iBuddy: Inverse Buddy for Enhancing Memory Allocation/Deallocation Performance on Multi-Core Systems

Heekwon Park, *Member, IEEE*, Jongmoo Choi, Donghee Lee, and Sam H. Noh, *Member, IEEE*

Abstract—We present a new buddy system for memory allocation that we call the lazy iBuddy system. This system is motivated by two observations of the widely used lazy buddy system on multi-core systems. First, most memory requests are for single page frames. However, the lazy buddy algorithm used in Linux continuously splits and coalesces memory blocks for single page frame requests even though the lazy layer is employed. Second, on multi-core systems, responses to bursty memory requests are delayed by lock contention caused by concurrent accesses of the multi-cores. The lazy iBuddy system overcomes the first problem by managing each page frame individually and coalescing pages only when an allocation of multiple page frames is requested. We devise the lazy iBuddy algorithm so that single page frame allocation can be done in O(1). The second problem is alleviated by dividing main memory into multiple buddy spaces and applying a fine-grained locking mechanism. Performance evaluation results based on various workloads on the XEON 16core with 32 GB main memory show that the lazy iBuddy system can improve memory allocation/deallocation time by up to 47 percent with an average of 35 percent compared with the lazy buddy system for the various configurations that we considered.

Index Terms—Dynamic memory manager, splitting and coalescing, Buddy algorithm

1 Introduction

Multi-core systems with large memory are now becoming the norm. We revisit the classic buddy memory management problem in view of these environmental changes. Consequently, a new dynamic memory manager, which we refer to as the *lazy iBuddy (inverse Buddy)* system, that is more effective than the widely used lazy buddy system on large memory, multi-core systems is presented.

The motivation behind the development of the lazy iBuddy system is based on three observations. First, the allocation and deallocation time show large variations due to the non-trivial splitting and coalescing overhead in the traditional buddy system. Second, most allocation requests are for single page frames. Third, lock contention has a significant effect on performance. These observations lead us to set up the following three goals; 1) avoid splitting and coalescing as much as possible, 2) perform efficiently for single page allocation requests, and 3) alleviate lock contention as best one can, while maintaining light overhead.

The lazy iBuddy system that we develop achieves the first two goals by managing each page frame individually. This totally eliminates splitting overhead, allowing single page allocation to be done in O(1). Coalescing overhead per se is eliminated as well for single page deallocation. However, for managing purposes, we make use of the notion of a buddy group, and this housekeeping incurs O(log n) overhead, where n is the number of page frames. Complexity-wise, this is the same as the standard buddy system. However, its execution is much more efficient as only simple management is involved. Finally, the time complexity of multiple page frame allocation and deallocation is O(m) where m is the number of requested page frames.

The third goal is achieved by introducing the notion of buddy space at the buddy layer and applying a lazy bypass technique. Buddy space is simply a unit of main memory that is used to manage main memory at the buddy layer by iBuddy. Dividing main memory into multiple buddy space units enables iBuddy to exploit a fine-grained locking mechanism, leading to enhanced parallel execution of multiple requests from multiple cores. The enhanced parallelism at the buddy layer allows the use of the lazy bypass technique that allocates/deallocates a page frame directly from/into the buddy layer without lazy layer intervention, that is, bypassing the lazy layer, to lessen the lazy layer management overhead.

We implement the lazy iBuddy system in the Linux kernel version 2.6.32 on the Intel XEON system. Using six benchmarks, we conduct various experiments that show that the lazy iBuddy system can improve memory allocation/deallocation time by up to 47 percent with an average of 35 percent compared with the traditional lazy buddy system. Furthermore, it can improve the predictability of memory allocation/deallocation requests by reducing the standard deviation of the response latency from around 1,400 to 250 cycles.

- *H. Park is with the Department of Computer Science, University of Pittsburgh, Pittsburgh, PA 30332. E-mail: parkhk@cs.pitt.edu.*
- *J. Choi is with the Department of Software Science, Dankook University, South Korea. E-mail: choijm@dankook.ac.kr.*
- *D. Lee is with the Department of Computer Science, University of Seoul, South Korea. E-mail: dhl_express@uos.ac.kr.*
- *S.H. Noh is with the School of Computer and Information Engineering, Hongik University, South Korea. E-mail: samhnoh@hongik.ac.kr.*

Manuscript received 08 Oct. 2012; revised 11 Nov. 2013; accepted 11 Dec. 2013. Date of publication 15 Jan. 2014; date of current version 11 Feb. 2015. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below. Digital Object Identifier no. 10.1109/TC.2013.2296049

The remainder of this paper is organized as follows. In the next section, we give the motivation behind this work. Then, in Section 3, we describe the design of the lazy iBuddy system concentrating on the key concepts that relate to the buddy layer. The lazy layer is discussed separately in Section 4. The implementation details of the system are described in Section 5. In Section 6, we present the performance evaluation results. Previous studies related to this work are examined in Section 7, and finally, a summary and the conclusions are presented in Section 8.

2 Motivation

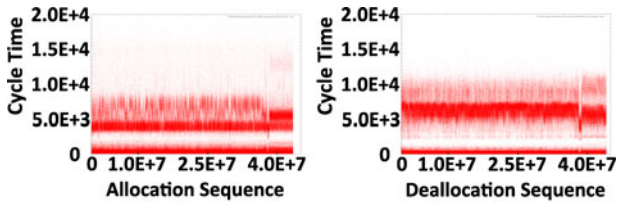
In this section, we first give a short review of the lazy buddy system. Then, we present the three observations that motivate this work.

2.1 Lazy Buddy

The standard buddy system is a well-known dynamic memory allocation/deallocation mechanism that tries to satisfy a memory request as suitably as possible [1], [2]. Basically, it manipulates a memory unit whose size is a power of two. When an allocation request for a memory of size s is issued, the buddy system splits a free memory unit into two sub-units, called buddies, recursively until one sub-unit satisfies the allocation request with the smallest size of 2^n such that $s \leq 2^n$. Then, it allocates one sub-unit, marking it as allocated (or used), while marking the other sub-unit as free. For a deallocation request, it coalesces the freed memory unit with its buddy recursively as long as its buddy is marked as free.

Research has been conducted to enhance the performance of the standard buddy system [3], [4], [5], [6], [7]. One such research is the lazy buddy system proposed by Barkley and Lee [3]. The idea is to divide the deallocation process into two steps; one step pushes a freed memory unit into the buddy area and marks it as free, while the other step coalesces the freed unit with its buddy, if possible. While the standard buddy system performs both steps on each deallocation request, the lazy buddy system performs only the first step and defers the second step until it becomes necessary. This deferring, hence the term lazy, allows the freed memory unit to be allocated again without paying the coalescing and splitting overhead. The lazy layer is essentially a cache area where deallocated page frames are temporarily kept to avoid coalescing and where page frame allocation requests are first attempted to be serviced to avoid splitting.

A variety of Unix-like systems make use of the lazy buddy system [8], [9]. For instance, the Linux kernel adopts the lazy buddy system and manages main memory using two layers, that is, the buddy layer and, in Linux terminology, the *per-CPU page list* as each CPU maintains its own lazy layer [6]. The size, that is, the number of page frames, of the lazy layer is controlled by a watermark. When the size drops below the low watermark a batch of page frames are brought in from the buddy layer, while when the size climbs over the high watermark a batch of page frames are released into the buddy layer. In the buddy layer, splitting and coalescing are always performed. Note that in Linux, since the lazy layer is maintained by each CPU independently, the lazy layer also helps in reducing lock contention among the CPUs.



(a) Allocation (b) Deallocation

Fig. 1. (a) Allocation and (b) deallocation time (cycles) for requests in the lazy buddy system. Each graph shows two distinct bands depending on whether requests are serviced at the buddy layer or not.

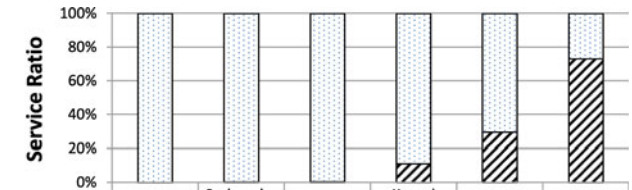
2.2 Observation 1: Splitting and Coalescing Overhead Are Non-Trivial

Fig. 1 shows the latency of allocation and deallocation requests in the lazy buddy system as we execute a Kernel Compile application on our 16 core experimental system running the Linux kernel version 2.6.32. (The details regarding the experimental environment will be elaborated on in Section 6.) The x-axis is the allocation/deallocation request in time sequence, while the y-axis represents the CPU clock cycles to serve each request.

