# *sumC*: A VM-based Scalable Unbounded Multicast Channel for Pipeline Parallelism

#30

## Abstract

Pipeline parallelism is an important parallel programming pattern. However, as the core counts increase, even highly-efficient algorithms and data structures can cause bottlenecks in pipeline parallelism.

In this paper, we investigate two representative pipeline applications, dedup and ferret. Our investigation indicates that multithreaded programs with hybrid shared mutable data structures do not scale on multicore. They suffer from serious lock contention and frequent inter-processor interrupts (IPI) when the core count exceeds 8. We then present easy-to-use ***sumC*** (scalable unbounded multicast channel) multithreaded programming model as well as its VM based reliable and scalable runtime system to support efficient pipeline parallelism. We check the programmability by successfully converting both dedup and ferret using ***sumC***. Our evaluations show that the ***sumC*** multicast channel has good scalability, and has nearly 2.05X faster than dedup built with jemalloc, a scalable memory allocator, on 32 CPU cores.

## 1. Introduction

With the explosion of multicore processors, performance and ease of programming for a single shared memory system have been critical to the modern computing. As core counts increase, even highly-efficient algorithms and data structures can become bottlenecks due to the cost of synchronization and communication. Developing well-designed concurrent data structures, although difficult, has become an effective way to increase productivity and improve performance.

Since pipeline parallelism [8] is an important parallel programming pattern for streaming and other emerging applications, we focus on concurrent data structures for pipeline parallelism. To understand the feature of concurrent data structures and the cost of synchronization and communication, we investigate two representative pipeline applications – contented-based image retrieval (`ferret`) and compression (`dedup`) and – from PARSEC [1], and evaluate them on a 32-core Linux system (see Section 2 in detail). The main shared data structures used in them are lock-based FIFO queues and a hash table with per-entry mutex, referencing a large number of heap objects. Our investigation indicates current multithreaded programs with hybrid shared mutable data structures do not scale on multicore. They suffer from serious *lock contention* and frequent *inter-processor interrupts* (IPI) when the core count exceeds 8, e.g., `dedup` built with `glibc` occupies 11.7% and 23.3% of total runtime on 16 and 32 CPU cores, respectively. Further evaluation on `dedup` built with `jemalloc` (denoted as `dedup-j`), a scalable memory allocator for multicore system, indicates that the locking overhead can be significantly reduced to less than 1% of total runtime on 32 cores. However, IPI overhead still increases with more cores, occupying more than 4% of total runtime, making `dedup` slow down on 32 cores.

Except lock-based queues, nonblocking queues [7, 9, 10, 14] have been studied for decades. These designs, whether lock-based or not, provide parallelism via fine grained synchronization among threads, and do not scale past a small amount of concurrency because threads contend on the queue's head and tail [6]. Moreover, due to limited buffer size, a producer or consumer of a queue may be blocked or wait as the queue is full or empty.

Besides, some applications require reliable multicast or a splitter of *Duplicate* type mentioned in data flow languages [13], where sender(s) transmit data to a group of receivers in a reliable manner. Commonly used algorithms for multicast such as MPI broadcast are binormial tree algorithm for short messages, and the Van de Geijn algorithm for long messages used in MPICH [12]. These algorithms still have at least a logarithmic latency term.

Our goal with this paper is to characterize the performance requirements and considerations of concurrent FIFO channels to support either multicast or unicast, and to create a scalable multicast channel for high throughput. Specifically, this paper makes the following contributions:

1. We investigate two representative pipeline applications using Pthreads, and analyze how shared data structures

and memory allocators influence the performance of the two applications on multicore system.

2. We put forward the design goals of scalable unbounded multicast channels (***sumC***) to support efficient pipeline parallelism, and present the ***sumC*** multithreaded programming model. The model is C library-based and distinct from traditional thread model sharing address space. It only allows threads to share channels. We further successfully converted `dedup` and `ferret` using ***sumC***.
3. We propose a set of virtual memory (VM) based approaches to build the ***sumC*** runtime in Linux user space. The ***sumC*** runtime integrates synchronization into the VM by distinguishing sender(producer) mapping and receiver(consumer) mapping of a shared page frame, and provides extension mechanism via memory remapping for unbounded multicast.
4. We evaluate the ***sumC*** using a simple `bcast` program using ***sumC***, and various versions of the original and converted `dedup` and `ferret`. Evaluations show that the ***sumC*** multicast channel has good scalability within a CPU processor, and the little overhead of extension allows to establish more channels with smaller channel size for communication among more threads. Furthermore, the converted `dedup` has better scalability than `dedup-j` and has nearly 2.05X faster than `dedup-j` on 32 cores.

The rest of the paper is organized as follows. Section 2 describes the case study of pipeline applications. Sections 3 and 4 describe the ***sumC*** programming model and its runtime system, respectively. Section 5 evaluates the prototype. Section 6 summarizes relevant work and Section 7 concludes.

