CBUFs: Efficient, System-Wide Memory Management and Sharing *

Yuxin Ren Gabriel Parmer Teo Georgiev
The George Washington University, USA
{ryx, gparmer, tgeorgiev}@gwu.edu

Gedare Bloom
Howard University, USA
gedare@scs.howard.edu

Abstract

Modern systems are composed of many different protection domains separating privilege levels, subsystems, users, clients, and software of differing levels of assurance. System-wide memory management must consider not only allocation to single processes, but also efficient sharing of data across protection domains, and the allocation of memory based on the performance of applications that span multiple protection domains.

This paper introduces the CBUF system for the global management of virtual and physical memory, including zero-copy sharing between protection domains. We present the design and implementation of both garbage collection techniques to enable efficient sharing, and policies that balance memory between protection domains specifically to satisfy system and application constraints such as quality of service. We show that a CBUF-enabled webserver achieves over a factor of 2.5 throughput speedup while using less processing time than Apache on Linux, and that the system can intentionally control system throughput through intelligent memory allocation.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Allocation/deallocation strategies

Keywords zero-copy, shared memory, garbage collection, memory management

1. Introduction

Systems are increasingly focused on ensuring isolation between different principals. From cloud systems to embedded systems, multi-tenancy and mixed-criticality [4] systems demand that isolation barriers are erected to provide system security and reliability constraints. However, with increased isolation, the intra-system data movement paths that transmit data across those protection boundaries have a significant impact on system structure and performance. In traditional kernels such as those that implement POSIX, these paths are mediated by APIs like `read`, `write`, and `mmap`. Such APIs often either 1. include data copying, which enables the developer to avoid reasoning about concurrency when managing their memory, or 2. rely on the developer to explicitly lay out and manage memory as with memory mapped files, and shared memory. At the lower-levels, page flipping in Xen [5], inter-VM shared memory [11], and [10] creates shared regions between user- and kernel-level for more efficient networking.

However, many of these approaches to intra-system data movement are narrowly applicable, or imply significant performance overheads, such as copying. With traditional APIs, shared memory does not always imply zero-copy. Because of the separation of memory management and sharing, data is often copied into the shared region. In addition, traditional approaches do not provide system-wide management of the sizing of different memory pools. As data movement increases in importance, the explicit, system-wide memory management of buffers used for data movement is essential to provide both simple APIs (*e.g.* similar to `malloc/free`), and high performance. To avoid narrow APIs that apply only to special-cases, system-wide garbage collection abstracts away concurrency and liveness issues inherent in moving data between asynchronously interacting parts of the system.

This paper introduces CBUFs that rethink system-wide memory management by coupling three traditionally disparate system functions: efficient page granularity allocation/deallocation, shared memory, and management of virtual and physical memory. CBUFs back all memory allocations in the system, including static memory such as `.text`, `.rodata`, and `.data`. In this way, CBUFs fill a role similar to `mmap` in UNIX systems. However, one of CBUF's main uses is as an efficient means for *sharing and moving data* between separate protection domains while still maintaining isolation guarantees. CBUFs pair shared memory with efficient allocation to provide zero-copy inter-protection domain message passing.

Memory management facilities, such as `malloc` implementations or garbage collection run-times, manage a specific process' dynamically-allocated memory on the heap. Such facilities rely on the operating system's page-granularity memory management API (including `mmap`, `madvise`, and `munmap`) for backing memory to the generally-sized allocations they provide. CBUFs provide a similar API to request extents of pages, and to release them, but also focus on the *sharing* of those buffers across different protection domains, thus enabling data movement. As such, determining the *liveness* of a given buffer (i.e. if references within any protection domain exist to the buffer) is complex. Execution within a protection domain can reuse a buffer only when all other protection domains are no longer referencing it. For example, domains (such as the kernel) that must maintain references to the buffer until an I/O device transmits the data at an uncertain point in the future. The system-wide garbage collection of CBUFs requires a global view of how and when each protection domain is accessing the buffer.

* This material is based upon work supported by the National Science Foundation under Grant No. CNS 1149675 and ONR Award No. N00014-14-1-0386. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation or ONR.

Conceptually, as buffers of data move throughout the system, references to them are tracked, and global policies for garbage collection reuse them, and rebalance them to where they are needed.

Another aspect of system memory management that CBUFs assume is policies for memory management to different protection domains. System-wide memory management algorithms such as buddy allocators for page-granularity allocations are often paired with quotas to allocate memory both for the kernel, and to satisfy application requests made through system calls such as `mmap` or `sbrk`. CBUFs must comparably optimize not only for efficient allocation, garbage collection, and sharing, but also for balancing memory allocations throughout the system to maintain higher-level goals such as throughput optimization, and Quality of Service (QoS). In this way, such goals satisfy the functionality of being the physical memory allocation infrastructure for the system.

Contributions and organization. CBUFs make the following contributions:

- *Efficient, simple message passing.* Data movement throughout the system must avoid expensive copying by using zero-copy communications, while providing familiar APIs for memory management. Section 2 discusses this design.
- *Generic data movement design.* The separation of data and control transfer paired with full-system garbage collection enable data sharing across asynchronous flows of control. Section 3 discusses the CBUF implementation.
- *System-wide memory management.* The amount of memory allocated to each protection domain is determined by a global CBUF policy that can optimize for multiple metrics such as end-to-end QoS of data transmission. This is discussed in Section 4.
- *System evaluation.* We provide a thorough evaluation of CBUFs in a number of contexts in Section 5.

Sections 6 and 7 present the related work and conclusions, respectively.

2. CBuf Design

2.1 CBuf Goals and Terminology

CBUFs unify three disparate functions in traditional systems: shared memory for zero-copy message passing, efficient memory allocation, and system-wide physical memory allocation. CBUF's management of *all* of these concerns enables efficient data-movement, and application performance-aware allocation of physical memory, but leads to a number of design challenges. The goals of the CBUF design follow.

G1 *Efficient memory allocation, deallocation, and sharing.* Unlike `mmap`, CBUFs are allocated with a high frequency, for example, as packets pass through the system. Thus, CBUFs must provide overheads similar to `malloc`, but for memory that can span multiple components.

G2 *Simple API for zero-copy buffer sharing and reclamation.* The burden of the manual management of shared memory is assumed by CBUFs, and garbage collection is used to reuse shared buffers that are not accessed.

G3 *Controlled trade-off between memory allocation, and latency via physical memory scheduling.* Large pools of memory lower thread block-time spent waiting for memory. CBUFs control this trade-off at run-time based on how given allocations impact application execution, and how much physical memory is available.

G4 *Isolation.* Though CBUFs focus on enabling efficient shared memory, different system components must maintain isolation to prevent fault propagation.

Terminology. CBUFs focus on system-wide memory allocation across a set of *components*. We use this term to generalize protection domains that have access to different, possibly disjoint, sets of memory. Processes in POSIX systems are components. Systems such as COMPOSITE are designed to focus on dependability, and isolate system-level services (*e.g.* scheduling, file-systems, device drivers) as separate components. Inter-component communication is frequent in such systems, thus emphasizing the need for efficient data-movement.