From Fig. 1a, we observe that the allocation time can be clustered into two bands. One band, which runs just above, and along with, the x-axis, is a cluster of requests serviced quickly in at most 1,000 cycles with an average of 100 cycles. The other band represents requests serviced relatively slowly, in the 3,500 to 8,000 cycle range. Through closer analyses, we find that requests in the former band are those allocation requests that were serviced from the lazy layer, while the requests in the latter band are those serviced from the buddy layer. At the lazy layer, the operations needed to allocate a page frame are just simple list manipulations and some variable updates leading to the quick response. In contrast, in the buddy layer, recursive splitting of large memory units into smaller buddy units are required leading to relatively large overhead. Similarly, coalescing of the buddy units into larger buddy units at the buddy layer, and the lack thereof, results in a similar phenomena for deallocation requests as shown in Fig. 1b.

This observation implies that even though the lazy layer can satisfy some of the allocation/deallocation requests, there are still a considerable number of requests handled at the buddy layer. This is more evident in Fig. 2, where we show the number of allocation and deallocation requests serviced at the lazy layer and buddy layer while executing the Kernel Compile application and five other benchmarks. (Again, detailed descriptions of these benchmarks will be given in Section 6.) Fig. 2 shows that requests handled at the lazy layer are quite small, less than 1 percent, for the first three benchmarks. The other three benchmarks show relatively higher service ratio at the lazy layer, ranging from 11 to 73 percent. Even so, a significant number of requests are actually being serviced at the buddy layer as we can see from the numbers at the bottom of the figure.

The reason for such behavior is that allocation requests from these applications, especially the first three



	Stream	Sysbench - memory	Ramspeed	Kernel Compile	Dbench	Unixbench
Buddy Layer	1563462	2111527	31426172	84715042	70369094	402359140
Lazy Layer	3320	8466	161359	10575148	29511677	1085906322

Fig. 2. Ratio of requests serviced by the lazy and buddy layers. The bottom table shows the actual number of requests serviced at each layer.

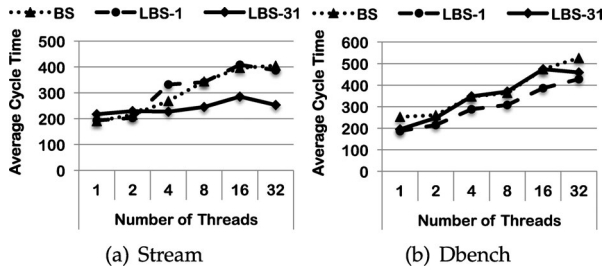


Fig. 4. Performance comparison of the Stream and Dbench applications with multithreading for two different types of lazy buddy systems.

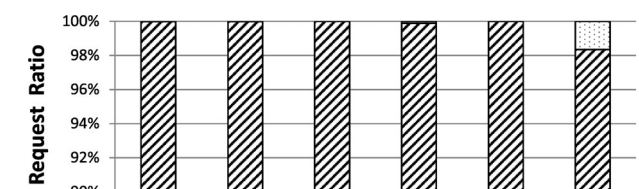
applications, are frequently made within a short time span. The consequence is that the lazy layer, having an insufficient number of page frames, is unable to service the sudden surge of requests. Hence, allocation requests are driven down to the buddy layer. Also, these applications have a tendency to deallocate page frames in bursts as well leading to an overflow of page frames from the lazy layer into the buddy layer. In other words, a significant number of memory requests are actually handled in the buddy layer incurring non-trivial splitting and coalescing overhead. This, in turn, leads to results as depicted in Fig. 1, where the allocation/deallocation time shows large variations. It is likely that this trend will be amplified as the capacity of main memory and the memory intensity of applications increase.

One could argue that the proportion of requests serviced at the lazy layer may be increased simply by increasing the size of the lazy layer. This may be true to some extent. However, when, eventually, the lazy layer runs out of page frames, the time to refill from the buddy layer with more page frames may even worsen the variation in service time.

Our conclusion from this observation is that the execution time of splitting and coalescing of the buddy system has a significant effect on performance, even when applying the lazy layer to defer such operations.

2.3 Observation 2: Most Allocation Requests Are for Single Page Frames

The second observation that motivates this study is the memory request size distribution as depicted in Fig. 3. This figure shows, for the aforementioned six benchmarks, that more than 98.4 percent of the requests are for 4 KB, that is, a single page frame. That is to say, essentially all allocation requests made by applications, libraries, and the Linux kernel are for single page frames, and requests for multiple page frames occur only very rarely. Such behavior is a natural outcome considering the characteristics of recent virtual memory systems that only construct mapping tables for large memory allocation requests, while the actual allocations are conducted on-demand through the page fault handling mechanism. Our conclusion from this observation is that we need to focus on optimizing single page requests in order to improve dynamic memory allocation/deallocation performance.



	Stream	Sysbench - memory	RAMspeed	Kernel Compile	Dbench	Unixbench
Multiple page	194	241	759	105812	334	24490375
Single page	1566588	2119752	31586772	95184378	99880437	1463775087

Fig. 3. Memory request size distribution. The bottom table shows the number of single and multiple page requests. Note the y-axis starts from 90 percent.

2.4 Observation 3: Lock Contention Plays an Important Role in Multi-Core Systems

The final observation is based on Fig. 4, which depicts the results of executing the Stream and Dbench benchmarks with varying number of threads (x-axis). (A full set of results for all the benchmarks are given in Table 1. The graphs in Fig. 4 are excerpts from this table.) This figure compares the average elapsed time to serve memory requests (y-axis) for the traditional buddy without the lazy layer, denoted BS, and two instances of the lazy buddy system. The first lazy buddy, denoted as LBS-31, is the default configuration in the Linux kernel where the batch size is 31. Recall that a batch refers to the page frames that are brought in or taken out to/from the lazy layer from/to the buddy layer when the watermark reaches the low/high watermark. The other, denoted LBS-1, is a configuration where the batch size is 1. In this case, since making a single page request to the buddy layer through the lazy layer would simply be extra overhead, we service the requests directly from the buddy layer bypassing the lazy layer. Note that this bypassing happens only when the watermark is reached. Normally, requests are serviced from the lazy layer.

Let us first compare the performance of BS and LBS-31 for Stream as depicted in Fig. 4a. The key difference between BS and LBS-31 is that the lazy layer acts as a buffer for LBS-31. However, recall from Fig. 2 that, for Stream, very few requests are serviced by the lazy layer and that most requests are actually serviced by the buddy layer. Hence, the performance of the two should not differ much. This is what is observed for the thread 1 and 2 cases. However, beyond four threads, we see a wide performance gap between BS and LBS-31 with LBS-31 doing considerably better than BS. So, where is this performance gap coming from? We will refer to this question as Q1 (for Question 1) as we answer this question later.

Let us now move on to Fig. 4b that depicts the performance of Dbench. Unlike Stream, recall from Fig. 2 that

with Dbench a considerable number of requests are serviced at the lazy layer. This is to say, if we compare BS and LBS-31, we should expect LBS-31 to perform much better as the lazy layer is acting as a cache layer. However, in fact, we see that the performance between the two does not differ much. This is counterintuitive; what is happening to all the benefits reaped by the lazy layer? This is the second question that we raise, which we refer to as Q2 (for Question 2).

To answer Q1 and Q2 we contend that in the lazy buddy system there are two different forces acting against each other. One is the overhead involved in managing the lazy layer. This overhead involves bringing a batch of page frames from the buddy layer into the lazy layer before the actual allocation (and vice versa, that is, draining pages from the lazy layer to the buddy layer for deallocation). The other, counteracting force, is the lock contention involved in accessing the buddy layer, which is a shared resource. Lock contention should be affected by various factors such as the number of requests, the intensity of the requests, and the number of threads vying for the lock.

As such, regarding Q1, we contend that the performance improvement brought about by LBS-31 is due to the alleviated lock contention at the buddy layer. With LBS-31, less requests are made to the buddy layer as requests are bundled into a large batch size. Hence, lock contention remains relatively light for LBS-31 even as the number of threads vying for the lock increases.

This reasoning is supported from the results of LBS-1. Compare LBS-1 and BS, whose only difference is the existence of the lazy layer. We see that the existence of the lazy layer has only a slight effect meaning that the lazy layer management overhead is very light. Now, compare LBS-1 and LBS-31, whose main difference is the batch size, which affects the number of requests made to the buddy layer and hence, lock contention. We see that the performance gap between these two are similar to that of BS and LBS-31.