## 2. Case Study

There are three benchmarks in PARSEC (Table 1) that exhibit pipeline parallelism in different forms using Pthreads and provide a good example of the cases that programmers might encounter. Since `x264` has the most source lines of code and its pipeline feature is not obvious, we do not investigate it. We investigated `dedup` and `ferret`, including parallel feature analysis and performance analysis.

| Program | SLOC | Description |
|---|---|---|
| dedup | 2566 | A compression kernel that uses the "de-duplication" |
| ferret | 10769 | A content-based image search application |
| x264 | 29328 | An HD video encoder for the H.264/MPEG4 standard |

**Table 1.** Pipeline Parallel Programs in PARSEC v2.1.

### 2.1 Parallel Feature Analysis

`dedup` is a 5-stage pipeline, while `ferret` has a 6-stage pipeline. Both of them have a serial input stage and a serial output stage. The main thread in each program creates one thread for each serial stage, $n_t$ threads for each parallel stage, and joins all stage threads when they finish. The shared data structures in the two programs are as follows.

**Queue.** Both `dedup` and `ferret` use a set of circular queues for both communication between two adjacent stages and load balancing. Queues in `ferret` are relatively simpler, each of which has one producer and one consumer. However, the producer-consumer relationship of queues in `dedup` might be 1:$m$, $m$ : $m$ or $m$ : 1. Each queue is synchronized by using a mutex to avoid data races, and using conditional variables for *empty/full* blocking.

**Hash Table.** Only `dedup` maintains a shared hash table with per-entry mutex to store unique fingerprints of data chunks, which are operated by the last three stages. To coordinate among the three stages, each hash value has a conditional variable *empty*, which is initialized by a ChunkProcess thread after the hash value is created, and signaled by a Compress thread to notify the SendBlock thread that the data has been compressed.

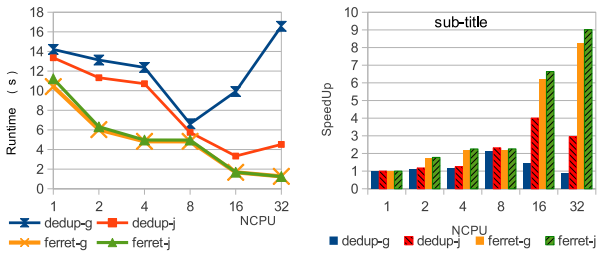
**Pointers to Heap Objects.** Both `dedup` and `ferret` widely represent data items in the queues and hash table as pointers, accordingly indirectly sharing heap objects among threads. These data structures are often called mutable data structures. Using pointers may reduce data copy overhead, but make data sharing more implicit and error-prone.

### 2.2 Performance Analysis

To understand the cost of synchronization and communication, we evaluate original `dedup` and `ferret` in Pthreads using the *large* test input on a 32-core Intel 4× Xeon E7-4820 system equipped with 128GB of RAM. The operating system is Ubuntu 12.04 with kernel 3.2.0 and `glibc-2.15`. Benchmarks were built as 64-bit executables with `gcc -O3`.

We logically disable CPU cores using Linux's CPU hotplug mechanism, which allows to disable or enable individual CPU cores by writing "0" (or "1") to a special pseudo file (/sys/devices/system/cpu/cpuN/online), and the total number of threads was matched to the number of CPU cores enabled. Each workload is executed 10 times. To reduce the effect of outliers, the lowest and the highest runtimes for each workload are discarded, and thus each result is the average of the remaining 8 runs.

We first evaluate `dedup` and `ferret` built with default C library–`glibc`, denoted as `dedup-g` and `ferret-g`, respectively. Fig.1 presents their performance at various core counts. From the runtime overhead shown in Fig. 1(a), we observe that both `ferret-g` and `dedup-g` run faster when the core count (`NCPU`) increases from 1 to 2, and smoothly from 2 to 8 for `ferret-g`, 2 to 4 for `dedup-g`. That is because in each of the above cases, numbers of threads in each parallel stage ($n_t$) are all 1. When `NCPU` increases from 8 to 16 to 32 with $n_t$ increasing from 1 to 3 to 7, `ferret-g` continues to run faster, but slows down to speed up from 16 to 32 cores. `dedup-g` runs faster from 4 to 8 cores with $n_t$ increasing from 1 to 2, however slows down significantly from 8 to 16 to 32 cores with $n_t$ increasing from 2 to 4 to 9. As




(a) (b)

**Figure 1.** Performance of dedup and ferret. (a) Runtime overhead. (b) Parallel speedup over its own single-CPU performance.

shown in Fig. 1(b), `ferret-g` scales well as the core count increases, but `dedup-g` does not scale.