2.2 Shared Buffer Model

A single CBUF (*i.e.* a single buffer) is a contiguous region of memory, sized to be a multiple of a page that is associated with a set of attributes. Each CBUF is named by a CBUF identifier, an opaque, unique, global identifier. The CBUF id is integral to sharing the buffer between components as it is used to specify which buffer is being passed. The CBUF subsystem manages the physical frames on the system, thus controls (though the kernel system-call API) their mapping into component's page-tables. This motivates the CBUF page-granularity requirement. The Speck kernel underlying COMPOSITE enables the CBUF manager to *safely* control the page-tables of each component through a capability-based system [22]. When a CBUF is shared between components, it is mapped into both of their page-tables. This (and especially unmapping them due to TLB shootdown) is an expensive operation, and CBUFs attempt to minimize it by *caching* CBUFs in components. This caching of CBUFs in a component's virtual address space is essential to maintain efficiency and satisfy **G1**.

There are four types of CBUFs: 1. garbage collected buffers that are shared between components (for **G2**); 2. *aggregate* CBUFs which are discussed below; 3. buffers whose lifetime is controlled by the allocating component (*e.g.* memory to back `malloc` implementations), thus don't require full-system garbage collection (similar to previous work on TMEM [20]); and 4. statically allocated memory that isn't deallocated until a component is destroyed to back the likes of `.text` and `.data` sections. The first two types are the focus of this paper.

Isolation and sharing are often opposed, but CBUFs wish to provide a level of isolation such that *any* component cannot modify shared data while other components might be parsing and accessing it. To satisfy **G4**, CBUFs are *immutable* outside of the component that originally populated them. This immutability can be inconvenient in components such as the networking stack that need to prepend headers onto packets, or to split up a buffer into multiple packets. *Aggregate* CBUFs are an array of (CBUF id $\times$ offset $\times$ length) tuples that are, themselves, stored in a CBUF. Thus, aggregate CBUFs are shared consistently with normal CBUFs. Copy-on-write can be used instead, where a new CBUF is created from the old, with the required additions or modification. This is appropriate when the modifications are significant, but aggregate CBUFs enable zero-copy for common operations like append.

2.3 Client Programming Interface

Client API	Description
`void *cbuf_alloc(size_t sz, cbuf_t *cb)`	Allocate cbuf
`void cbuf_free(cbuf_t cb)`	Deallocate the cbuf
`void cbuf_send(cbuf_t cb)`	Send out the cbuf
`void *cbuf2buf(cbuf_t cb, size_t sz)`	Retrieve cbuf

Table 1. Summary of main CBUF API

Table 1 lists the main operations provided by the client library that are used directly by component code. Allocation returns both a pointer to the buffer, and the `cbuf_t`, the CBUF id. In addition to allocation and deallocation, the API includes a message-like API for sending and receiving a CBUF. `cbuf_send` aids in tracking liveness and is called directly before a CBUF is sent to another component. `cbuf2buf` is used by a component receiving a CBUF

and it translates from the global CBUF id into a local buffer that the component can access. These latter two operations are often hidden from components, and instead performed by stub code that interposes on the communication mechanism. This code is often generated by an Interface Definition Language (IDL) Compiler. The implementation details of this API will be discussed in section 3.

2.4 System architecture

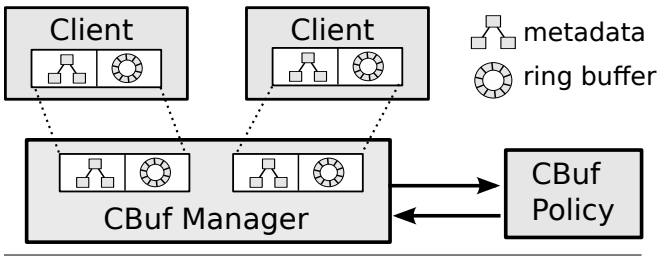

Figure 1. CBUF system architecture

The CBUF system involves three parts: a centralized CBUF manager component, a library that is used in *client* components, and a CBUF scheduling policy component that is responsible for deciding how much memory should be allocated to each component. Figure 1 illustrates the system's architecture. The CBUF manager and policy components instance are responsible for managing the CBUFs for a set of client components. For each client, the manager maintains a *pool* of CBUFs and manages its size. The total amount of memory requested by a client cannot exceed its pool size, thus the manager controls system-wide memory allocations. If the client makes a request that requires a larger pool, the client will be blocked to either wait for a previous CBUF can be reused, or for the policy to expand its pool. CBUF policy interacts with the manager through a well-defined interface that is used to (1) collect information about clients' memory usage and how long they have blocked waiting for memory and, (2) to adjust their pool size.

The pool of memory available for allocation is mapped into the client, thus directly accessible even after it is freed (much like `malloc` implementations delay returning memory to the OS). One of the main difficulties in the design of CBUFs involves the tension between (1) the efficient allocation and deallocation (**G1**), and (2) the manager's management of pool sizes and garbage collection (**G2** and **G3**). For efficient allocation, IPC with the CBUF manager must be avoided. Even if the manager were implemented in the kernel in a monolithic system, system call overheads would dominate allocation costs. The fast-path of memory allocation for a client component from its pool is to use size-partitioned, lock-free freelists. Most importantly, the high costs of mapping and unmapping memory [6] must be avoided. At the same time, the manager must be able to ascertain when each component is accessing each CBUF, and must coordinate with each client to add and remove those CBUFs from the client's pool.

CBUF client and manager coordination. Figure 1 shows two structures that are shared between each client and the CBUF manager. First, a structure maintains the metadata about each of the CBUFs in that client's pool. This metadata is indexed by CBUF id and includes the CBUF's virtual address, size, and reference count information. This structure is shared between client and manager and enables the manager to synchronize the map and unmap of CBUFs, with client access and reference to CBUFs. Importantly, it enables the client to implement the API in Section 2.3 entirely in the library with the exception of handling out of memory cases. Second, a ring buffer is mapped between client and manager. When the client finds that it has no free CBUFs of the requested size, it asks the manager to perform garbage collection of CBUFs it has previously sent to other components. The set of CBUFs that are collected and can be reused are passed to the client in the ring buffer, thus amortizing the cost of communication across many garbage collected buffers.

CBUF garbage collection. Liveness of a CBUF is tracked based on the API functions `cbuf_alloc` and `cbuf2buf` which signify a reference in the component that creates the CBUF, and those that have that CBUF passed to them, respectively. `cbuf_free` designates dropping a reference. As even low-level systems code (such as networking stacks) uses CBUFs, we rely at the API-level on explicit memory management. The CBUF client library tracks all CBUFs that we know have no references to them, and uses them to provide efficient allocations (using per-size freelists). However, when this supply of CBUFs diminishes, the manager is asked to perform a collection. This action has a significantly higher overhead than the fast-path for allocation, but its cost is amortized by the client's CBUF pool size – the larger it is, the larger the number of un-referenced CBUFs *cached* in the client. Garbage collection requires the manager to walk through the metadata for each of the CBUFs that belong to the client requesting collection, and determine liveness of each. Those that are not live (*i.e.* referenced), are passed for reuse to the client.