Now, regarding Q2, we contend that the effect of the lazy layer of LBS-31 disappears because of the management overhead incurred by the lazy layer. For Dbench, lock contention is not a factor in performance as we see the performance difference among BS, LBS-31, and LBS-1 remains relatively constant for different thread counts as shown in Fig. 4b.

The key factor affecting performance for Dbench is the lazy layer management overhead. First, observe the performance gap between BS and LBS-1, which is evidence that the lazy layer has a positive effect on performance, as the only difference between BS and LBS-1 is the existence of the lazy layer. However, this benefit disappears for LBS-31. Now, observe the performance difference between LBS-1 and LBS-31. Recall, LBS-1 bypasses the lazy layer when the high/low watermark is met. Aside from the batch size (which has only minimal effect for Dbench), this is the only difference between LBS-1 and LBS-31. This tells us that LBS-31, which never bypasses the lazy layer, is being negatively affected by this management overhead.

Based on these observations, our conclusion is that lock contention and lazy layer management can significantly affect the performance of the lazy buddy system. The iBuddy system that we present in the following considers both these issues.

3 Key Concepts of iBuddy

The lazy iBuddy system consists of the lazy layer and the buddy layer, just like the traditional lazy buddy system. In this section, we focus on the core allocation/deallocation algorithms at the buddy layer, and issues related to the lazy layer will be addressed in the next section. For simplicity, we refer to the buddy layer part of the lazy iBuddy system as the iBuddy system and that of the lazy buddy system as the (traditional) buddy system.

The core algorithms of the iBuddy system utilize two key concepts. One is managing each page frame individually with buddy group manipulation and the other is dividing memory into several buddy spaces and applying a fine-grained locking mechanism.

3.1 Individual Page Frame Management with Buddy Group Manipulation

Recall that the majority of the requests from applications are single page frame requests and that a significant portion of these requests are serviced at the buddy layer. Hence, the design goal for our lazy iBuddy system is to handle single page frame requests efficiently at the buddy layer while possibly sacrificing performance for larger requests.

To describe the core algorithms of the iBuddy system, we contrast these algorithms with those of the traditional buddy system. We will use Fig. 5 as a walk-through example. We assume that physical memory consists of 16 page frames and that some of these are *free*, depicted as white rectangles, while others are *allocated*, depicted as shaded rectangles, as shown in Fig. 5a.

As shown in Fig. 5b, the traditional buddy system uses two data structures, that is, the bitmap and a set of free lists. The bitmap and free lists are constructed in multiple levels labeled $2^0, 2^1, ..., 2^i, ...$ At level 2^i, there are 2^i number of consecutive page frames that are treated as a single unit, which we refer to as a *buddy group*. For example, at level 2^3 of Fig. 5b, eight consecutive pages from page 0 to page 7 form a buddy group. Each bit in the bitmap is used to indicate whether the corresponding buddy group is allocated or free, while the free lists are used to maintain the actual free buddy groups. In Fig. 5b, there are three buddy groups: pages 0-7, pages 8-9, and page 11. The three bits with value 1 in the bitmap indicate that these buddy groups are free.

Fig. 5c, in contrast, shows the bitmap and free lists used for the iBuddy system. The following three points summarize where the iBuddy system differs from the buddy system with each point being clarified through subsequent examples.

- The free list is managed per page frame, that is, each page frame is an independent entity in the iBuddy system. This is in contrast to the buddy system where the free list is managed per buddy group, which is composed of one or more page frames. (Note the free lists portion of Figs. 5b and 5c.)
- Each bit in the bitmap of the iBuddy system indicates not only the state of the corresponding buddy group, but also that of the corresponding page frame. In the buddy system, the bit indicates the state of the corresponding buddy group only.

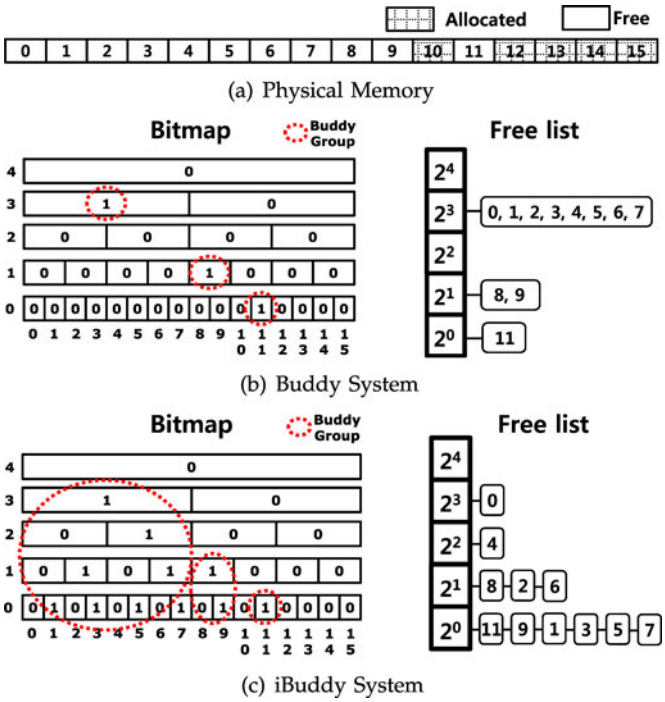


Fig. 5. Comparison between the traditional buddy system and our iBuddy system. (a) Depicts the state of the physical memory with the white box denoting a free page frame, while the shaded box refers to an allocated page frame. (b) Shows how the traditional buddy system would manage the memory state of (a), with each 1 in the bitmap representing a buddy group, which may be composed of one or more pages, as represented in the free list. This is in constrast to (c), which shows how memory state of (a) would be managed with the iBuddy system.

- When allocating in the iBuddy system, search for a free buddy group starts from the highest level, working downward. In the buddy system, search works upward starting from the lowest level. This is one reason behind the name iBuddy (inverse Buddy).

3.1.1 Allocation of a Single Page Frame

Let us now see how the two systems differ when allocating memory. Assume that the starting state is as given in Figs. 5b and 5c, and that two consecutive requests for a single page arrive.

In the buddy system, search starts from level 0. Hence, page frame 11 is found and allocated. On the second allocation request the buddy system again starts to search for memory from the lowest level, and this time finds free memory at level 1. Thus, the algorithm splits the buddy group of page frames 8-9, and allocates one page frame and marks the other page frame as free. The data structures after these allocations are depicted in Fig. 6a.

The iBuddy system, in contrast, starts to search from the highest level. As a result, the system detects page frame 0 at level 3 of the free lists. The corresponding bit of the detected page frame is 1, which has dual meaning. One is that the buddy group of page frames 0-7 is free so that it can be allocated when page frames of aggregate size 2^3 is requested. The other meaning is that page frame 0 is free and can be allocated without any modification to the lower levels. Since the first request was for a single page frame, iBuddy allocates page frame 0 and clears the bit at level 3.

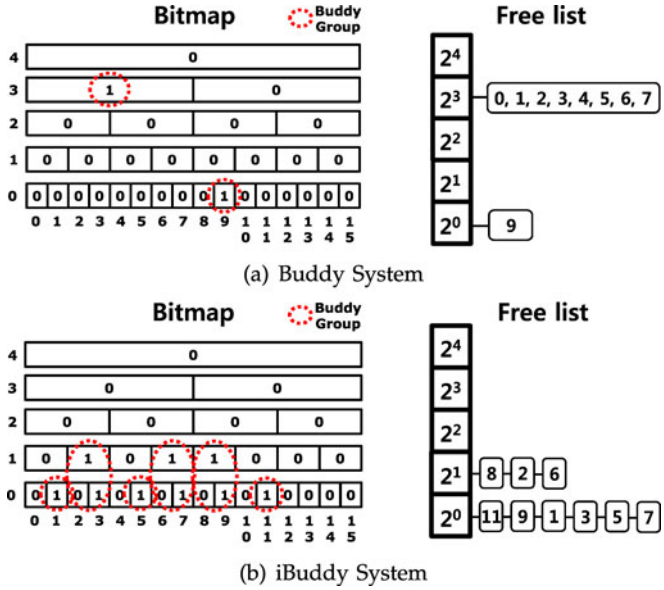


Fig. 6. Changes from Fig. 5 after two single page frame requests are allocated: (a) for the buddy system and (b) for the iBuddy system.

For the second request, the algorithm allocates page frame 4, which is the page frame at the highest level with bit set to 1 at this point. Note that no splitting is involved. The data structures after these allocations are depicted in Fig. 6b.

Note here that the key benefit of the iBuddy system is that it always allocates a single page frame in O(1). This is in contrast to O(log n) of the buddy system that requires recursive splitting operations. Also, note that by keeping track of the last allocated level, the iBuddy system can locate the highest level with free page frames without scanning from the top level.