The above performance changes can be explained from the shared data structures used ( see Section 2.1). `ferret` only uses some 1:1 concurrent queues to transfer data from one stage to the next stage, while `dedup` may cause many contention situations, such as multiple consumers contending one queue, multiple producers contending one queue, multiple threads contending one hash entry, and so on. The significant slowdown of `dedup-g` from 8 to 16 to 32 cores indicates that programs with hybrid shared mutable data structures do not scale on current multithreaded system.

Since there are many heap objects shared among threads in `dedup`, and `ptmalloc` [4], the memory allocator in `glibc`, does not scale on multicore system, we then build the two programs with `jemalloc-3.5.0` [2], a scalable memory allocator for multicore system, denoted as `dedup-j` and `ferret-j`. As shown in Fig. 1, `dedup-j` scales better than `dedup-g`, but still slows down when running from 16 to 32 cores, while `ferret-j` performs on par with `ferret-g` and scales a little better than `ferret-g` on more than 8 cores.

To inquire the reason of the slowdown of `dedup`, we further use Linux Perf to profile `dedup` at 8, 16 and 32 cores. Table 2 lists cycles of default events sampled by Perf (listed in Columns 2 and 3), as well as the runtime overhead percentages (listed in Columns 4–9) of the overall samples collected in three significant functions, *i.e.,* deflate_slow(), __ticket_spin_lock() and _IPI_mask_sequence_phys(). We call them deflate, lock, IPImask in turn for short. The deflate function comes from zlib library in PARSEC and performs main job of `dedup`—deflation. Its percentage reflects the rate of computation. The lock and IPImask functions come from Linux kernel, which perform locking and sending an IPI mask to all processors, respectively. The percentages of these two functions reflect the rate of contention and the rate of overhead across processors, respectively.

From the table, we observe that `dedup-g` costs more event cycles than `dedup-j`, and reaches 2.56X, 4.56X of its own 8-core cycles at 16 and 32 cores, respectively. From the percentages given in the table, we realize that `dedup-g`

| NCPU | K cycles | | deflate(%) | | lock(%) | | IPImask(%) | |
|---|---|---|---|---|---|---|---|---|
| | G | J | G | J | G | J | G | J |
| 8 | 25 | 15 | 37.4 | 42.7 | 1.0 | 0.2 | 2.5 | 0.3 |
| 16 | 64 | 17 | 27.5 | 40.2 | 11.7 | 0.6 | 10.5 | 1.4 |
| 32 | 114 | 18 | 18.8 | 36.0 | 23.3 | 0.7 | 15.4 | 4.6 |

**deflate**: deflate_slow **lock**: __ticket_spin_lock
**IPImask**: _IPI_mask_sequence_phys **G**: dedup-glibc, **J**: dedup-jemalloc

**Table 2.** Runtime overhead of dedup profiled by Linux Perf.

suffers from serious lock contention and frequent interprocessor interrupts. The overhead of lock in `dedup-g` is increased from 1% on 8 cores to 23.3% on 32 cores, while correspondingly the overhead of IPImask is increased from 2.5% to 15.4%. These two occupy 38.7% of total runtime, far more than the overhead of deflate—18.8%. `dedup-j` significantly reduces locking overhead, making it less than 1% of total runtime even on 32 cores. However, the IPImask overhead continues to increase from 1.4% on 16 cores to 4.6% on 32 cores, and is the main factor of performance slowdown when running from 16 cores to 32 cores. The IPI actions are mainly triggered by flushes of memory management unit (MMU) caches on other processors when memory mappings are changed by one processor. The percentage increase of IPImask in `dedup-j` indicates that a scalable memory allocator cannot make a program with arbitrary heap objects shared among threads scale well.

## 3. A Programming Abstraction for Multicast

Based on the investigation of pipeline applications, we aim at providing an easy-to-use programming abstraction to support scalable multicast (unicast is a special case of multicast) communication, called ***sumC*** (***s***calable ***u***nbounded ***m***ulticast ***C***hannel).

### 3.1 Design Goals

Goals motivating the design of the ***sumC*** are as follows:

1. *Multicast communication*: *i.e.,* dissemination-oriented communication, that which a sender sends is transmitted to a group of receivers.
2. *Loose-coupling between sender and receivers*: a channel's sender and receivers need not know each other. The channel is the only common object that both sender and receivers are aware of, reducing the complexity of connection establishment.
3. *Declarative programming interface*: that is used to describe the send-receive (or produce-consume) relationship of workers (threads or processes), and provides easy-to-use send and receive functions enforcing the prespecified send-receive relationship at user-defined message granularity.
4. *Heterogeneity*: messages which a sender sends to a channel can have different sizes, and each receiver can reliably get the size and content of each message in turn.

5. *Unboundedness*: that a channel has unbounded size of communication buffer. Thus a sender can send any number of messages without blocking or waiting.
6. *Reliability*: that each receiver can get access to a message only after it has already been published (made visible) by the sender.
7. *Asynchronism*: that both sender and receivers can send or receive messages at their own paces under the above *Reliability* goal.