Altering client pool sizes. The policy component asks the CBUF manager to change the size of the pool for each component. The manager can easily *rebalance* CBUFs from a component to another during garbage collection. CBUFs that are found to not be live, can be unmapped from their current components, and mapped into the destination. However, rebalancing must also work when clients are inactive, and don't trigger a collection, or when all of a component's CBUFs are currently in use.

FBufs design comparison. FBufs [6] also provide zero-copy data movement, but use a very different design than CBUFs. A fixed portion of each component's virtual address space is used to *globally* address shared buffers. The references for these buffers are tracked explicitly [6, 16], thus avoiding the needs for intelligent sharing and collaboration between manager and clients, and swapping is used to rebalance memory away from a client. Modern systems often disable swapping (*i.e.* swapping to and from disk to virtually increase memory size) due to the erratic behavior it can cause. Embedded systems cannot use swapping due to their need for predictability. In some sense, CBUFs are an update to FBufs given the needs of modern systems that enable (1) a client fast-path that completely avoids manager invocations, and (2) system-wide garbage collection of CBUFs, and policies around how to rebalance memory to compensate for time spent blocked waiting for a CBUF allocation.

3. CBuf Implementation

We implement CBUFs in the COMPOSITE [17] component-based OS. In this system, all system components including the lowest-level policies such as scheduling, memory management (in the CBUF manager), networking, and filesystems are implemented as user-level components, each isolated in separate protection domains. Thus, the CBUF manager and policy are implemented as hardware-isolated, user-level components. Our prototype currently executes on 32-bit x86. The evaluations in this paper are for a single core. A multi-core implementation simply includes a pool per-component, per-core. A different implementation could implement the CBUF manager and policy in the kernel of a different OS such as Linux. In those systems, it is straightforward to implement the CBUF manager's functionality in the kernel since the kernel has full control of system resources. User-level CBUF client and policy communicate through system calls. However, this requires either

adding new system calls or breaking existing interfaces, such as `mmap` and `munmap`. This is an area of future work.

3.1 CBUF Client Implementation

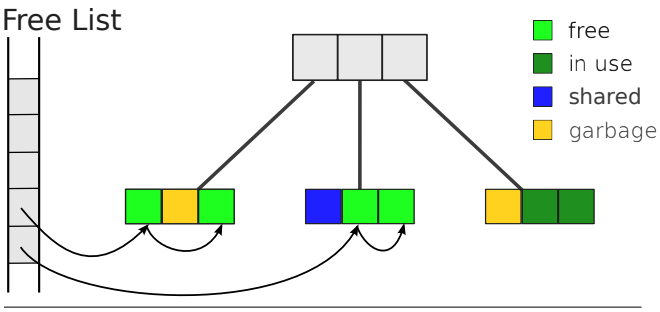

Figure 2. shared metadata radix tree

The shared metadata structure depicted in Figure 1 is implemented as a radix trie for the efficient lookup of a CBUF's metadata by its identifier. This radix trie is depicted in Figure 2. Each metadata item for each CBUF is in a single state: `free` in which case it is in a freelist for CBUFs of a specific size; `in use` if the CBUF is currently referenced within the component; `garbage` if the CBUF is not live within `this` component, though it might be referenced from another; and `shared` in which case it has been `cbuf_send` to another component (see next section).

When the CBUF manager collects all non-live CBUFs for a component, they are transmitted back to the client, and the client places the CBUFs back on the appropriate freelists. Our current prototype uses power-of-two sized freelists.

3.2 CBUF Liveness

Central to the CBUF design is the capability to determine a CBUF's liveness across multiple client components that are sharing the buffer. Essentially, the CBUF system must answer the key question: "when can a CBUF memory be reused?". CBUFs accomplish this with a combination of three, per-CBUF, per-component counters.

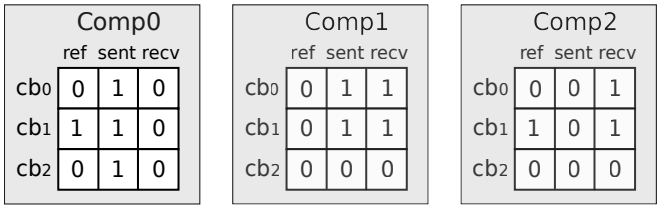

Figure 3. Examples showing a referenced, sent and receive counters for three different CBUFs. cb_0 can be reclaimed, as no components have reference to it and sent counter's sum and receive counter's sum are equal. cb_1 cannot be reused because of the non-zero reference counter in component 0 and 1. cb_2 cannot be reclaimed since it has a pending send.

The CBUF system uses two mechanisms to track a CBUF's liveness. Firstly, for each CBUF, each client component has its local reference counter to indicate how many threads are currently using this CBUF with a component. Whenever there are non-zero reference counters, the CBUF cannot be reclaimed. This reference count is incremented on `cbuf_alloc` and `cbuf2buf`, and decremented on `cbuf_free`.

Second, we note that due to the possibly asynchronous nature of execution in different components, the distributed reference counters are not sufficient to track liveness. The race condition happens when a CBUF is freed by its sender before receiver receives it. In such a case, the reference counter of sender and receiver are each zero, so a collection at this point would inappropriately treat the CBUF as garbage. But the CBUF should *not* be collected as receiver will immediately attempt to `cbuf2buf` it. This race is possible because of the *distributed* nature of how CBUFs are tracked. The metadata for the CBUF exists per-component so that different clients are isolated, and so that the CBUF can be at different virtual addresses in each. To avoid the race, two additional per-client, per-CBUF counters are used: the *sent* and *received* counters. Sent counter records how many times a CBUF is sent by this client (via `cbuf_send`), and receive counter tracks the number of times the CBUF has been received (via `cbuf2buf`). For a specific CBUF, when the sum of all clients' sent counter is equal to the sum of all clients' receive counter, then all sends to other components have been received. Finally, only when a CBUF has a reference counter set to zero all in clients, and across all clients the sum of the sent and receive counters are equal, it can be collected and reused. Figure 3 gives examples of different combinations of reference counters, sent counter and receive counter for three different CBUFs in three different components.

3.3 Metadata structure

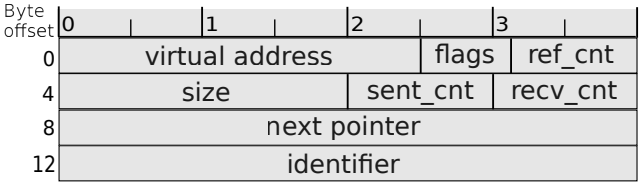

Figure 4. CBUF metadata layout. The *virtual address* is 20 bits, as CBUFs are always page-size aligned. Control flags include the *inconsistent* and *relinquish* bits. They occupy 5 bits, and the *reference count* is 7 bits. The *size*, in number of pages, is 16 bits. Both *sent* and *receive* counters are each 8 bits.

The metadata structure (in the last level of the radix trie) is detailed in Figure 4. Both the client and the manager synchronize modifications and access to this structure using atomic instructions as using locks would compromise their fault resilience. Thus, this layout is chosen mainly to enable synchronization around necessary fields. Related fields that must be modified atomically are packed in a single word so that atomic instructions can be leveraged. The first word includes the virtual address of the first page of the CBUF, enabling `cbuf2buf` to properly translate between id and address. This virtual address is set to `NULL` if the CBUF is not present in the component. That word also includes flags that encode a state-machine for the CBUF, and a reference count. The second word holds the CBUF size (in terms of number of pages) and send and receive counters. The last two words include a next pointer for the freelist and the CBUF identifier (used to return the CBUF id returned from `cbuf_alloc`).