3.1.2 Deallocation of a Single Page Frame

Let us now consider deallocation. Again, starting from Fig. 5, assume that page frame 10 is released.

The traditional buddy system starts by putting the released page frame at the lowest level and checking if coalescing with its buddy is feasible. In our example, the released page frame is coalesced with page frame 11, which is again coalesced with page frames 8-9, leading to a new buddy group composed of page frames 8-11. This results in the 1 bits at level 0 and level 1 being reset to 0, and the appropriate bit at level 2 being set to 1. The result of releasing page frame 10 after coalescing is depicted in Fig. 7a, where we have two buddy groups, one of eight page frames and the other of four page frames. The corresponding bits of the buddy groups are set to 1.

The action sequence for the iBuddy system is similar to the traditional buddy system. It starts by putting the released page frame at the lowest level and checking if it can push up the released page frame to a higher level. This is done by checking on its neighbor page frame to see if a larger buddy group can be formed, just as it is done for the buddy system. However, instead of coalescing when it is possible, the released page frame is simply pushed up a level. Then, the process is repeated until it is no longer possible. When it reaches this level, the bit for this buddy group is set to 1 indicating that this buddy group is available for service.

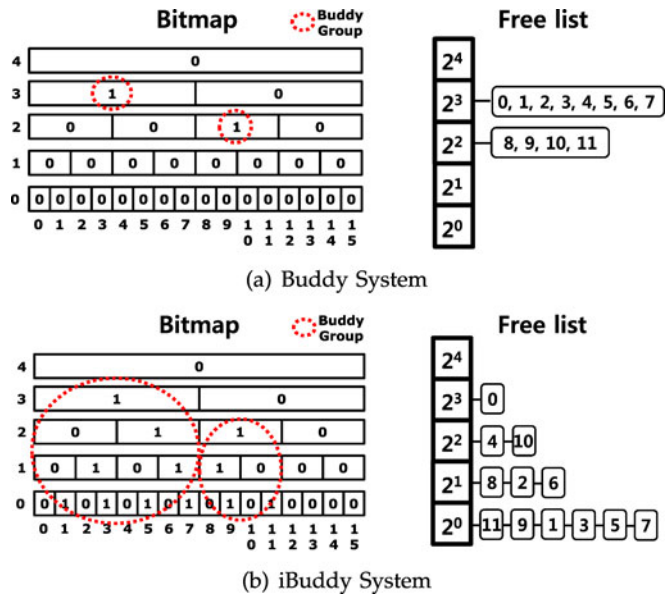


Fig. 7. Changes from Fig. 5 after page frame 10 is released: (a) for the buddy system, and (b) for the iBuddy system.

In our example, as page frame 10 is being released, it first checks the bit setting on the right, that is, of page frame 11. Seeing that it is 1, it moves up to level 1 and checks the bit setting on the left. Seeing that it is 1, it moves up another level and checks the bit on the right. Here, the bit is 0, so it stops at that level setting its own bit to 1. The result of this sequence is depicted in Fig. 7b. Note that the only change that has occurred is at level 2, where the single page frame 10 has been added to the free list and the third bit at this level has been set to 1.

Note that the time complexity of deallocation for the iBuddy system is O(log n), where n is the total number of page frames in the buddy system, just like the traditional buddy system. Just like coalescing, this pushing up (or ascending) of a page frame is carried out as long as its buddy is free. However, page frame ascending is computationally much less demanding than coalescing since ascending requires only one insertion while coalescing may require multiple insertions and deletions in the lists.

3.1.3 Allocation/Deallocation of Multiple Page Frames

For the traditional buddy system, whether the memory request is for a single page frame or multiple page frames is immaterial. Allocation and deallocation starts from the lowest level moving up, and the running time is O(log n) for both cases.

As the iBuddy system is designed to handle the common single page frame requests efficiently, price is paid for the occasional multiple page requests. Since every page frame is independently linked in the free list, multiple page frame requests in the iBuddy system require actual splitting or coalescing. As an example, consider an allocation request for eight page frames starting from the state in Fig. 5. In the buddy system, a buddy group of 0-7 is deleted from the free list of 2^3 and then, allocated. For the iBuddy system, using the bitmap information, the fact that there is a buddy group of page frames 0-7 available for allocation can be checked. However, it cannot allocate them altogether. Instead, each page frame 0-7 is, respectively, deleted from its free list, its corresponding bit cleared, and then, finally allocated for the request. Similarly, even for a multiple page frame deallocation request, each page frame is individually inserted into its free list and its corresponding bit set. Hence, in the iBuddy system, the running time for allocating and deallocating multiple page frame requests is O(m) and O($m$log n), where m is the number of requested page frames and n is the total number of page frames in the buddy system.

The basic philosophy behind the iBuddy system is to manage each page frame individually since a vast majority of memory requests are for single page frames as discussed in Section 2.3 (using Fig. 3). In other words, splitting is carried out as soon as possible and coalescing is delayed until it is not avoidable (say, by multiple page frame requests). Note that even though the iBuddy system manages each page frame individually, it can find consecutive page frames efficiently through buddy group manipulation. Also, note that splitting is carried out upon deallocation requests, while coalescing is done at allocation time, completely opposite of the traditional buddy system. This is another reason behind the name iBuddy.

3.2 Multiple Buddy Space Management

To alleviate lock contention among multiple cores, we incorporate the conventionally well-known fine-grained locking mechanism into our iBuddy system as depicted in Fig. 8. In the traditional buddy system (Fig. 8a, the buddy layer is treated as a single shared resource controlled by a single lock. In contrast, in the iBuddy system (Fig. 8b, we introduce the notion of buddy space, which is simply a unit of memory. For fine-grained locking, the buddy layer is divided into multiple buddy spaces controlled by independent locks. At any time, each core is assigned its own buddy space, and allocation and deallocation of page frames are done within the assigned buddy space. If a core uses up its assigned buddy space, the iBuddy system assigns another unassigned buddy space to the core. This independent mapping between a core and a buddy space leads to reduced lock contention among the cores. We allow two or more cores to share a buddy space when necessary. This may happen when buddy spaces run out or when page frames must be deallocated into a previously assigned buddy space. However, this situation occurs very rarely.

In addition to enhancing the parallelism among multiple cores, employing the notion of multiple buddy space brings about two positive effects. The first is that it can curtail fragmentation in the iBuddy system. One concern of the iBuddy system is that by allocating page frames from the highest level, memory may become fragmented. This is exemplified in Fig. 6 where the buddy system has two buddy groups while iBuddy has six buddy groups. In effect, iBuddy behaves like the worst-fit memory allocation policy that has a drawback of producing numerous small fragments and incurring a shortage of large consecutive memory. Now, with multiple buddy space management, since allocation is carried out from a particular buddy space until page frames run out, fragmentation occurs only within a buddy space not affecting other buddy spaces that may contain larger consecutive page frames.

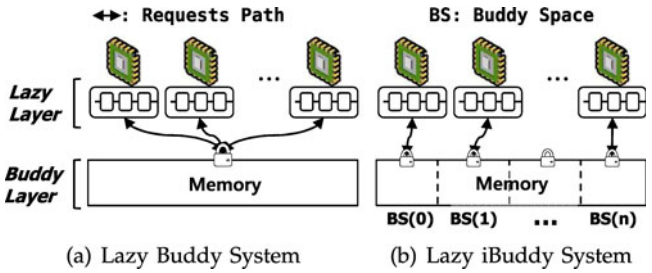


Fig. 8. (a) In the lazy buddy system the buddy layer is a single resource controlled by a single lock, while (b) in the lazy iBuddy system, the buddy layer is partitioned into buddy space units and each buddy space is assigned to a core for allocation and deallocation.

The other positive effect of multiple buddy space management is that it allows us to choose a small batch size when we integrate the lazy layer into the iBuddy system. This is so as lock contention is alleviated through the use of the buddy space notion. This issue will be elaborated on in the next section as it is related to the lazy layer.

Note that the notion of buddy space management is independent of the buddy system. Hence, the traditional buddy system may also take advantage of the benefits of the notion of buddy space.

4 The Lazy Layer and Lazy Bypass

Until now, we have discussed issues related to the buddy layer. In this section, we turn our focus to the lazy layer and discuss issues such as the effect of the batch size on performance and the lazy bypass technique.