### 3.2 Programming Abstraction: *sumC*

We now describe the basic concepts of the ***sumC***. A *channel* is one-to-many communication abstraction which transfers messages between a sender (producer) and multiple receivers (consumers). A channel is created by a worker, making the worker become the sender. Workers refer to both processes and threads for the ease of discussion.

```
int chan_alloc()
        Allocate a channel and return its ID.
int chan_setprod(int chan, int child, bool ascons)
        Transfer the send-port of the channel to the given child.
int chan_setcons(int chan, int child)
        Assign a receive-port of the channel to the given child.
size_t chan_send/chan_sendLast(int chan, void *buf, size_t sz)
        Send a message stored in buf of sz bytes via the channel.
size_t chan_recv(int chan, void *buf)
        Receive a message from the channel and save it in buf.
```

**Figure 2.** Main functions of ***sumC*** channel API.

A *port* is an access point to a channel. A channel can have a single port to send, and any number of ports to receive. A worker can dynamically invoke chan_setprod to transfer the single send-port of a channel which it owns to its child thread, making itself occupy a receive-port or not. A worker can also dynamically invoke chan_setcons to assign a receive-port of a channel it owns to its child worker. By invoking these two parent-to-child functions, the send-receive relationship of a channel can be set up among a group of workers. Parent-to-child port transfer makes ***sumC*** preserve the conceptual simplicity, control and composition power of strictly hierarchical "nested process model" [3]. Once the channel relationships are set up, workers can invoke chan_send or chan_recv to send or receive messages in these channels incrementally, allowing for arbitrary peer-to-peer "dialog" short-cutting the worker hierarchy. A worker can invoke chan_sendLast to notify the ***sumC*** implementation that this is the last message sending to the channel. Fig. 2 lists main functions to set up and operate ***sumC*** channels.

### 3.3 A *sumC* Thread Model

We can construct either process-level or thread-level parallelism via ***sumC*** channels. In this paper, we focus on the thread-level parallelism. The ***sumC*** thread model is a major departure from more traditional ones such as Pthreads,

```
int thread_alloc(int gid)
        Allocate a child thread of global ID and return its internal ID.
int thread_start(int child, void *(*fn)(void*), void *args)
        Start the given child to run (*fn)(args).
void thread_join(int child)
        Wait for the child to terminate, merging the control flow.
int thread_ret()
        Return to its parent thread.
int thread_sync(int child)
        Wait for the child to return, then restart the child again.
```

**Figure 3.** Main functions of ***sumC*** thread API.

in that each ***sumC*** thread can only share channels with other threads, but isolate other memory from other threads. The initial state of isolated memory in a thread is inherited from its parent thread when it is started. Currently, there is no shared heap among threads, but private heap for each thread. We offer basic threading functions shown in Fig. 3 to construct fork-join parallelism and parent-child barriers.

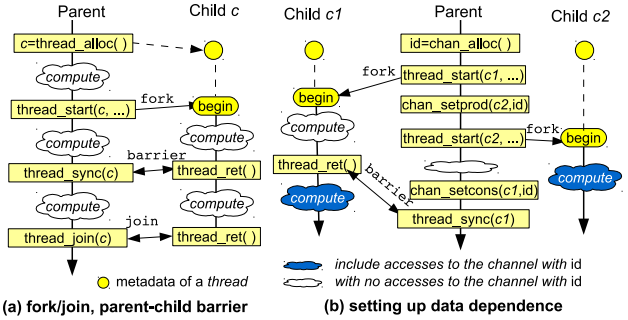



**Figure 4.** Synchronization among ***sumC*** threads.

**Fork/Join, Parent-child Barrier:** As shown in Fig. 4(a), a thread can call thread_start(child, f, args) to copy its entire memory state except the channel state into the child, set up the child's registers, and start the child to run f(args). Each thread_sync(child) call in a thread should have a matched thread_ret call in its specified child thread, forming a synchronization pair between parent and child. A thread_join(child) call waits for the exit of the child, and merges the control flow without copying any memory state from the child.

**Creating and Operating a *sumC* Channel Reliably:** For simplicity, a thread can create a channel and set up the send-receive relationship of the channel between itself and its kids before starting these kids. After setting up, the group of threads can interact with each other according to the specified send-receive relationship. This use mode can represent master-slaves thread relationship used in many applications such as `dedup`.

To support more flexible use modes such as the case of setting up a channel among active threads, we need to create parent-child barriers like the barrier shown in Fig. 4(b) to serialize the setting-up stage and the later send/receive stage. In this subfigure, the parent assigns a receive-port of the channel id to an active child c by invoking chan_setcons(c1,

id). To prevent child c1 from receiving incorrect messages due to the incompletion of setting up, child c1 needs to invoke thread_ret() to suspend itself and wait for the parent to wake up via the parent's thread_sync(c1) invocation.