3.4 Synchronization and Consistency

CBUFs rely on shared metadata structures between clients and the CBUF manager to enable client fast-paths that avoid calls (system calls, component invocations) to the manager. However, in sharing the metadata, synchronization is required. There are three different classes of interactions that require synchronization. First, the manager is invoked by many asynchronous threads, thus its access to its own data-structures must be synchronized. Second, multiple threads within a client component must synchronize access to the freelists, and CBUF meta structures. Third, modifications to the CBUF metadata must be synchronized between client and manager. We discuss these in turn.

CBUF manager synchronization. Data-structure access within the CBUF manager is protected by a lock. This design emphasizes

simplicity, and predictability, but would not scale on a parallel system. See Section 3.8 for a discussion of CBUF scalability.

Client synchronization. In a component, both CBUF metadata and the ring buffer must be synchronized. The ring buffer is a typical wait free design, but requires single producer (manager), and single consumer (client). To maintain the latter, a lock is used to serialize client thread's access to the ring buffer. We use the lock-free, Treiber stacks [19] to implement free lists which uses the compare and swap (CAS) atomic instruction to maintain stacks of CBUFs. To deal with "ABA problem" [23], the head of each freelist is used as an allocation/deallocation counter that is atomically incremented along with the head of the freelist on each allocation/deallocation. The current implementation uses double-word CAS that is available on commodity x86 processors. Should CBUFs be used on an architecture without such support, the counter could be inlined with the head pointer, or deferred reclamation could be used [23]. The reference count, and other fields in each CBUF's metadata are also updated atomically.

Note that unlike many `malloc` implementations that must manage the transition of memory between different freelists, and to and from a global pool, the CBUF design maintains a simple client API, and places the intelligence of memory movement not just between different freelists, but also between different client components into the manager which has a global perspective on client performance.

Client/Manager synchronization. It is essential that the CBUF manager and each of the clients it interacts with synchronize without using locks. Locks would significantly impair the isolation of the manager from client misbehavior. A client that never releases a lock hinders progress in the manager. Thus, the manager also uses only atomic instructions to modify the shared metadata. When adding a CBUF into a client component, it initializes the metadata structure (for example, adding the virtual address, CBUF identifier).

When the manager wishes to shrink the pool of a client component, it must remove one or more CBUFs from the component. There are three cases that must be considered here:

- There are CBUFs that are both not live (as defined in Section 3.2), and have not been collected and passed to the client. They are garbage pending collection. This is the simplest case, as they are not currently in use in any client. They can be unmapped, and their metadata updated (in each component) without further synchronization.
- A CBUF is not live, but it *is* present in the freelist of a client component. To solve this, the manager sets an *inconsistent* flag along with setting the CBUF's address to `NULL`. This flag is monitored during a subsequent client allocation. If it is set, the allocation immediately unsets the flag, ignores that allocation, and immediately allocates another CBUF. To discriminate between this, and the previous case, the manager checks if the metadata's freelist next pointer is `NULL` or not (the last item on the freelist has next set to 1).
- When the manager wants to shrink a client's pool of CBUFs more than can be accomplished with the previous two techniques, it means that there *live* CBUFs that we want to re-balance elsewhere in the system. In this case, removing the CBUF would likely cause faults in clients, so instead the manager wishes to receive a notification immediately when a CBUF is no longer live. A *relinquish* flag is set in all of a client's CBUFs with non-zero reference counts in each component. When a client calls `cbuf_free`, it checks for the relinquish flag if its reference count is now zero. In such a case it invokes the manager enabling it to eagerly re-balance the CBUF.

Synchronization between clients. We want to emphasize that clients do not interact except to send CBUF ids between each other, and via the send and received counts. Thus, they remain isolated except via the shared memory of the CBUFs themselves.

3.5 CBUF Manager

In addition to the techniques for synchronization and liveness previously discussed, the manager maintains a number of CBUF-related data-structures. The manager maintains two index tables, one is indexed by component id and the other by CBUF id. For each CBUF, a list of all components it is mapped into with references to that component's CBUF metadata. For each component, the manager maintains a list of all CBUFs in the pool for this component.

When a client component attempts to allocate a CBUF, but its freelist is empty, the client calls the manager. The manager attempts to immediately collect memory (this is how a collection is triggered). If no memory is available (not live), it blocks the thread after setting the *relinquish* flag as in Section 3.4 to get a notification when memory is available. When CBUFs are available for reuse, they are placed in the shared ring buffer with the client, and the thread is activated.

3.6 Client Library

The client library operations are intentionally designed to be simple, so that the fast-path is efficient. `cbuf_alloc` tries to fetch a CBUF from the freelist, while discarding inconsistent metadata. If the freelist is empty, the manager is invoked to perform garbage collection. Upon return, CBUFs are dequeued from the ring buffer, and added to their corresponding freelists.

In a component receiving a CBUF from another, `cbuf2buf` is called, and the CBUF id is indexed into the radix trie to get the CBUF's metadata. If the metadata does not exist, or the virtual address in it is `NULL`, the manager is invoked to map the CBUF and set up its metadata. Then both reference counter and receive counter are incremented, and the virtual address is returned.

`cbuf_send` only increments the send counter. `cbuf_free` decrements the reference count, and the checks the relinquish flag. If it is set, manager is called to appropriate allocate the CBUF.

Sub-page allocations. If allocations to be shared between components are frequently less than a page, a memory allocator can be layered on top of CBUFs. We do this, for example, by using a slab allocator [23] to allocate of packet headers for use with aggregate CBUFs in a networking stack (a modified lwIP [13]). Care must be taken in the allocator to properly reference count the CBUF (*i.e.* if *any* memory allocation is active in the page).

3.7 CBUF Security

Although immutable sharing provides some protection, the shared structures may expose the manager to attack via malicious modification of the metadata. As previously discussed, we avoid locks for synchronization between client and manager to prevent denial of service attacks. The manager has its own copy of each CBUF's information. Hence even if malicious clients modify important parts of the metadata such as virtual address or size, the manager can detect these modifications and act accordingly. These reactions are beyond the scope of this paper.

The client may fake the sent counter and receive counter to induce the manager to prematurely collect a CBUF. Clients will never crash due to the manager removing a CBUF from their address space as the CBUF is guaranteed to be accessible while the reference count is non-zero (reference count and CBUF are in the same word to enable consistency of these values). Alternatively, a client can maintain a reference to a CBUF or maintain high send counter to prevent collection of a CBUF. This is essentially an *accounting* problem, were the manager must simply account the memory for such CBUFs to that component after it sees it maintaining references for longer than would be appropriate for data streaming

through the system. Lastly, a client might pass the size of the buffer as larger than the backing CBUF supports. This is similar to a buffer overflow. `cbuf2buf` takes the supposed size as an argument, and validates that the CBUF's memory is correspondingly sized.