Fig. 8a shows the overall structure of the lazy buddy system that consists of the lazy and buddy layers. The lazy layer is managed by each core separately using a doubly linked list. The size of the lazy layer is controlled by high and low watermarks. When the number of page frames in the lazy layer drops below the low watermark, a batch of page frames are brought in from the buddy layer; when the number goes beyond the high watermark, a batch of page frames are released into the buddy layer. When the number is between the two watermarks, allocation and deallocation requests are handled in the lazy layer, deferring splitting and coalescing at the buddy layer.

In the current implementation of the Linux kernel version 2.6.32, the batch size is calculated using the formula min (((size_of_memory ÷ 4 MB) − 1) , 31) [8]. Also, the high watermark and low watermark is set to $6 \times batch\ size$ and 0, respectively. Mauerer comments that these values have been set empirically to reduce the number of batch transfers between the lazy and buddy layers and to minimize cache aliasing effects for most system loads [8]. Our experimental system has 32 GB of main memory, hence the batch size, high watermark, and low watermark are 31, 186, and 0, respectively.

Initially, when we started to implement the lazy iBuddy system, we adopted the same lazy layer management configuration used by the traditional lazy buddy system. However, during the implementation, we found the current batch size value to be the cause of non-trivial lazy layer management overhead as discussed in Section 2.4 (using Fig. 4).

To mitigate this overhead, we introduce the lazy bypass technique in the lazy iBuddy system. The lazy bypass technique is a technique that simply bypasses the lazy layer upon allocation and deallocation for the buddy layer. Leaving the high and low watermarks unchanged (i.e., 186 and 0 in our experimental system), we change the batch size from 31 to 1. Then, when the number of page frames in the lazy layer becomes lower/higher than the low/high watermark, a page frame is allocated/deallocated directly from/into the buddy layer without involving the lazy layer. Note that the technique can lessen the lazy layer management overhead by removing the insertions/deletions of a batch of page frames into/from the lazy layer before the actual allocation/deallocation.

Note that the lazy bypass technique is feasible because of the two features introduced through the iBuddy system. The first is allowing single page frame requests to be serviced efficiently in the buddy layer. Supporting this in the traditional buddy system may be difficult due to the splitting and coalescing that happens at the buddy layer. The other feature of the iBuddy system that makes lazy bypass possible is the multiple buddy space management scheme. By dividing the buddy layer into multiple buddy spaces and assigning each of them to a particular core, most individual page frame requests serviced at the buddy layer may be done without lock contention among the cores.

5 Implementation Details

In this section, we describe the implementation details of the lazy iBuddy system. We present the main data structures and procedures for implementing the lazy iBuddy system and discuss some implementation issues such as the buddy space size and the buddy space selection policy.

The lazy iBuddy system employs two data structures, *lazy_list* and *free_area*. *Lazy_list* is used for the lazy layer, managing page frames using a doubly linked list and the high and low watermarks. *Free_area* is used for the buddy layer containing three main fields: an array of *free_list*s, *bit_map*, and *next_allocation_level*.

Each *free_list* is responsible for the management of each level in the buddy layer as shown in Fig. 5. Since the Linux kernel sets the highest level at 2^{10}, there are 11 *free_list*s for managing levels from 2^0 to 2^{10} (in other words, the size of the array of the *free_list*s is 11). This setting implies that the maximum number of page frames that can be allocated for a single request is bound to 2^{10}, that is, 4 MB in our experimental system where the page frame size is 4 KB. *Bit_map* is used for bit manipulation as shown in Fig. 5. Finally, *next_allocation_level* is an index indicating the current highest *free_list* that has free page frames allowing allocation to be conducted without scanning from the highest *free_list*.

One implementation issue is the size of the buddy space in the multiple buddy space management scheme. If we reduce the size of the buddy space, the number of buddy spaces will increase. This will increase parallelism resulting in reduced lock contention. Also, internal fragmentation will be reduced. Hence, a smaller buddy space size may be preferable. However, if the size becomes too small the number of buddy spaces may be so large as to incur considerable processing overhead. Hence, we may need to balance our choice carefully. In this study, we leave this issue for future work and simply set the buddy space size to 4 MB. This

value is the minimum buddy space size as 4 MB is the maximum allocation size. Consequently, the number of buddy spaces is 8,192 in our experimental environment that has 32 GB main memory. We also have the same number of *free_area*s since each *free_area* manages each buddy space in our current implementation.

Algorithm 1 shows the pseudo code for allocation in the lazy iBuddy system. It first checks the number of page frames requested. For a single page frame request, it tries to service the request at the lazy layer. When there is a free page frame, allocation is carried out by deleting a page frame from the *lazy_list* and returning that page. Otherwise, it goes into the buddy layer.

Algorithm 1 Allocation for lazy iBuddy system

```
1:  procedure __ALLOC_PAGES(sz)
2:      lazy_list ← get lazy_list of current core
3:      if sz == 4KB and lazy_list is NOT empty then
4:          delete page from lazy_list
5:          Return ptr of page
6:      else
7:          free_area ← get buddy space assigned for this core
8:          if no page satisfies this request then
9:              free_area ← get new buddy_space
10:         end if
11:         lock free_area
12:         get a page (or pages) from free_area
13:         adjust next_allocation_level if necessary
14:         unlock free_area
15:         Return ptr of the page (or the first_page of the multiple pages)
16:     end if
17: end procedure
```

At the buddy layer, it initially verifies that the assigned buddy space related to the requesting core has enough page frames to satisfy the requested size. If not, it releases the buddy space into the unassigned buddy space pool, selects a new buddy space, and assigns the new buddy space to the core. Finally, allocation is carried out from the assigned buddy space by deleting page frames from the *free_list*s, manipulating the corresponding bits in the *bit_map*, and adjusting the *next_allocation_level* if necessary. Note that the code is rather simple compared to the traditional lazy buddy system since code for bringing page frame batches from the buddy layer and inserting them into the lazy layer before the actual allocation is absent.

Algorithm 2 shows the pseudo code for deallocation. For a single page frame deallocation request, it also attempts to keep the page in the lazy layer. If this is not possible (since the number of page frames is at the high watermark) or the request is for multiple page frames, it handles the request directly at the buddy layer.

Algorithm 2 Deallocation for lazy iBuddy system

```
1:  procedure __FREE_PAGES(ptr of first_page, sz)
2:      lazy_list ← get lazy_list of current core
3:      if sz == 4KB and lazy_list is NOT full then
4:          insert page to lazy_list
5:      else
6:          free_area ← get buddy space from first_page
7:          lock free_area
8:          put a page (or pages) into free_area
9:          adjust next_allocation_level if necessary
10:         unlock free_area
11:     end if
12: end procedure
```

Another implementation issue is the policy for selecting a new buddy space during allocation. In the current implementation, we utilize a simple policy that can reduce fragmentation as much as possible. Each buddy space in the iBuddy system is in one of four states: assigned, unassigned with all frames used, unassigned with partial frames used, and unassigned with all frames freed. The policy chooses an unassigned with partial frames used buddy space first. If not possible, it chooses an unassigned with all frames freed buddy space.

6 Performance Evaluation

In this section, we first describe the setup of the experiments and the six benchmarks that we use as workloads for the experiments. We concentrate on the memory management scheme showing how iBuddy differs, performance-wise, from the buddy system. Finally, we discuss the fragmentation issue that may be of concern as the iBuddy algorithm focuses on optimizing single page allocation/deallocation.

6.1 Experimental Environment

Our experimental system is an IBM x3650 M2 server system that consists of two Intel XEON x5570 quad core processors (when we turn on the hyperthreading capability, each processor can have at most eight threads), 32 GB DDR3 main memory and eight 2.5 inch SAS disks of 450 GB capacity each. On this hardware platform, we implement the lazy iBuddy system on the Linux kernel version 2.6.32.

To compare the performance of lazy iBuddy with the traditional lazy buddy system, we use the following six benchmarks representing a variety of characteristics. The first three benchmarks, Stream, Sysbench-memory, and Ramspeed are memory intensive benchmarks, and the performance metric that these benchmarks report is the bandwidth in MB/s units.

- *Stream*. This benchmark is a synthetic benchmark used for measuring sustainable memory bandwidth in high performance systems [10]. In our experiments, the memory size is set to 3 GB, incurring roughly 1.5×10^6 memory requests.
- *Sysbench-memory*. This benchmark is a memory allocation and transfer speed benchmark [11]. To use the benchmark, one needs to specify the memory block size, which limits the maximum memory size used to three times the memory block size. In our experiments, the memory block size is set to 4 GB, incurring roughly 2.1×10^6 memory requests.
- *Ramspeed*. This benchmark is a utility to measure cache and memory performance of computer systems [12]. Of the 18 sets of workloads in this benchmark, we use the representative six sets that incur both read and write operations, resulting in roughly 3.2×10^7 memory requests.