### 3.4 Programmability

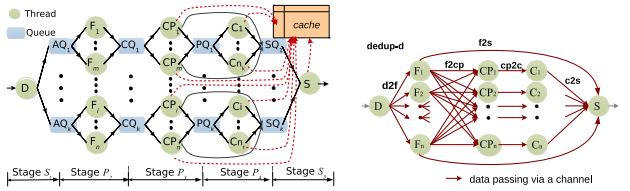
The proposed ***sumC*** programming model satisfies Goals 1–3 mentioned in Section 3.1 in aspects of programming interface and high-level semantics. Thereinto, reflections of Goals 1 and 3 are straightforward, while Goal 2 is reflected in that each thread only knows kids to which it once transferred or assigned a port of a channel, but is not aware of the final group of threads sharing the channel. The other 4 goals would be achieved by the ***sumC*** runtime system mentioned in Section 4.

From Sections 3.2 and 3.3, we realize that current ***sumC*** multithreaded programming model cannot allow threads to share mutable heap objects and other data structures such as a hash table used in `dedup`. So, could the ***sumC*** programming model express actual pipeline computations like `dedup`? By deeply analyzing the usage of concurrent data structures and heap objects, we argue that ***sumC*** can express many pipeline parallel features:

1. **Any $m$:$n$ queue can be replaced with several *sumC* channels.** For FIFO queues, each item in a queue would be consumed only by one consumer. Any 1:$m$ or $m$:1 queue can be replaced with $m$ channels of a single send-port and a single receive port (called *unicast* channel, a special multicast channel). And the single sender (or receiver) would send or receive items on the $m$ channels in a round-robin fashion. Any $m$:$m$ queue, for simplicity, can be replaced with $m$ unicast channels by matching each sender to only one receiver. Thus a receiver can directly process items sent from the matched sender.

2. **A shared hash table can be split into several thread-local hash tables via a schedule change.** As shown in Fig. 5(a), the hash table cache in `dedup` makes more interdependence between stages, no matter whether they are adjacent or not. Each F thread splits a received data block into several smaller chunks which would be sent to next CP threads in a round-robin fashion. Each CP thread computes the hash value of each chunk and searches the hash table cache to decide whether to insert a chunk. Each C thread compresses a received chunk and searches cache to decide where to insert the compressed data. The S thread checks cache to acquire the compressed data and identify whether it has been written.

We think the shared hash table can be privatized by separating it into $n$ sub-hashtables, and let each CP or C thread deal with items whose hash values are in the range of the same sub-hashtable. We further move the hash value computation from CP stage to F stage, each F thread decides which CP thread would receive a chunk according to its hash value, and each CP thread decides whether a chunk need be compressed by checking and maintaining its own sub-hashtable.




(a) Original schedule (b) Deterministic schedule

**Tasks**:
D - DataProcess, F - FindAllAnchor, CP - ChunkProcess, C - Compress, S - SendBlock
**Queues**:
AQ - anchor_que, CQ - chunk_que, PQ - compress_que, SQ - send_que

**Figure 5.** Program architecture of dedup.

To eliminate that the S thread depends on the hashtable, each chunk item received by the S thread should contain its compressed data or its fingerprint. Fig. 5(b) shows the changed schedule, which is also deterministic, that is, each thread always deals with the same data items in different runs when using the same input.

3. **Shared pointers can be eliminated by transferring the contents, not the pointers, via channels.** This conversion may lead to more data copy overhead, but decrease the interdependence among threads, potentially eliminating contentions on these shared pointers.

We successfully convert both `dedup` and `ferret` using the ***sumC*** programming model, where low-level synchronization constructs such as mutex and conditional variables are removed. The converted programs denote as `dedup-d` and `ferret-d`, respectively. Table 3 gives the `SLOC` comparisons between the converted version and the original one. From the table, we know that the amount of modification is very small, and the converted version has smaller code size than the Pthreads one.

| Versions | Same | Modified | Added | Removed |
|---|---|---|---|---|
| dedup-d vs. dedup | 1863 | 121 | 150 | 364 |
| ferret-d vs. ferret | 8738 | 63 | 98 | 254 |

**Table 3.** Code size comparisons.

## 4. The *sumC* Runtime System

In this section, we introduce our key designs of the ***sumC*** runtime to achieve Goals 4–7 listed in Section 3.1, and have implemented them totally in Linux user space. We emulate a ***sumC*** thread using a single threaded Linux process to isolate memory except channels among ***sumC*** threads. Next, we focus on designing novel memory mapping technology on Linux processes to implement ***sumC*** channels.