3.8 Multicore Concerns

Though our evaluation of the system is based on a single-core machine, the CBUF design accommodates multi-core systems. Each thread (or each core in COMPOSITE) has its own pool of memory that it allocates from, and the CBUF policy re-balances not just between components, but also between cores that execute code in the same component. The only portion of the system that would require a significant redesign to scale well on a multi-core system is the CBUF manager. The single lock that protects all CBUF manager data-structures would quickly prevent scaling. Future work includes implementing the manager's data-structures using wait-free algorithms and deferred memory reclamation [23], but for this work, we focus on system design for single-core systems.

4. CBuf Scheduling

The CBUF policy component is in charge of *scheduling* memory between domains. Over time, how large should the pool of memory be in each component? In order to flexibly support different CBUF scheduling policies, we separate the CBUF scheduling policy from the CBUF manager. The policy harnesses the manager's interface to harvest information about thread's and component's memory access patterns, and to tell the manager how component memory pools should be sized.

The policy separates predication from allocation decisions. The policy predicts the performance given component's CBUF pool size of that component's threads. It does this using metrics provided by the manager such as the amount of time threads spent blocked waiting for memory, and the frequency of garbage collection. System-wide constraints such as limits on available memory, and component priority or Quality of Service (QoS) are considered alongside the predictions of behavior when making allocations. Here we consider two policies that determine component CBUF pool sizes.

Minimize total blocking/GC time. This algorithm tries to reduce the system-wide amount of time spent on CBUF garbage collection and blocking as much as possible. In doing so, it attempts to maximize system throughput. The policy executes periodically, and records the memory usage in the last and current run, as well as blocking/GC time. It decides the quota of next run based on following cases. (1) blocking/GC time decreases because of more CBUF, the policy continues to increase the pool size by the same amount; (2) blocking/GC time decreases due to fewer CBUF requests (fewer allocations), the policy keeps the pool size constant; (3) blocking/GC time increases resulting from a pool size decrease, the policy resets the pool size to previous value; (4) blocking/GC time increases as a consequence of more frequent allocations, the policy increases pool size proportionate to the increase in memory requests.

QoS-aware. For each application, we define the QoS target as the percentage of its total blocking/GC time over its execution time. This is the decrease in application performance due to CBUFs. This algorithm tries to maintain that percentage around the given QoS level. This is done using a technique similar to binary search. The policy maintains three average CBUF pool sizes which correspond to larger, smaller, and equal to the desired QoS value. In the current period, if the blocking/GC time percentage is not close to the target value, the policy uses the average of current amount and the average value in the opposite side as the new quota.

Using the above two prediction algorithms, we implement and evaluate four policies under different system constraints. (1) `Maximize system-wide throughput, sufficient memory` – using the minimize total blocking/GC time algorithm, where all allocations can be satisfied as there are enough memory. (2) `Maximize system-wide throughput, limited memory` – Using minimize total blocking/GC time where the total prediction amount exceeds the memory limit, each component receives a decreased pool size proportional to their designed allocation. (3) `Priority based policy, limited memory` – Using the minimize total blocking/GC time policy, allocations are satisfied first for higher-priority applications, and, if memory remains, lower-priority applications then get their pool satisfied. (4) `QoS policy, limited memory` – it uses the QoS allocation algorithm and satisfies all applications with QoS guarantees first.

5. Experimental Evaluation

All experiments in this section are run on a 2.9 Ghz Intel Core i7-3520M processor with 1GB of memory. All Linux experiments use 32-bit Ubuntu 14.04 with a Linux kernel 3.19.0. We boot and execute COMPOSITE using the Hijack technique [21]. The goals of our evaluation include:

- Measure the efficiency of the CBUF implementation, comparing with existing techniques.
- Assess the cost of communication using CBUFs in both micro-benchmarks and real applications.
- Understand to what extent memory consumption impacts memory operation overheads.
- Illustrate the ability of CBUF to support flexible memory scheduling policies and evaluate those policies to see if they can achieve their specific trade-offs.

5.1 Micro benchmarks

Here we conduct a set of micro benchmarks to evaluate the overhead of the client API and compare it with similar techniques in Linux. All operations are executed 100,000 times, and we report the average results.

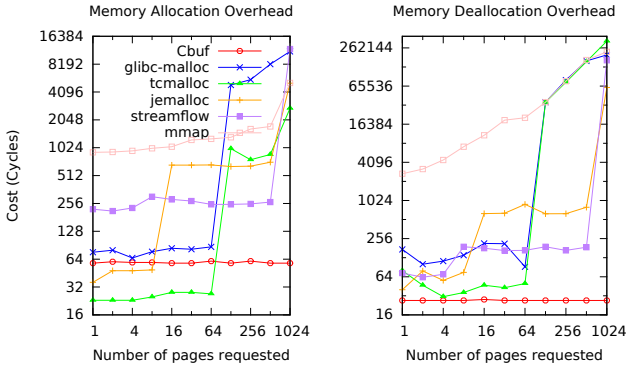

Figure 5. Memory allocation/deallocation cost.

Component-local CBUF operations. To apply CBUFs to real-world applications, we need to first understand the underlying costs of basic CBUF operations, especially the fast-path, client operations. We first compare CBUF memory allocation and deallocation performance with some modern allocators. We chose the malloc implementation from glibc-2.19, TCmalloc within gperftools-2.4, jemalloc-4.0.4, and streamflow (git-8ac345c). Though CBUFs have very different goals from those allocators, they all use common techniques to enable fast memory allocation/deallocation using thread- or core-local caches. We wish to compare the costs of the CBUF fast-path to assess whether or not the overheads could

Operation	Avg cost (cycles)
Cached `cbuf_alloc`	58
`cbuf_free`	27
`cbuf_send`	41
Cached `cbuf2buf`	68

Table 2. Basic CBUF Operations

inhibit CBUF use. We also include the cost of mmap and munmap as a comparison since CBUFs perform similar functions in our system. Figure 5 shows the comparison. For allocation of a small number of pages, excluding streamflow, all those allocators have very low and constant cost. Taking one page allocation for example, CBUF is 58 cycles, glibc is 76 cycles, tcmalloc is 23 cycles, jemalloc is 36 cycles, streamflow is 222 cycles and mmap is 912 cycles. As the number of pages allocated increases, CBUF overheads remain constant due to the uniform allocation methodology across sizes. By contract, general-purpose allocators allocate different size-classes using different technqiues (*e.g.* using best-fit allocators, or directly using `mmap`). The results show that the cached allocation costs of the CBUF operations should not inhibit the use of CBUFs. The impact of garbage collection on allocation will be discussed shortly.

Table 2 shows the overhead of all client-side CBUF operations. Memory size is set to one page for all operations, and we avoid garbage collection.