The other three benchmarks are the Kernel Compile (application) and Unixbench benchmarks, which are CPU intensive, and Dbench, which is I/O intensive. For its performance metric, the Kernel Compile benchmark reports execution time in seconds, Dbench reports bandwidth in MB/s, and Unixbench reports a benchmark index score; higher index scores represents better performance.

TABLE 1
Average Elapsed Time for Memory Requests (Cycles)

	Stream						Sysbench-memory						Ramspeed						Kernel Compile						Unixbench						Dbench						Average of Six Benchmarks
	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	
BS	190	215	268	344	396	405	223	246	230	274	321	296	225	235	435	529	956	1154	306	292	296	358	482	515	481	508	555	648	990	1237	253	261	347	364	474	526	440
LBS-1	193	203	332	341	408	387	231	233	230	275	292	278	223	237	396	517	1044	1141	296	286	301	363	490	519	318	269	274	287	341	348	187	215	288	308	386	428	357
LBS-31	217	229	227	245	285	253	262	247	242	291	314	295	233	243	273	346	468	546	303	293	302	341	417	431	347	280	287	289	318	323	195	248	348	370	474	459	312
iS	106	111	116	162	190	169	170	153	152	171	182	177	152	154	222	276	369	410	207	192	200	223	290	307	387	340	346	351	403	423	160	197	265	292	365	393	244
LiS-1	117	121	142	152	206	170	154	145	144	164	174	180	148	159	209	248	349	393	194	184	186	206	262	268	276	212	219	218	241	240	113	146	201	219	284	287	204
LiS-31	151	189	176	230	244	245	181	199	182	179	215	214	185	194	233	300	421	504	242	230	233	262	308	328	313	249	255	254	269	267	139	177	258	283	351	358	251
PIR(%)	46	47	37	38	28	33	41	41	40	44	45	39	36	35	23	28	25	28	36	37	38	40	37	38	20	24	24	25	24	26	42	41	42	41	40	37	35%

BS: Standard Buddy System LBS-1: Lazy Buddy System(Batch Size = 1) LBS-31: Lazy Buddy System(Batch Size = 31)
iS : iBuddy System LiS-1 : Lazy iBuddy System(Batch Size = 1) LiS-31 : Lazy Buddy System(Batch Size = 31)
PIR : Performance Increase Ratio (between LBS-31 and LiS-1)

- *Kernel compile*. This benchmark builds a new kernel image by compiling the Linux kernel version 2.6.32 and by linking the compiled objects. The total size of the kernel source is roughly 422 MB and around 9.5×10^7 memory requests are triggered during the execution.
- *Dbench*. This benchmark is a file system benchmark, included in the dbench suite, generating a workload similar to the commercial Netbench benchmark [13]. This benchmark is now considered the de-facto standard for generating load on the Linux VFS. We use the default setting that triggers around 9.9×10^7 memory requests.
- *Unixbench*. This benchmark is a general-purpose benchmark designed to evaluate the performance of a Unix-like system, integrating CPU and file I/O tests as well as system behavior under various user loads [14]. We use the default setting that issues roughly 1.5×10^9 memory requests.

6.2 Summary of Performance Results

Table 1 presents the average elapsed time for memory requests measured in this experiment. We have executed the benchmarks with various numbers of threads, ranging from 1 to 32, under the six different buddy systems, namely, the standard buddy system (that does not employ the lazy layer), lazy buddy system with batch size 1 (denoted LBS-1), lazy buddy system with batch size 31 (denoted LBS-31), the iBuddy system (that does not employ the lazy layer), lazy iBuddy system with batch size 1 (denoted LiS-1), and lazy iBuddy system with batch size 31 (denoted LiS-31). The results for each of these systems are presented in each of the rows *BS* through *LiS-31* in Table 1. Note that the buddy system that is used in current Linux kernels is LBS-31, while the system with all the features that we propose is LiS-1. Hence, the performance difference of these two systems are presented in the last row.

From Table 1, we observe the following:

- Among the three buddy systems based on the traditional buddy algorithms, LBS-31 shows the best performance. Specifically, compared with the standard buddy system, it lowers the average elapsed time for handling memory requests from 410 cycles to 312 cycles. Also, compared with LBS-1, it reduces the time from 357 cycles to 312 cycles. This shows that the current batch size choice made for the Linux kernel is appropriate.
- Among the three buddy systems based on our proposed iBuddy algorithm, LiS-1 shows the best performance. It handles memory requests with an average elapsed time of 204 cycles, while LiS-31 and the iBuddy system takes 251 and 244 cycles, respectively.
- When we compare our proposed system LiS-1 with the current Linux implementation LBS-31, the former outperforms the latter by up to 47 percent with an average of 35 percent.

6.3 Detailed Performance Analysis

In this section, we quantify the effect of the proposed components of the iBuddy system, that is, single page frame management, buddy space, and lazy bypass. We also consider the effect of the batch size and the predictability of allocation and deallocation time.

6.3.1 Effect of iBuddy, Buddy Space, and Lazy Bypass

Three features affect the performance of the lazy iBuddy system. The first is the iBuddy algorithm that makes use of single page frame management enabling efficient allocation and deallocation for the common case. This, however, may lead to higher lock contention leading to degraded performance, especially with multithreading and multi-cores. This limitation is overcome with the introduction of the second feature, that is, buddy space management. Finally, lazy layer overhead is reduced with lazy bypass.

The results for each benchmark in Fig. 9 is the average of each benchmark executed with 1, 2, 4, 8, 16, and 32 threads taking into account only the memory allocation and

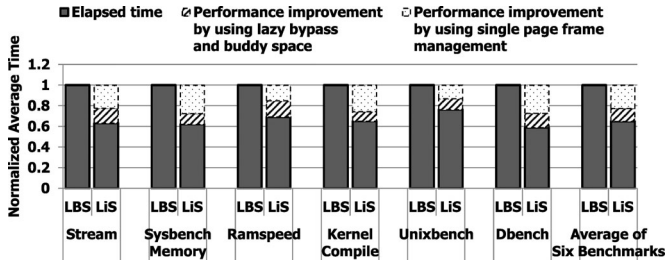
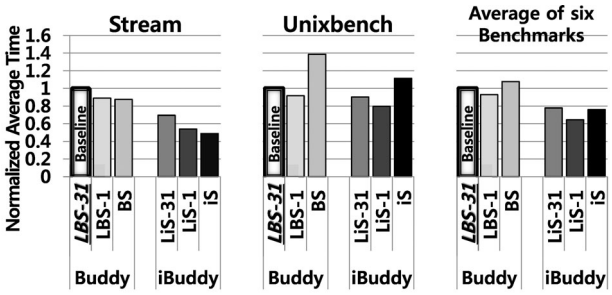
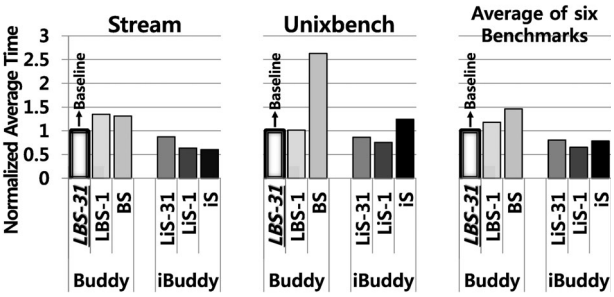


Fig. 9. Performance improvement brought about by the iBuddy algorithm, that is, single page frame management and the lazy bypass and buddy space components of the lazy iBuddy system.



(a) Performance effects for the single thread case.



(b) Performance effects for the multiple threads case(average of from 2 threads to 32 threads).

Fig. 10. Performance effects of the lazy layer service ratio.

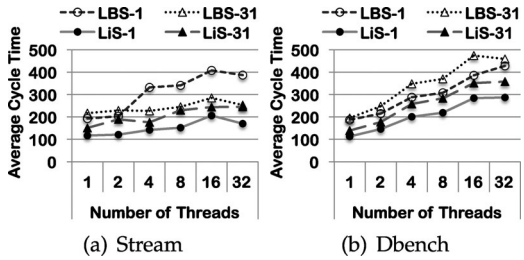
deallocation portion of the benchmark performance. Over the six benchmarks, an average of 35 percent performance improvement is observed. Among these 20 percent is due to the iBuddy algorithm component (that is, exploiting single page frame allocation/deallocation), while the rest is due to the lazy bypass and buddy space components.