### 4.1 Basic VM-based Idea for Reliable Multicast

***Multicast Virtual Memory.*** To ensure each receiver gets accesses to messages sent by the sender in turn, the following two kinds of data races should be avoided by the ***sumC*** runtime:

(1) RAW (*read-after-write*) races, where receiver A attempts to fetch a result that has not yet been yielded by sender B.

(2) WAR (*write-after-read*) races, where sender A attempts to update the data that has not yet been fetched by receiver B.

To support scalable multicast, we establish multicast virtual memory regions among Linux processes. As shown in Fig. 6, each region has a corresponding virtual address range in each relevant process's address space, all of which are mapped to the same shared page frames.

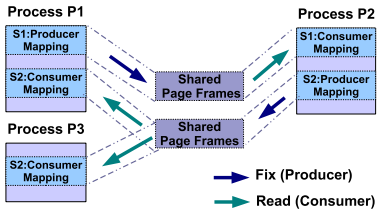



**Figure 6.** Processes sharing multicast regions.

(1) To avoid RAW races, any receiver attempting to read a shared object before the object is ready shall be blocked or wait until the sender completes writing and publishes (called *fixes*) the memory region occupied by the object. Receivers will thereafter be able to read the fixed and shared page frames concurrently.

(2) To avoid WAR races, each shared page frame mapped into both the sender's and receivers' segments can only be fixed at most once. Once a page frame is fixed, the sender will lose the write permission on it.

***Distinguishable Memory Mappings for Multicast.*** We introduce two types of mappings, *i.e.,sender(producer) mappings* and *receiver(consumer) mappings* to synchronize the mapped shared page frames. A producer mapping has at least two kinds of states—*invalid* and *writable*, where the former means there is no real page frame corresponding to it, and the latter means the mapped page frame is writable. A consumer mapping also has at least two kinds of state, *i.e.,invalid* and *readonly*, where the former means there is no real page frame or the mapped page frame is not fixed, and the latter means the mapped page frame has been fixed.

***Synchronization on Multicast Virtual Memory.*** Any attempts to access a virtual page of invalid producer/consumer mapping or write to a virtual page of readonly consumer mapping will raise a page fault.

Upon a producer page fault, the fault handler will allocate a real page frame and update the faulting page with a writable producer mapping so that the producer can write the page afterwards. When a consumer attempts to read a page with invalid mapping, the fault handler will check whether the target page frame is present and further fixed or not: if fixed, the faulting page will be mapped to the frame, and updated with a readonly mapping; otherwise, the consumer will be blocked to avoid RAW races.

A producer must explicitly call setfixed multicast VM primitive to make page frames mapped into a given multicast segment visible to consumers. Any blocked consumer waiting for one of the page frames should be waken up. Each mapping in the segment becomes readonly consumer mapping to avoid WAR races.

### 4.2 Unboundedness of Multicast Regions

Unboundedness goal is the key to achieve high throughput, and also influences other goals such as asynchronism and heterogeneity. In order to let a multicast region of limited address range save unlimited amount of data, we design an `extend` mechanism which allows remapping a producer's or a consumer's segment to a new generation of page frames.

***The `extend` Primitive.*** When a producer expects to rewrite a multicast region, it can call `extend` primitive to remap its segment to new page frames where new data will be updated, without changing the old page frames that consumers may still require. After a consumer reads the data on the page frames mapped into its segment, it can call `extend` to find the next generation of page frames generated by the producer, and remap its segment to the newer page frames. The older page frames decrease their reference counts and are freed automatically when the counts reach zero.

***Challenges to Maintain Produce-Consume Relationship.*** Goal 2 provided by the ***sumC*** programming model causes a producer or consumer of a channel does not know which thread shares the channel with it. Although the ***sumC*** runtime can gradually establish complete produce-consume relationship by tracking the chan_setprod and chan_setcons calls, once a producer invokes setfixed primitive to make page frames visible, it would lose its producer mappings and become a consumer, accordingly missing the preset produce-consume dependence.

***The extension Page.*** To record the produce-consume dependence and trace generations of page frames among the sharing parties, a special anchor—*extension page*—is introduced and shared between the producer and consumers. When a producer calls extend(eva,va, size) primitive to remap the specified segment to a new generation of page frames, the remapping information would be saved into the specified extension page starting from address eva in the segment. Thus a consumer can invoke extend primitive afterwards to get the remapping information via the extension page, and then remap its segment to the next generation of page frames got from the remapping information.

The producer and consumer must agree on which pages in the multicast region are regular data pages and which are extension pages. Both the producer and consumer in application code should not attempt to read or write an extension page (doing so causes a page fault). The page fault handler will then report an error.

### 4.3 Building Multicast VM in Linux User Space.

We first introduce the concept of MMU for multicast VM, then present how to build the MMU in Linux user space.