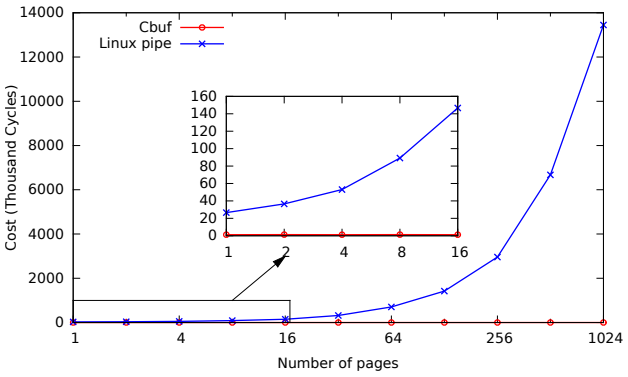

Figure 6. Round-trip message passing performance

Message passing performance. We simulate the pipe-based IPC test in the LMBench test suite using CBUFs in COMPOSITE and compare its performance with Linux pipes. In COMPOSITE, two components communicate with each other by both sending and receiving CBUFs. Similarly, in Linux pipes are used to send and receive message between two processes. Figure 6 reports the average round-trip cost with various message size. Communication for a single page is more efficient in COMPOSITE than in Linux (around 1278 vs. 26,554 cycles), so the base-line for each system is different. However, as the number of pages sent increases, CBUFs incur much less overhead than Linux pipes due to zero-copy data movement.

CBUF allocation cost vs. pool size. Figure 7 shows the impact on allocation cost of a component's pool size, and of the number of components that the CBUF is shared with. All allocations allocate one page of memory, and when there is no available memory in local cache, garbage collection is invoked. Thus the cost of allocation also includes garbage collection's overhead. The figure shows that the larger the pool size, the less frequently collections happen, and the more CBUFs are collected. This demonstrates the trade-off between processing time, and memory pool size. Allocation costs decrease to 253 cycles. This motivates high-level policies that can

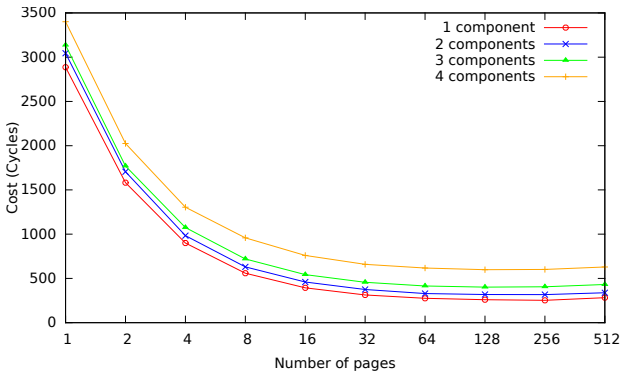

Figure 7. Amortized CBUF Allocation Cost

explicitly control this trade-off to meet performance goals, and best commit memory to component's pools.

5.2 Image stitching application

To evaluate the use of CBUF in existing applications to understand how we can increase isolation by using CBUFs along with separating the application into multiple components, we use image stitching. Image stitching is a process which combines multiple images with overlapping regions to generate a segmented panorama. Image stitching is widely used in camera or video applications, such as object insertion, panorama creation and video stabilization. Image stitching includes multiple stages that form a typical pipeline structure. Figure 8 illustrates the stitching pipeline.

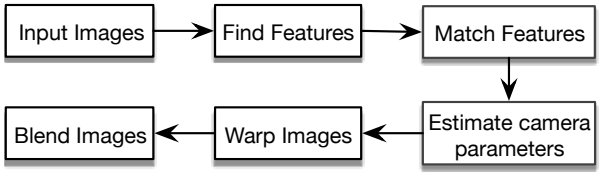

Figure 8. Image Stitching Pipeline

We use the sample image stitching application in the popular open source computer vision library – OpenCV [15]. Since OpenCV is a huge project consisting of many image processing modules, we only port those modules used by image stitching to COMPOSITE. We break the image stitching into multiple stages based on its pipeline showed in Figure 8, and implement each stage in a separate component. Most data passed between those stages are matrices. Thus we add a new matrix alloctor in OpenCV, which uses CBUFs to hold matrix data. When multiple images or matrices are transferred, they are assembled into a single aggregate. We compare the total completion time of image stitching in COMPOSITE with Linux. In Linux, it takes 385.52ms to complete, and in COMPOSITE it needs 389.63ms.

Discussion. Thanks to the simplicity of the CBUF interface, we found it is convenient and straightforward to apply CBUFs to this legacy application. During our porting process, no modifications are needed to the application's code, and we only add marshalling and unmarshalling functions for transferring objects between components. Because of zero-copy, CBUFs incur negligible overhead, and at the same time we gain significant increases in isolation.

5.3 Networking application

In this section, we investigate the benefit of CBUF's zero-copy data movement, and the effectiveness of various CBUF schedul-

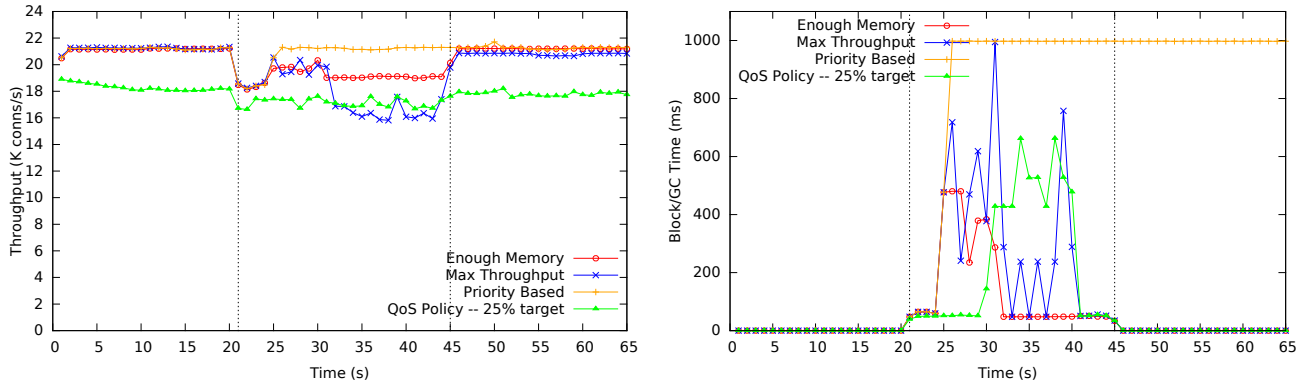

Figure 10. Web server throughput and interference thread blocking/GC time with different policies

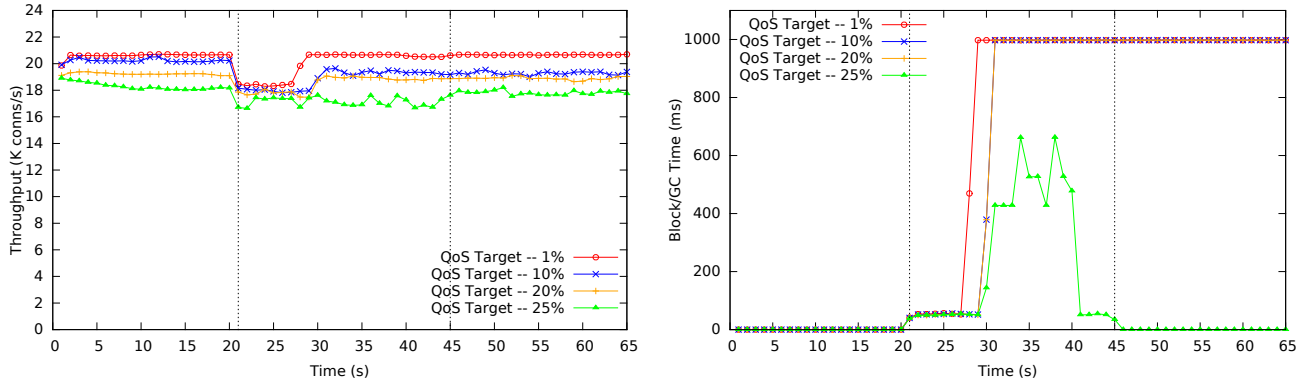

Figure 11. Web server throughput and interference thread blocking/GC time with different QoS target