On a side note, performance improvement due to the lazy bypass component seems to imply total elimination of the lazy layer. Though we cannot discuss this matter in detail due to space limitations, our experimental results (that we do not present here) indicate that this is not true. The lazy layer still has positive effects on performance as it is exploited when memory accesses are not bursty, and hence, still remains in our design.

6.3.2 More on Lazy Layer, Lazy Bypass, and Lock Contention

Here, we discuss how the service ratio of the lazy layer affects performance. We choose two representative benchmarks, Stream and Unixbench, that has the lowest and the highest service ratio at the lazy layer, respectively, and depict the normalized average elapsed times for the two benchmarks under the six buddy systems. Also, we depict the average of all six benchmarks as well.

Let us discuss the single thread case of Fig. 10a first. For the Stream benchmark buddy systems without the lazy layer (for both the buddy system and the iBuddy system) perform better than the others. Note that the iBuddy system without the lazy layer even performs better than the lazy iBuddy system with batch size 1 that employs lazy bypass. This is to say that the lazy layer management overhead is quite significant that if only a small number of memory requests are serviced at the lazy layer the gains obtained here may simply be overshadowed by the management overhead.



(a) Stream (b) Dbench

Fig. 11. Performance of lazy buddy with batch size 1 (LBS-1) and 31 (LBS-31) and lazy iBuddy with batch size 1 (LiS-1) and 31 (LiS-31) as the number of threads is increased.

On the other hand, for the Unixbench benchmark the buddy systems without the lazy layer results in the worst performance. This is to say that the caching effect of the lazy layer has a significant effect on the performance. Note that for this benchmark both LBS-1 and LiS-1 that utilize the lazy bypass technique results in the best performance. Overall, when we consider the average results of the six benchmarks, the results show that the lazy bypass technique is effective.

Now, let us consider the multiple threads case of Fig. 10b. For the Unixbench benchmark, the overall trend of the results of Fig. 10b are similar to those of Fig. 10a. This is to say that the lazy layer has a significant effect when the service ratio at the lazy layer is high. For the Stream benchmark, however, there is a slight difference. Among the three buddy systems that use the traditional buddy algorithms (denoted as LBS-31, LBS-1 and BS in the figure), LBS-31 shows the best performance. This was not the case for the single thread case, where the system that did not employ the lazy layer performed the best. This shows that the lazy layer plays a role in reducing lock contention among the multiple threads for the traditional buddy system. However, for the iBuddy systems (denoted as LiS-31, LiS-1 and iS in the figure), since lock contention is already considerably reduced by buddy space management, the lock contention reduction at the lazy layer has only little effect.

6.3.3 Effect of Batch Size

A practical issue that must be considered for actual deployment of the lazy buddy and lazy iBuddy systems is the effect of the batch size. For this, we consider four cases, that is, the lazy buddy with batch size 1 (LBS-1) and 31 (LBS-31) and lazy iBuddy with batch size 1 (LiS-1) and 31 (LiS-31). Fig. 11 shows the average elapsed time (in cycles) consumed for allocation and deallocation for the six benchmarks under the four buddy systems as the number of threads is varied.

For the traditional lazy buddy system, LBS-1 outperforms LBS-31 in some cases and vice versa in other cases due to the trade-off between the lazy layer management overhead and lock contention, already discussed with Fig. 4. In other words, the performance of the traditional buddy system is sensitive to the batch size, and that the appropriate batch size depends on the number of concurrent threads employed. In contrast, for our proposed iBuddy, LiS-1 consistently outperforms LiS-31 regardless of the number of threads. This is because the multiple buddy

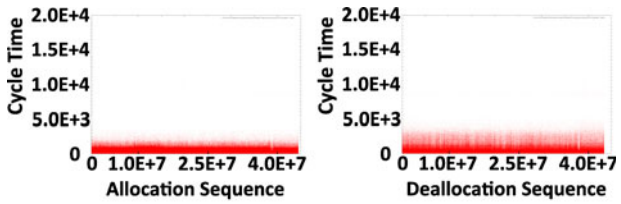


Fig. 12. (a) Allocation and (b) deallocation time (cycles) for requests in the lazy iBuddy system for the Kernel Compile benchmark for single thread execution.

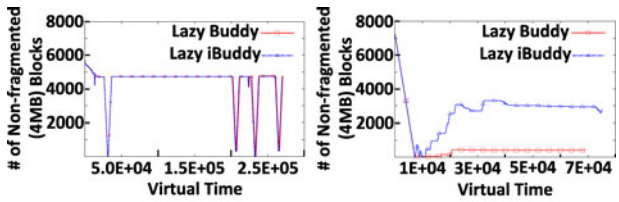


Fig. 13. Long term effect of iBuddy on fragmentation. The x-axis is virtual time, while the y-axis is the number of free non-fragmented 4 MB blocks.

space management scheme reduces lock contention while the lazy bypass technique lessens the lazy layer management overhead as well. The results show that one can expect the best performance for the lazy iBuddy system when the batch size is set to 1 independent of the number of concurrent threads that are executing.

6.3.4 Variation in Allocation/Deallocation Time

Fig. 12 shows the allocation and deallocation time for memory requests in the lazy iBuddy system for the Kernel Compile benchmark. Comparing these results with the results of the traditional lazy buddy system presented in Fig. 1, we find that the average elapsed time drops from 303 cycles for the lazy buddy system down to 194 cycles for the lazy iBuddy system. More importantly, the standard deviation of the service time drops from 1,427 to 241 cycles improving the predictability of the responses. The results presented here are only for measurements for single thread executions, but results for multiple thread executions as similar. These results are also consistent for all the workloads that we considered.

6.3.5 Overall Performance

The performance discussion so far have concentrated on the allocation and deallocation aspect of the iBuddy system. One has to wonder how much of an effect iBuddy has on the overall performance of executing an application. Unfortunately, as our study is an implementation study it is difficult to separate out the effects of the individual components of executing applications in a system. Hence, we are not able to separately quantify the effect of the iBuddy algorithm itself. However, in a separate study that incorporates the iBuddy algorithm, we find that the performance of applications used in this study improves by an average of roughly 20 percent, with a maximum improvement of roughly 85 percent [15]. We find that the reason behind such improvements is not simply due to replacing the buddy algorithm with the iBuddy algorithm, but also due to randomization that actually helps reduce conflicts in the row-buffer. We refer readers interested in the overall effects of iBuddy to the paper by Park et al. [15].

6.4 Degree of Fragmentation

Based on our description of the lazy iBuddy system, a natural question that arises is its long term effect on memory fragmentation. As the iBuddy algorithm tries to allocate page frames from a larger buddy group first, it may be possible that fragmentation will increase resulting in a lack of successive free page frames. For a quantitative evaluation to this question, we count the number of 4 MB blocks that are free, that is, where all pages in this block are free. Note that 4 MB represents the largest consecutive number of pages that can be allocated at once ($2^{10} \times 4$KB) on our experiment platform.

To show long term effects we perform two sets of experiments: one where our benchmarks are executed in sequence, that is one after the other, and one where our benchmarks are executed concurrently. When the benchmarks are executed in sequence, we execute the benchmarks with the maximum number of threads. When executing concurrently, six instances of each benchmark, running with 1, 2, 4, 8, 16, and 32 threads, are all executed at once. Measurements are made at particular intervals (2^{12} memory requests when executing in sequence and 2^{14} memory requests when executing concurrently) or when the number of free 4 MB blocks change.

Fig. 13 shows the number of free, non-fragmented 4 MB blocks (y-axis) as virtual time progresses (x-axis). The results show that when executed in sequence the number of free 4 MB blocks is almost indistinguishable between the traditional lazy buddy system and the lazy iBuddy system (Fig. 13a. When the benchmarks are executed concurrently, the results show that the number of non-fragmented 4 MB blocks actually increase substantially for iBuddy as shown in Fig. 13b.

This is evidence that buddy space management is playing a major role in curtailing fragmentation. Recall that through buddy space management, 4 MB blocks are allocated to each core, and threads in each core will be allocated memory from that particular 4 MB block until all pages run out. Hence, buddy space management confines memory allocation and deallocation to within 4 MB boundaries as best it can. This results in many 4 MB blocks being left untouched in contrast to the traditional lazy buddy system.

The results here show that memory fragmentation is not a serious issue for the lazy iBuddy system. Though single page management of the iBuddy algorithm may seem to encourage fragmentation, the multiple buddy space management component of the system acts as an effective countermeasure, even reducing fragmentation.

7 Related Work

There are two research domains relevant to this paper. One is the buddy system management domain and the other is

the scalable lock management domain, both in terms of main memory allocation and deallocation. [3], [4], [5], [6], [7], [16], [17], [18], [19], [20], [21], [22]. Both these research domains have not been very active in recent years. We revisit this old problem in view of the new world of large memory, multi-core systems.

