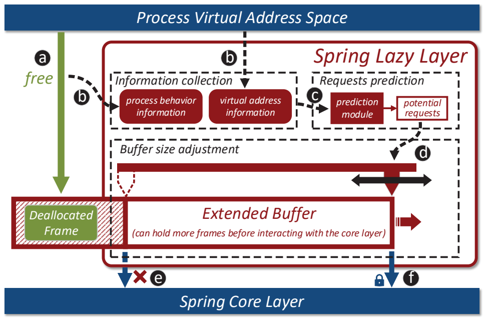


Fig. 7: Spring lazy layer that predicts subsequent memory requests and dynamically adjusts buffer size based on process request patterns.

IV. Performance Evaluation

In this section, we evaluate the performance of various memory management schemes to demonstrate the effectiveness of Spring Buddy. This section contains three parts: In Section IV-A, the evaluation environment, the workloads, and the metrics are described. And In Section IV-B the overall performance improvement of Spring Buddy is explained and analyzed. Moreover, In Section IV-C, the independent evaluation on each component of Spring Buddy is performed to demonstrate further its optimization result, including effectiveness and resource utilization of the spring lazy layer (Section IV-C1), and the concurrency and fragmentation of the spring core layer (Section IV-C1).

A. Evaluation Methodology

1) Experiment Environment: The experiments are conducted between three different memory management schemes, involving the standard buddy system [18] (denoted std-Buddy), inverse buddy system [15] (denoted iBuddy), and Spring Buddy with both spring core layer and spring lazy layer (denoted Spring-Buddy).

The environment we configured is shown in Table III. All memory management schemes are implemented and integrated into the Linux kernel within such configuration.

TABLE III: Evaluation environment

Item	Description
CPU	Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz
CPU Cores	24
RAM	128 GB
OS	Ubuntu Server 16.04
Kernel	Linux 4.14

The workloads we selected for generating memory requests are shown as below:

- **Ramspeed [33]:** This workload is used to measure the cache and memory performance of computer systems. In our experiments, the memory size is set to 1GB per pass, incurring roughly 4.0×10^5 memory requests per second.
- **Stream [34]:** This workload is a synthetic workload used for measuring sustainable memory bandwidth in high-performance systems. In our experiments, the problem size is set to 100 million, incurring averagely 8.8×10^5 memory requests per second.
- **Linux-scalability [16]:** This workload measures memory allocation latency and the scalability of the system. In our experiments, the default parameters are used, incurring roughly 2.6×10^5 memory requests per second.
- **Threadtest [16]:** This workload is used to measure system throughput in a multi-threaded situation. In our experiments, the object size is set to 1KB, incurring roughly 8.1×10^5 memory requests per second.
- **Sysbench [35]:** This workload is a scriptable multi-threaded workload. In our experiments, we use the memory bench, with the block size set to 2GB, incurring roughly 3.5×10^5 memory requests per second.

2) Metrics for Experiment: We use the following metrics to compare the performance of different methods.

- **Request latency:** The average latency between initiating a memory request and getting the response, which is an important and the most intuitive measurement of the memory allocation/deallocation performance.
- **Lazy buffer miss rate:** the ratio of requests that cannot be served by the lazy layer. The lower the miss rate, the higher the effectiveness of the buffer.
- **Lazy buffer residual:** the number of unused pages left in the buffer at the end of a batch of memory allocation/deallocation requests. The lower the residual, the higher the buffer utilization.
- **Resource contention rate:** the probabilities of contention with other processes when requesting. The lower the contention rate, the better the parallelism of the system.
- **Fragmentation rate:** the ability to satisfy requests for high-order physical frames. The lower the fragmentation rate, the better the scheme's ability to aggregate resources

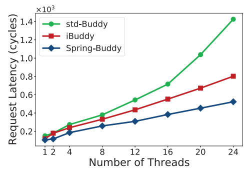
B. Overall Performance

To evaluate the overall optimization effect of various memory management approaches, we have executed the workloads with 1 to 24 threads under three different memory management schemes. The experimental results are as follows.