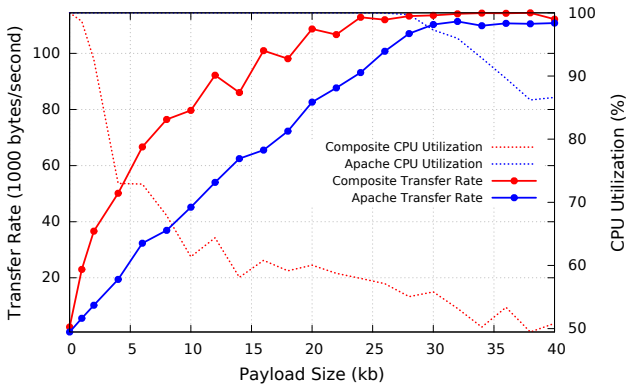

Figure 9. Web-server throughput comparison

ing policies using a custom HTTP server. We integrate CBUF to a web server which uses a separate FastCGI [8] component to handle HTTP requests. The webserver implemented in COMPOSITE is composed of more than 20 components, and each packet traverses through six components. CBUFs underlie all data movement through the system. We modified lwip-1.4.1 [13] to use CBUFs for both packet data and header. The header of each protocol layer and the HTTP payload are organized as an aggregate CBUF. The client and server machines are connected directly via ethernet cable.

Web server performance. We first compare our web server's throughput to an Apache web server with FastCGI, and vary HTTP payload size. We use an Apache 2.4.7 server with the FastCGI 2.4.7 module and the FastCGI developers toolkit 1.23. We disable logging, and ensure that superfluous modules aren't installed. The goal of these tests is to determine if CBUFs can provide practical throughput, and to validate that for increasing payload sizes, the system benefits from CBUF's zero-copy. We measure the performance of each system using the Apache benchmark program (`ab`) to send 100,000 HTTP requests for each payload size. Through trail-and-error, we found 22 concurrent connections maximizes throughput for both COMPOSITE and Apache.

Discussion. Figure 9 reports both transfer rate and cpu utilization. Transfer rate of COMPOSITE is consistently higher than Apache until the network is saturated. Even after the network is saturated, Apache still has higher cpu utilization than COMPOSITE implying that for a higher capacity network, COMPOSITE could continue to increase in throughput. However, we haven't identified why COMPOSITE fails to achieve 100% cpu utilization *before* network is saturated. This implies that the throughput should be higher for all but the lowest packet sizes. The most likely reason is that `lwIP`, which is not focused on high performance, does not aggressively enough saturate the network (*i.e.* the TCP window isn't aggressively enough increased).

CBUF scheduling in an open system. We evaluate all policies listed in section 4 with a web server, and an application that provides memory contention. The web server runs 65 seconds and replies to HTTP request with 1KB payloads, dynamically generated by a CGI component. At 21 seconds, the interference thread begins to execute periodically every 10ms, and at 45 seconds, it completes execution. In each period, the interference does 100,000

CBUF allocations and deallcations in total and thus it contends for CBUFs with the web server. On our machine, those operations take around 10% cpu time. The minimal memory requirement for those allocations varies over the time. It starts from 5 pages at 21 seconds, and increases to its peak value 64 pages at 31 seconds, then it keeps this value for 10 seconds, and after that it goes back to 10 pages. This gradually increases memory pressure, then decreases it. All policies are implemented as a periodic thread and run 4 times per second. They manage all web server component's memory plus the interference component thread.

Discussion. Figure 10 plots the web server's throughput and blocking/garbage collection time the interference spends on CBUF with different CBUF scheduling policies. In the first figure, we evaluate the policy that attempts to maximize system-wide throughput. The case where system has surplus memory serves as as a baseline. When the interference thread arrives, the web server's throughput drops about 10% due to the reduction of its cpu time. But in the first 4 seconds of interference's execution, the throughput decreases more than 10%. This is because the interference thread needs to create lots of new CBUF which takes more cpu resource. From 25 seconds to 30 seconds, the interference's memory requirement exceeds its CBUF pool size and it has to wait for the policy thread to expand its CBUF pool, therefore the web server gets more time to run and we get higher throughput. After 30 seconds, both web server and interference thread have sufficient memory and they all execute with optimal performance. And the total CBUF usage is 340 pages in this case.

However, for other policies the total CBUF usage is limited to 300 pages at most, and this amount cannot support both a web server and an interference thread to run at their highest performance. Hence this limit causes memory contention and complicates the memory scheduling. From figure 10, we can see under the memory constraint, the maximized system-wide throughput policy achieves similar system performance with the enough-memory case before 30 seconds. But after that, the web server's throughput has an additional 15% decrease, and the blocking/GC time of interference thread increases a bit. What is worse, there are some performance jitters in both web server and interference thread. This is because, after 30 seconds, the interference thread's memory requirement reaches its maximal value, and expanding its CBUF pool can significantly reduce system's total blocking time and improve whole system's performance as a consequence. So the policy moves much memory from web serve to the interference thread and degrades web server's throughput. However, if the web server's throughput degradation is too large, giving memory to web server has more benefit and some CBUFs are returned to the web server. This memory movement between web server and interference thread is the source of the inconsistent performance.

The results for the priority-based policy is relatively straightforward. We set all components belonging to the web server to a higher priority than the interference thread. So the web server's memory demand is satisfied first and the rest of available memory is left to the interference thread. After 25 seconds, the interference thread cannot get enough memory and is blocked waiting for available CBUF. The last one is QoS policy for web server whose QoS target is 25%. With such a high blocking/GC time, the web server's throughput indeed decrease significantly, but not down to the worst case throughput for the max throughput policy. Additionally, its performance is more consistent, while the interference thread also gets adequate memory to finish.

To more closely investigate the QoS policy, we run it with different web server QoS targets. This result is shown in Figure 11. We can see our QoS policy succeeds in maintaining predictable performance for all QoS targets. This demonstrates the web server's end-to-end latency can be precisely managed by an external CBUF scheduling policy. There are some other interesting observations. For the 10% QoS target, the throughput after 29 seconds is worse than before interference arrives. The reason is that some memory is allocated to the interference thread, and the policy doesn't rebalance them back to the web server when the interference thread is blocked. Comparing the 1% QoS target with the priority based policy, we found the interference thread can run longer. This illustrates the maximum throughput policy is more aggressive and it offers more CBUF memory to the web server than the QoS policy. In our experience, appropriate QoS targets can be derived from empirical investigation.

6. Related Work

Efficient memory allocators. Some similar techniques are used by both CBUFs and modern scalable memory allocators [7, 9, 18]. For instance, all of these employ hierarchical allocation with thread-local memory caches, and then from within a global allocation pool (within the process). Similar to CBUFs, local free lists are segregated by different size classes enabling fast allocation. However there is fundamental difference between CBUFs and those works.