The buddy system with lazy layer manipulation is at the core of the dynamic memory manager employed by various UNIX-like systems. The goals of the buddy system are supporting efficient allocation/deallocation and offering guaranteed availability by satisfying memory requests of various sizes. Barkley and Lee suggested a lazy buddy system where coalescing delays are bounded to two per class by utilizing a DELAY-2 coalescing policy [3]. Brodal et al. proposed a superblock based algorithm for allocation and deallocation [4]. The superblock that contains allocatable blocks of various sizes provides lazy splitting merits. Page and Hagins designed a dual buddy system that can reduce fragmentation [7]. It supports a large number of buddy sizes without requiring a large tree height. Cranston and Thomas proposed the Fibonacci buddy system that can enhance the availability of memory management by providing a diverse range of block sizes [5]. Yadav and Sharma designed a new buddy system that allows different block sizes [21]. Herter et al. proposed a dynamic memory allocation algorithm for a hard real-time setting, where worst case execution time analysis is paramount [20].

The second research domain related to our work is scalable lock management. On multiprocessor environments, a memory allocator suffers not only from poor performance but also from reduced scalability due to lock contention. Berger et al. introduced Hoard, a fast, highly scalable allocator that effectively avoids false sharing [16]. Hoard combines one global heap and per-processor heaps with a discipline that provably bounds memory consumption and has low synchronization costs. Michael proposed a lock-free memory allocator using hardware atomic instructions [19]. Also, a variety of techniques such as a per-CPU data structure to reduce lock contention [6], lock-free malloc [17], [22], and a parallel buddy manager [18] have been proposed to enhance the performance of memory management for concurrent accesses.

Our approach differs from all previous research in that we propose a novel approach that exploits individual page frame management for allocation and deallocation. This management scheme allows us to further propose the lazy bypass technique to improve performance and the multiple buddy space management scheme to improve scalability.

8 Conclusion

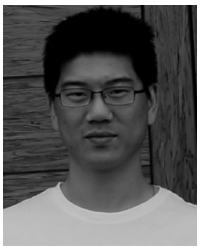
In this paper, we proposed a new dynamic memory manager called the lazy iBuddy system. This system was developed with three goals in mind, that is, 1) avoid splitting and coalescing as much as possible, 2) perform efficiently for the common case, that is, single page allocation requests, and 3) alleviate lock contention as best it can with lightweight overhead. The first two goals are achieved using the individual page frame management scheme and the last goal is accomplished using the multiple buddy space management scheme with the lazy bypass technique.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MEST) (No. 2012R1A2A2A01014233) and (No. 2012R1A2A2A01045733).

References

[1] D.E. Knuth, *The Art of Computer Programming*, vol. 1, third ed., Addison-Wesley, 1997.
[2] J.L. Peterson and T.A. Norman, "Buddy Systems," *Comm. ACM*, vol. 20, pp. 421-431, June 1977.
[3] R. Barkley and T. Lee, "A Lazy Buddy System Bounded by Two Coalescing Delays," *Proc. ACM Symp. Operating Systems Principles (SOSP '89)*, pp. 167-176, 1989.
[4] G.S. Brodal, E.D. Demaine, and J.I. Munro, "Fast Allocation and Deallocation with an Improved Buddy System," *Acta Informatica*, vol. 41, pp. 273-291, Mar. 2005.
[5] B. Cranston and R. Thomas, "A Simplified Recombination Scheme for the Fibonacci Buddy System," *Comm. ACM*, vol. 18, pp. 331-332, June 1975.
[6] P.E. McKenney, J. Slingwine, and P. Krueger, "Experience with an Efficient Parallel Kernel Memory Allocator," *Software—Practice and Experience*, vol. 31, pp. 235-257, Mar. 2001.
[7] I.P. Page and J. Hagins, "Improving the Performance of Buddy Systems," *IEEE Trans. Computers*, vol. C-35, no. 5, pp. 441-447, May 1986.
[8] W. Mauerer, *Professional Linux Kernel Architecture*. third ed., John Wiley & Sons, 2008.
[9] A.S. Tanenbaum, *Modern Operating Systems*. third ed., Prentice Hall, 2009.
[10] STREAM, "Stream Benchmark," http://www.cs.virginia.edu/stream, 2013.
[11] SysBench, "Sysbench: A System Performance Benchmark," http://sysbench.sourceforge.net/, 2014.
[12] Ramspeed, "Ramspeed Benchmark," http://alasir.com/software/ramspeed/, 2014.
[13] Benchmark, "Linux Benchmark Suite Home Page," http://lbs.sourceforge.net/, 2014.
[14] UnixBench, "Unixbench: A Fundamental High-Level Linux Benchmark Suite," http://www.tux.org/pub/tux/benchmarks/System/unixbench/, 2014.
[15] H. Park, S. Baek, J. Choi, D. Lee, and S.H. Noh, "Regularities Considered Harmful: Forcing Randomness to Memory Accesses to Reduce Row Buffer Conflicts for Multi-Core, Multi-Bank Systems," *Proc. 18th Int'l Conf. Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*, pp. 181-192, 2013.
[16] E.D. Berger, K.S. McKinley, R.D. Blumofe, and P.R. Wilson, "Hoard: A Scalable Memory Allocator for Multithreaded Applications," *ACM SIGPLAN Notices*, vol. 35, pp. 117-128, Nov. 2000.
[17] D. Dice and A. Garthwaite, "Mostly Lock-Free Malloc," *Proc. Third Int'l Symp. Memory Management (ISMM '02)*, pp. 163-174, 2002.
[18] T. Johnson and T. Davis, "Space Efficient Parallel Buddy Memory Management," *Proc. Fourth Int'l Conf. Computing and Information (ICCI '92)*, pp. 128-132, 1992.
[19] M.M. Michael, "Scalable Lock-Free Dynamic Memory Allocation," *ACM SIGPLAN Notices*, vol. 39, pp. 35-46, May 2004.
[20] J. Herter, P. Backes, F. Haupenthal, and J. Reineke, "Cama: A Predictable Cache-Aware Memory Allocator," *Proc. 23rd Euromicro Conf. Real-Time Systems (ECRTS)*, pp. 23-32, 2011.
[21] D. Yadav and A.K. Sharma, "Tertiary Buddy System for Efficient Dynamic Memory Allocation," *Proc. Ninth WSEAS Int'l Conf. Software Eng., Parallel and Distributed Systems (SEPADS '10)*, pp. 61-66, 2010.
[22] X. Huang, C. Rodrigues, S. Jones, I. Buck, and W.-M. Hwu, "Xmalloc: A Scalable Lock-Free Dynamic Memory Allocator for Many-Core Machines," *Proc. IEEE 10th Int'l Conf. Computer and Information Technology (CIT)*, pp. 1134-1139, 2010.



Heekwon Park received the BS, MS, and PhD degrees in computer engineering from Dankook University in 2005, 2007, and 2012, respectively. He has been a post-doctoral research associate at the University of Pittsburgh since 2012. His research interests include operating system, memory subsystem and management, DRAM architecture, file systems, flash memory, and embedded systems. He is a member of the IEEE.

Jongmoo Choi received the BS degree in oceanography from Seoul National University, Korea, in 1993, and the MS and PhD degrees in computer engineering from Seoul National University in 1995 and 2001, respectively. He is an associate professor in the Department of Software Science, Dankook University, Korea. Previously, he was a senior engineer at Ubiquix Company, Korea. He held a visiting faculty position at the University of California, Santa Cruz, from 2005 to 2006. His research interests include microkernels, file systems, flash memory, and embedded systems.

Donghee Lee received the MS and PhD degrees in computer engineering, both from Seoul National University, Seoul, Korea, in 1991 and 1998, respectively. He is currently a professor in the Department of Computer Science, University of Seoul, Korea. He was a senior engineer at Samsung Electronics Company, Korea, in 1998, and an assistant professor in the School of Telecommunication and Computer Engineering, Cheju National University, Korea, from 1999 to 2001. His research interests include operating systems, I/O systems, flash memory, and embedded real-time systems.

Sam H. Noh received the BS degree in computer engineering from Seoul National University, Korea, in 1986, and the PhD degree from the University of Maryland at College Park in 1993. He held a visiting faculty position at George Washington University from 1993 to 1994 before joining Hongik University in Seoul, Korea, where he is currently a professor in the School of Computer and Information Engineering. His current interests include operating system issues for parallel, distributed, and embedded systems with an emphasis on storage systems.