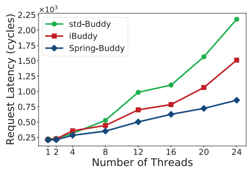
Request latency of allocation: Fig. 8 shows the average latency of memory allocation requests, where the x-axis coordinate represents the number of threads running concurrently and the y-axis coordinate represents the number of CPU cycles elapsed from the request to the response. Taking the high concurrency case represented by 24 threads, Spring Buddy reduces the latency by 71.47% and 42.27% on average compared to std-Buddy and iBuddy, respectively.

Request latency of deallocation: Fig. 9 reflects the average latency of memory deallocation requests. With 24 threads, Spring Buddy reduces the latency by an average of 93.20% and 75.66% compared to std-Buddy and iBuddy, respectively.

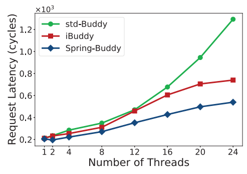
From the above results, we conclude that Spring Buddy achieves a significant improvement over std-Buddy and iBuddy in terms of both allocation and deallocation request latency in highly concurrent situations. Compared to std-Buddy, the spring core layer in Spring Buddy splits a single



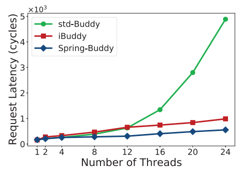
(a) Ramspeed



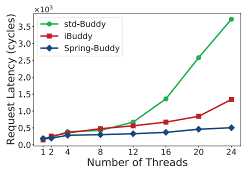
(b) Stream



(c) Linux-scalability

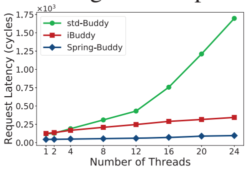


(d) Threadtest

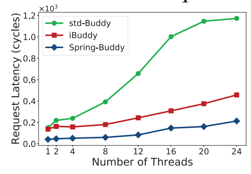


(e) Sysbench

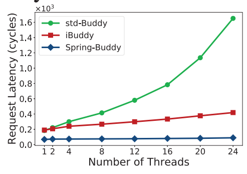
Fig. 8: Comparison of allocation request latency under different workloads with different concurrent pressures.



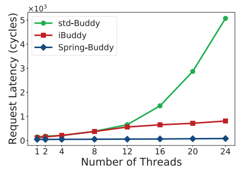
(a) Ramspeed



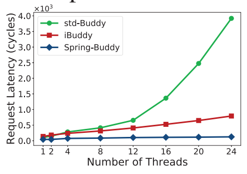
(b) Stream



(c) Linux-scalability



(d) Threadtest



(e) Sysbench

Fig. 9: Comparison of deallocation request latency under different workloads with different concurrent pressures.

resource pool into multiple domains for fine-grained resource management, where memory requests from multiple processes can be served by different domains simultaneously. With such design, the contention between processes is limited to simple operations of fetching/returning domains from/to the global spring, which make the time-consuming high-order frame merge and split operations to be moved out of the critical zone, resulting in the increased parallelizable part of the request response process. Moreover, the shorter critical zone allows less contention to occur in Spring Buddy, so there is less useless waiting time and more responsiveness.

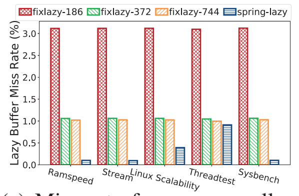
Moreover, compared with iBuddy which also uses a resource pool splitting mehcanism, Spring Buddy still performs much better because of the independence of the deallocation behavior. Within iBuddy's design, pages must be deallocated to their unique corresponding blocks, which introduces contention issues when pages are deallocated from other CPU cores. In fact, contention during deallocation also affects the allocation operation of that block, causing serialization problems to reappear. However, Spring Buddy does not have this constraint, as pages and domains are fully connected to each other, deallocation can be done in parallel for different processes as well as for allocations.