***Lazy Page Mapping for Multicast VM.*** Page frames for a multicast region are assigned only "on-demand" once they are actually touched by applications—*lazy-page-mapping*. This lazy-page-mapping causes a page frame is lazily allocated upon a producer write fault, thus each consumer cannot get the page frame at the time when it gets the receive-port via its parent's chan_setcons call. How can a consumer find lazily allocated page frames when needed? We introduce an *anchor page table* to solve this problem.

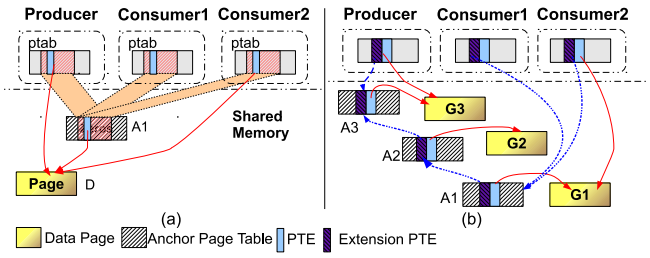



**Figure 7.** Lazy-tree-mapping mechanism.

Initially a sequence of pages in a newly created SPMC segment are mapped to a special page frame—anchor page table (A1 in Fig. 7(a)), in which each page has a corresponding page table entry (PTE). When a virtual page is mapped to a newly allocated page frame (D in Fig. 7(a)) upon a producer write fault, the corresponding entry in the anchor page table is updated to point to the page frame. Thus upon a consumer page fault, the fault handler can check the anchor page table to find the page frame and map the faulting page to the page frame. Fig. 7(a) shows the moment that Consumer2's page fault already happened, but Consumer1's has not yet.

***Extension via Tree Mapping.*** The key issue in implementing `extend` primitive is how a consumer finds new generation of page frames in production order?

We address the issue by reserving some virtual pages in a multicast region as *extension pages*, each of which is used to extend a separated segment it belongs in the region. Initially an extension page and each page in its corresponding segment are mapped to the same anchor page table (A1 in Fig. 7(b)). When the producer invokes `extend` primitive to remap the segment via the extension page, a new anchor page table is created to save entries pointing to a new generation of page frames lazily allocated in the future. The new anchor page table is then mapped into each page of the producer's segment. A special entry in the old anchor page table, to which the extension page corresponds, is set to point to the new anchor page table. When a consumer invokes `extend` primitive, the handler can follow the chain led by the extension page to find the new generation(s) of page frames. Fig.7(b) shows the state after Producer completes the third `extend` operation, where A1 to A3 represent anchor page tables of the first to third generations in turn, and G1 to G3 are the first to third generations in turn; at the moment, Consumer1 has not read the first generation of data (*i.e.,* page G1), and Consumer2 just read G1. We see that starting from each producer or consumer's last-level page table (as the root of a tree), through pointers in the extension PTEs, the pointed anchor page tables as well as data pages form other nodes of the tree, which represent generations being accessed or to be accessed.

All page frames including data and anchor page frames are reference-counted, so an anchor page table and its referred page frames will get garbage-collected once their producer and consumers need them no longer, *i.e.,* their reference counts reach zero.

***Page Table Management.*** Different OSes often adopt different multi-level page tables to store the mapping between virtual pages and page frames. No matter what multi-level page tables are, the key of integrating lazy-tree-mapping mechanism into a VM system is the design of the lowest-level page table entry and its state transformation.

To support multicast page mapping, each PTE contains a pointer to a page frame, either a data page frame or an anchor page table. A PTE need have at least three bits indicating the mapping state: the `P` bit indicates whether the page frame is valid, the `W` bit indicates whether the page frame is writable or readonly, and the `O` bit indicates whether it is a producer mapping. Each page frame has a page struct to hold metadata on how it is used, including reference count, whether it is a data page or not, who is the producer, who are waiting for the page, etc.

***Building MMU in Linux User Space.*** Since the ***sumC*** runtime is built in Linux user space, it cannot control actual page mapping in Linux kernel. So the ***sumC*** runtime creates two chunks of `mmap` shared memory to emulate page frames (EMEM) and per-space page tables. Each page frame has a corresponding per-page metadata. The runtime uses a simple page frame allocator by maintaining a single freelist of page frames. Each process's page table for multicast VM is just one-level page table, mapping a process's multicast virtual pages to emulated page frames. Each virtual page has a 32-bit PTE in the per-process page table, which consists of `P`, `W`, `O` bits and a 28-bit address indicating the mapped page frame. Thus a 4KB-page-frame can at most hold 1024 entries representing 4MB address range, supporting up to $2^{28}$ 4KB-page-frames.

Due to the emulation in user space, any write to a multicast region is emulated by a write to a producer's multicast virtual page and a write from the virtual page to the emulated page frame; while any read from a consumer's multicast virtual page also causes a write from the emulated page frame to the virtual page, leading to major memory copy overhead. The ***sumC*** runtime controls access privilege to multicast regions via disciplined use of `mprotect` system calls. Thus some read/write operations on multicast regions from appli-

cation code might trigger segmentation faults, the control is then transferred to the segmentation fault handler, emulating page faults and their handlers described previously.