- While those allocators aim for general purpose memory allocation, CBUFs are designed for efficient shared memory management among different protection domains, and are used as a backend for those allocators.
- CBUF liveness tracking is more complicated, as even after a cbuf is deallocated by one component, it can still be accessed by other components. Traditional allocators do not need to deal with such a problem as after a thread frees an object, other threads are not expected to use it any more.
- Most of the design of CBUFs is motivated by the need for a separation between the CBUF manager that makes memory allocations into component pools. Allocation from the pools within a client component must be efficient, yet the manager must be able to asynchronously re-balance memory between pools in response to measured application efficiency.

System shared memory. In Unix-like systems, shared memory can be created by using `shm_open` or the `mmap` API. These shared regions require that either memory be allocated inline in the shared memory region, and that it be copied into it. Shared memory does not provide a message passing API, so the programmer (or a library) must build one up. Zero-copy sharing along a pipeline (like the image processing application in Section 5) means comparable coordination among many processes. Doing so often makes different processes co-dependent as they rely on the proper management of the shared region. CBUFs perform this shared region management, and enable a message passing API, with central management of memory for the entire system.

In capability-based microkernel systems [12, 14], shared memory is built via capability transfer. This is the same lowest level mechanism used in COMPOSITE to create shared memory. However, in those systems, a fixed amount of data can also be copied during IPC. COMPOSITE takes a different approach. It separates data transfer from control transfer. All IPC parameters are passed in hardware registers and additional data is transferred via CBUF. This separation simplifies the design of the kernel and reduces the cost of IPC [22].

High-performance I/O. The design of CBUFs has been inspired by previous research [3, 6, 10, 16] that focuses on optimizing I/O performance. For example, Fbufs inspired the zero-copy data movement and aggregate buffers with immutable contents. But most of them deal with communication between kernel and user space, CBUF is more general as it can be used between any user level components. Furthermore, none of them have the ability to control trade-off between latency and memory consumption.

Cosh [2] provides inter-process sharing on heterogeneous cores. This can also be supported by CBUF, because the manager can decide to share the memory or copy the data according to the underlying hardware architecture.

Resource containers. Linux `cgroups` are an implementation of resource containers [1] and are aimed at limiting, accounting for, and isolating resource usage for a set of processes. `cgroups` use quotas to provide memory isolation between different groups. Though focused on memory, the CBUF management policy goes beyond quotas, and provides allocations that change over time in reaction to measured overheads, and system-wide goals (*i.e.* co-management of CPU and memory).

7. Conclusions

This paper presents CBUFs that rethink system-wide memory management by coupling three traditionally disparate system functions: efficient allocation/deallocation, shared memory, and management of virtual and physical memory. CBUFs use full-system garbage collection and policies for managing memory re-balancing in optimization of global system goals. We show that the design effectively trades memory (in pools for components) for efficiency, and that global policies manage this trade-off using a variety of metrics including system-wide throughput, and quality-of-service. We also show the effectiveness of zero-copy data-movement throughout the system using garbage-collected CBUFs. We show that existing legacy software (an image processing application, and a networking stack) can be adapted to use the CBUF API. A CBUF-enabled webserver, leveraging zero-copy communication between 6 components on the data-path, achieves over a factor of 2.5 throughput speedup while using less processing time than an existing webserver on Linux despite the increased isolation in an implementation that uses many protection domains.

Acknowledgments. We'd like to thank all of the anonymous reviewers and our Shepherd Peng Wu for their valuable comments and suggestions. These have, no doubt, improved the quality of this paper.

References

[1] G. Banga, P. Druschel, and J. C. Mogul. Resource containers: a new facility for resource management in server systems. In *Proceedings of the third symposium on Operating systems design and implementation (OSDI)*, 1999.

[2] A. Baumann, C. Hawblitzel, K. Kourtis, T. Harris, and T. Roscoe. Cosh: Clear os data sharing in an incoherent world. In *2014 Conference on Timely Results in Operating Systems (TRIOS 14)*, 2014.

[3] A. Belay, G. Prekas, A. Klimovic, S. Grossman, C. Kozyrakis, and E. Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, 2014.

[4] A. Burns and R. Davis. Mixed criticality systems – a review, retrieved feb, 2016. *https://www-users.cs.york.ac.uk/burns/review.pdf*, 2016.

[5] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[6] P. Druschel and L. L. Peterson. Fbufs: A high-bandwidth cross-domain transfer facility. In *Symposium on Operating Systems Principles*, pages 189–202, 1993.

[7] J. Evans. Scalable memory allocation using jemalloc, 2011. *URL https://www. facebook. com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919*, 2016.

[8] FastCGI. FastCGI: http://www.fastcgi.com, 2016.

[9] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. *goog-perftools. sourceforge. net/doc/tcmalloc. html*, 2009.

[10] S. Han, S. Marshall, B.-G. Chun, and S. Ratnasamy. Megapipe: A new programming interface for scalable network i/o. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, 2012.

[11] J. Hwang, K. K. Ramakrishnan, and T. Wood. Netvm: High performance and flexible networking using virtualization on commodity platforms. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, 2014.

[12] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: Formal verification of an OS kernel. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles*, Big Sky, MT, USA, Oct 2009. ACM.

[13] lwip. lwIP: http://www.sics.se/ adam/lwip/, retrieved 7/15/11, 2016.

[14] F. Mehnert, M. Hohmuth, and H. Härtig. Cost and benefit of separate address spaces in real-time operating systems. In *In Proc. of the 23rd IEEE Real-Time Systems Symposium (RTSS)*, December 2002.

[15] opencv. OpenCV: http://opencv.willowgarage.com/wiki/, retrieved 7/15/11, 2016.

[16] V. S. Pai, P. Druschel, and W. Zwaenepoel. Io-lite: a unified i/o buffering and caching system. In *OSDI '99: Proceedings of the third symposium on Operating systems design and implementation*, pages 15–28, 1999.

[17] G. Parmer. *Composite: A Component-Based Operating System for Predictable and Dependable Computing*. PhD thesis, Boston University, 2009. Adviser-Richard West.

[18] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. Scalable locality-conscious multithreaded memory allocation. In *Proceedings of the 5th International Symposium on Memory Management*, 2006.

[19] R. K. Treiber. Systems programming: Coping with parallelism. *Technical Report RJ 5118, IBM Almaden Research Center*, 1986.

[20] Q. Wang, J. Song, G. Parmer, G. Venkataramani, and A. Sweeney. Increasing memory utilization with transient memory scheduling. In *Proceedings of the 33rd IEEE Real-Time Systems Symposium (RTSS)*, 2012.

[21] Q. Wang, J. Song, G. Parmer, J. Wittrock, Y. Y. Wu, and T. Hossain. Hijack$^{cos}_{linux}$: Toward practical, predictable, and efficient OS colocation using Linux. In *Real-Time Linux Workshop*, 2012.

[22] Q. Wang, Y. Ren, M. Scaperoth, and G. Parmer. Speck: A kernel for scalable predictability. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2015.

[23] Q. Wang, T. Stamler, and G. Parmer. Parallel sections: Scaling system-level data-structures. In *Proceedings of the ACM EuroSys Conference*, 2016.