Furthermore, the spring lazy layer innovatively implemented by Spring Buddy also contributes to the reduction of request latency. Compared with the fixed-length buffer in std-Buddy and the lazy-bypass mechanism in iBuddy, spring lazy layer in Spring Buddy contains a flexible buffer, which can better reduce the frequency of global resource accessed and mitigate the concurrency pressure in the system.

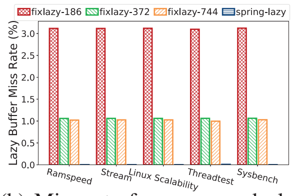
C. *Performance of Each Component*

In this section, we quantify the effect of the proposed components of the Spring Buddy system including spring core layer and spring lazy layer.

1) Spring Lazy Layer: We have executed the workloads with 24 threads in the former paragraphs, under four different lazy layer configurations. Fixlazy-186 refers to a buffer with a fixed length of 186 frames (fixlazy-372/744 is defined similarly). It triggers a fetch (or release) from the global free page list when the number of frames in the buffer is zero (or exceeds the upper limit), and the batch size processed each time is 1/6 of the buffer size, which is the default setting in the kernel. Meanwhile, spring-lazy means that the buffer size or batch is not artificially specified, but is dynamically adjusted using prediction, which can significantly reduce the buffer miss rate without bringing too many residual pages. The experimental results are as follows.

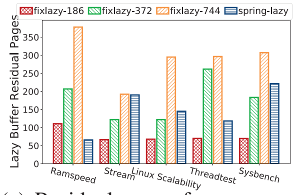


(a) Miss rate for memory allocation requests.

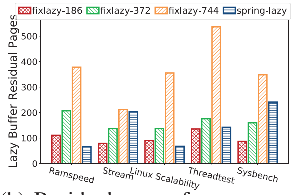


(b) Miss rate for memory deallocation requests.

Fig. 10: Comparison of the miss rate of lazy buffer with different configuration under different workload. Lower miss rate means higher buffer effectiveness.



(a) Residual pages for memory allocation requests.



(b) Residual pages for memory deallocation requests.

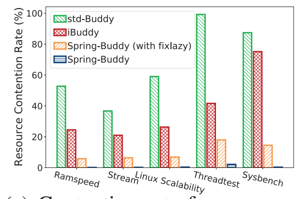
Fig. 11: Comparison of the amount of residual pages in lazy buffer with different configuration under different workload. Lower residual means higher buffer utilization.

Lazy buffer miss rate: At allocation time, as shown in Fig. 10(a), compared with fixlazy-186/372/744, spring-lazy can reduce the buffer miss by 2.79%, 0.74% and 0.70% on average. At deallocation time, as shown in Fig. 10(b), compared to fixlazy-186/372/744, spring-lazy can reduce the buffer miss rate by 3.11%, 1.06% and 1.02% on average.

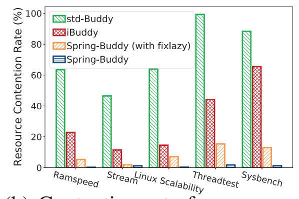
Lazy buffer residual: At allocation time, as shown in Fig. 11(a), the average residuals of spring-lazy are 2.40×, 1.10× and 0.64× of fixlazy-186/372/744, respectively. At deallocation time, as shown in Fig. 11(b), the average residuals of spring-lazy are 1.54×, 0.92× and 0.45× of fixlazy-186/372/744, respectively.

From the above results, it can be noticed that simply increasing the buffer size in the fixed-length case only leads to more resources being wasted, and cannot effectively reduce the buffer miss rate. On the contrary, the spring lazy layer owns prediction capability to automatically adjust the buffer size, which can significantly reduce the miss rate. Although spring lazy layer may make a compensation with the trade-off of the certain degree of the residual increase, it still wastes less resources than the fixed length buffer which can achieve the approximate miss rate. Thus this trade-off is negligible. This proves that the design of the spring lazy layer has a better overall performance.

2) Spring Core Layer: We have executed the workloads with 24 threads, under three different buddy systems including std-Buddy, iBuddy, and Spring Buddy. To get rid of the impact of lazy layer's buffering ability on resource contention, we additionally examined the Spring Buddy scheme without spring lazy layer (denoted Spring-Buddy with fixlazy). The experimental results are as follows.



(a) Contention rate for memory allocation requests.



(b) Contention rate for memory deallocation requests.

Fig. 12: Comparison of memory requests contention rate under different workloads. Lower contention rate means better system parallelism.

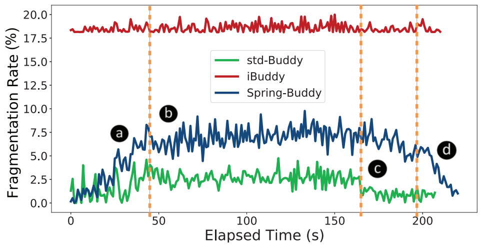


Fig. 13: Comparison of fragmentation rates of each method, when the system is in different stages. Lower fragmentation rate means better resource aggregation ability

Resource contention rate: At allocation time, as shown in Fig. 12(a), Spring-Buddy with fixlazy can reduce the contention rate by 56.71% and 27.41% compared to std-Buddy and iBuddy, respectively. At deallocation time, as shown in Fig. 12(b), Spring-Buddy with fixlazy can reduce the contention rate by 63.14% and 23.14% compared to std-Buddy and iBuddy respectively. If both spring core layer and lazy layer are used, the resource contention rate can be further reduced by 66.35% and 37.04% at allocation and 71.31% and 30.70% at deallocation compared to std-Buddy and iBuddy.

Fragmentation rate: The fragmentation rate is shown in Fig. 13, where we launch 5 workloads simultaneously, and each workload runs 5 threads. As can be seen, the change of fragmentation rate can be divided into four stages: a) When the workload starts, Spring Buddy detects an increase in system concurrency, so it extends the spring and generates multiple buddy domains, resulting in the increased fragmentation rate. b) After the system concurrency stabilizes, Spring Buddy optimizes the required domains to a minimum. At this point, the average fragmentation rate is 6.83% for Spring Buddy, 2.61% for std-Buddy and 18.61% for iBuddy. c) When the processes end, the system fragmentation rate decreases as the free pages are deallocated and merged, but many contiguous pages in Spring Buddy are separated in different domains. d) After a while, Spring Buddy detects that the system concurrency has stabilized at a low level again, then it merges multiple small fragmented domains into one larger one with high-order memory, bringing the fragmentation rate back down to the same level as std-Buddy.

The above results show that the spring expansion, in the spring core layer, significantly reduces resource contention in the system. Compared to std-Buddy, this optimization is due to the parallelism brought by multiple domains, which dramatically reduces the critical zone occupation time. Compared to iBuddy, such improvement is on account of the fully associative mapping among pages and domains, which avoids lock contention caused by deallocation.

Meanwhile, the compression of the spring core layer is used to dynamically adjust system fragmentation rate. Compared to iBuddy, which is always highly fragmented, Spring Buddy appropriately adjusts the degree of domain split based on the actual concurrency of the system, avoiding bringing in excessive fragmentation issues. Although Spring Buddy causes more fragmentation to meet concurrency requirements compared with the serialized std-Buddy, it is automatically aggregated after the concurrency phase has passed. Thus, such fragmentation trade-off can be ignored considering the significant performance difference between them.

D. Analysis

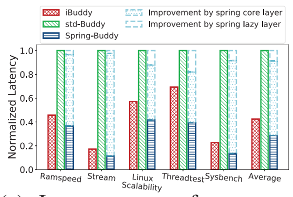
The above results illustrate that Spring Buddy outperforms the currently used memory management schemes with the achievements of self-adaptive spring core layer and spring lazy layer, which reduces the global resource contention and mitigates the memory allocation/deallocation latency. In addition, a brief summary table about the reduction on average memory request latency with different workloads of high concurrency is given in Table IV. And the source of the performance improvement is shown in Fig. 14.

V. Conclusion

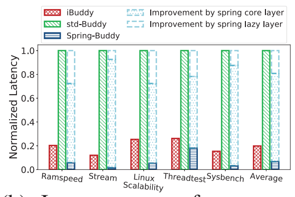
In this paper, we propose Spring Buddy to avoid the intensive latency overhead of memory requests in highly

TABLE IV: Overall improvement over std-Buddy and iBuddy

Improvement / Workloads	Comparison with std-Buddy (%)		Comparison with iBuddy (%)	
	allocation latency	deallocation latency	allocation latency	deallocation latency
Ramspeed	63.30%	94.34%	34.79%	72.09%
Stream	88.64%	98.38%	43.71%	89.86%
Linux-scalability	58.31%	94.61%	27.16%	78.76%
Threadtest	60.68%	81.84%	43.31%	53.39%
Sysbench	86.41%	96.81%	62.37%	84.22%
Average of five workloads	**71.47%**	**93.20%**	**42.27%**	**75.66%**



(a) Improvement of memory allocation requests.



(b) Improvement of memory deallocation requests.

Fig. 14: Performance improvement brought about by the spring core layer and the spring lazy layer, respectively.

concurrent scenarios. By dynamically adjusting the number of buddy domains based on actual system workload, Spring Buddy achieves a trade-off between system concurrency and fragmentation. Besides, Spring Buddy also predicts the subsequent request behavior of the process and dynamically adjusts the per-CPU buffer boundaries to further reduce the conflict of concurrent requests. In our experiments, Spring Buddy has the following advantages over std-Buddy and iBuddy. First, it can reduce the memory allocation/deallocation request latency by 71.47%, 93.20% on average in high concurrency cases, respectively. Secondly, the collaboration between core layer and lazy layer can reduce the contention cases by 68.83%, 33.87% on average, respectively.

Acknowledgment

We thank anonymous reviewers for their insightful comments. This work is partially sponsored by the National Key RD Program of China (No.2018YFB2101102), the Key RD projects in Sichuan Province (No. 2020YFSY0067), the Natural Science Foundation of China (NSFC) (No.61932014, 61972246), and Tencent Basic Platform Technology Project (Rhino Bird Special Research Plan).

References

[1] C. Kozyrakis *et al.*, "Server engineering insights for large-scale online services," *IEEE Micro*, vol. 30, pp. 8–19, 2010.
[2] H. Zhang *et al.*, "In-memory big data management and processing: A survey," *IEEE Trans. Knowl. Data Eng.*, vol. 27, pp. 1920–1948, 2015.
[3] A. Margaritov *et al.*, "Ptemagnet: fine-grained physical memory reservation for faster page walks in public clouds," in *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2021, pp. 211–223.
[4] R. Hood *et al.*, "Performance impact of resource contention in multicore systems," in *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 2010, pp. 1–12.
[5] S. Boyd-Wickizer *et al.*, "An analysis of linux scalability to many cores." in *OSDI*, vol. 10, no. 13, 2010, pp. 86–93.
[6] M. Frank *et al.*, "Parallelization, modeling, and performance prediction in the multi-/many core area: A systematic literature review," in *2017 IEEE 7th International Symposium on Cloud and Service Computing (SC2)*, 2017, pp. 48–55.
[7] Y. Cao *et al.*, "Dr dram: Accelerating memory-read-intensive applications," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018, pp. 301–309.
[8] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.
[9] H. Yun *et al.*, "Parallelism-aware memory interference delay analysis for cots multicore systems," in *2015 27th Euromicro Conference on Real-Time Systems*. IEEE, 2015, pp. 184–195.
[10] J. Huang *et al.*, "An evolutionary study of linux memory management for fun and profit," in *2016 USENIX Annual Technical Conference (USENIX ATC 16)*, 2016, pp. 465–478.
[11] R. Leite and R. Rocha, "A lock-free coalescing-capable mechanism for memory management," in *Proceedings of the 2019 ACM SIGPLAN International Symposium on Memory Management*, 2019, pp. 79–88.
[12] G. S. Brodal *et al.*, "Fast allocation and deallocation with an improved buddy system," *Acta Informatica*, vol. 41, pp. 273–291, 2005.
[13] R. Barkley and T. Lee, "A lazy buddy system bounded by two coalescing delays," *ACM SIGOPS Operating Systems Review*, 1989.
[14] R. Marotta *et al.*, "NBBS: A non-blocking buddy system for multi-core machines," *IEEE Trans. Comput.*, 2021.
[15] H. Park *et al.*, "iBuddy: Inverse buddy for enhancing memory allocation/deallocation performanceon multi-core systems," *IEEE Trans. Comput.*, vol. 64, pp. 720–732, 2014.
[16] E. D. Berger *et al.*, "Hoard: A scalable memory allocator for multithreaded applications," *ACM Sigplan Notices*, vol. 35, pp. 117–128, 2000.
[17] A. S. Tanenbaum and H. Bos, *Modern operating systems, 4nd Edition*. Pearson, 2015.
[18] L. Torvalds. Linux kernel source tree. https://github.com/torvalds/linux.
[19] W. Mauerer, *Professional Linux kernel architecture*. John Wiley & Sons, 2010.
[20] K. C. Knowlton, "A fast storage allocator," *Communications of the ACM*, vol. 8, pp. 623–624, 1965.
[21] K. C. Knowlton, "A programmer's description of l6," *Communications of the ACM*, vol. 9, pp. 616–625, 1966.
[22] B. Cranston and R. Thomas, "A simplified recombination scheme for the fibonacci buddy system," *Communications of the ACM*, 1975.
[23] I. P. Page and J. Hagins, "Improving the performance of buddy systems," *IEEE Trans. Comput.*, vol. 35, pp. 441–447, 1986.
[24] J. Herter *et al.*, "CAMA: A predictable cache-aware memory allocator," in *2011 23rd Euromicro Conference on Real-Time Systems*, 2011.
[25] M. Aigner *et al.*, "Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures," *ACM SIGPLAN Notices*, vol. 50, pp. 451–469, 2015.
[26] A. Prasad and K. Gopinath, "Prudent memory reclamation in procrastination-based synchronization," in *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS*, 2016, pp. 99–112.
[27] T. Liu *et al.*, "An optimized low-power optical memory access network for kilocore systems," *IEICE TRANSACTIONS on Information and Systems*, vol. 102, pp. 1085–1088, 2019.
[28] H. Zhao *et al.*, "Bandwidth and locality aware task-stealing for manycore architectures with bandwidth-asymmetric memory," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, pp. 1–26, 2018.
[29] C. Hanel *et al.*, "Vortex: Extreme-performance memory abstractions for data-intensive streaming applications," in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020, pp. 623–638.
[30] X. Wei *et al.*, "Fast in-memory transaction processing using rdma and htm," in *Proceedings of the 25th Symposium on Operating Systems Principles*, 2015, pp. 87–104.
[31] M. Zhang *et al.*, "Simpo: A scalable in-memory persistent object framework using nvram for reliable big data computing," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 15, pp. 1–28, 2018.
[32] Y. Wang *et al.*, "Towards memory-efficient allocation of cnns on processing-in-memory architecture," *IEEE Trans. Parallel Distrib. Syst.*, vol. 29, pp. 1428–1441, 2018.
[33] R. M. Hollander and P. V. Bolotoff. Ramspeed, a cache and memory benchmarking tool. http://alasir.com/software/ramspeed/.
[34] J. D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. https://www.cs.virginia.edu/stream/.
[35] A. Kopytov. sysbench. https://github.com/akopytov/sysbench.