### 4.4 Heterogeneity and Asynchronism of Channels

A ***sumC*** multicast channel internally contains a multicast VM region to store messages sent by its sender. We denote the size of address range occupied by an internal multicast region as *channel size*, which is a fixed-size such as 512KB. The multicast VM only enforces reliable produce-consume relationship at page granularity, However, the multicast channel should enforce reliable send-receive relationship at user-defined message granularity. To achieve the goal of heterogeneity, *i.e.,* to support messages with different sizes, a message stored in an internal multicast region consists of its real length and its content, and we need to design strategy to fix (publish) a message sent by a sender.

***Fix a Message Eagerly or Lazily.*** If fixing a message immediately after it is written to the internal region—*eager-fix*, the item could be consumed in time. This policy requires to put a message on a page-aligned virtual address range. Even if a message is not filled in a page, the page cannot save any other message, accordingly increasing the demands for more pages.

In some cases, we can store several short items on the same page, that is, fixing a page only when the page is filled or when the user explicitly requires to fix a message in time—*lazy-fix*. This policy increases the utilization rate of page frames, but also increases the message access latency. Moreover, an additional argument indicating whether to fix in time need be introduced into the send function, *i.e.,*sendLast accordingly restricting programming.

***Support Large Messages in a Channel.*** Since a channel receiver provides a buffer to receive the next message, the `send` implementation can split any large message into several sub-messages, which can be put on the multicast region in turn. The `recv` implementation can then get each sub-message in production order and copy them to the private buffer one by one. This split mechanism also helps to achieve pipeline parallelism.

To let consumers know the length of each message, a message—which might be a sub-message—is stored on the multicast region with an internal header, *i.e.,* $(a, b)$, where $a$ and $b$ indicate the message length and the whole message length, respectively. To reuse multicast segment for unlimited data transfer, we arrange the last page in an internal multicast region as an extension page.

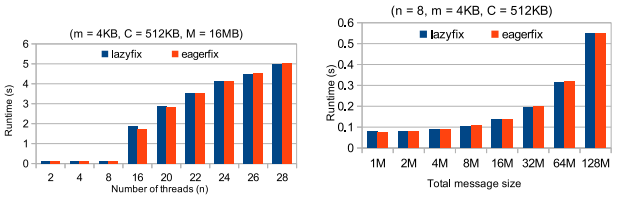
***Asynchronism of Channels.*** We introduce per-thread metadata for each active channel to record its status such as offset at which a sender puts or a receiver gets the next message in the internal multicast region, so that each sender or receiver can put or get items at their own pace. The send and recv handlers can invoke `extend` primitive to remap a multicast segment to new page frames, so that the producer can continue rewriting the multicast segment, while consumers can still read old data however late it may be.

## 5. Evaluation

We perform our evaluation on a 32-core Linux system described in Section 2.2.

### 5.1 Capability of Multicast

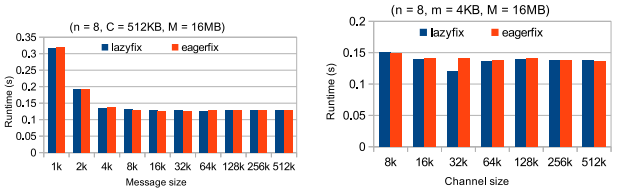
We write a simple multithreaded C program `bcast` using the ***sumC*** library, in which the master thread broadcasts total messages of $M$ (MB) with message size $m$ (KB) to $(n-1)$ slave threads. Assume the channel size is $C$ (KB). We evaluate how various factors influence the performance of multicast implemented with eager or lazy fix strategy.




(a) (b)

| | | 1K | 2K | 4K | 8K | 16K | 32K | 64K | 128K | 256K | 512K |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **eager** | $n_e$ | 132 | 65 | 32 | | | | | | | |
| | $n_f$ | 16384 | 8192 | 4096 | | | | | | | |
| **lazy** | $n_e$ | 32 | | | | | | | | | |
| | $n_f$ | 4096 | | | | | | | | | |

(c) Extension counts and fix counts in different message sizes




(d) (e)

**Figure 8.** Performance of multicast influenced by various factors.

***Performance Influenced by Number of Threads*** $n$**.** Fig. 8(a) presents the runtime of `bcast` influenced by the number of threads, in which the message size ($m$=4KB), channel size ($C$=512KB) and total message size ($M$=16MB) are fixed. We observe that runtimes of broadcast within no more than 8 threads are similar, no matter whether using eager or lazy fix strategy, reflecting good scalability when multicasting within a CPU processor. However, the runtime overhead is significantly increased when the multicast is across processors. That is because our current implementation is not NUMA-aware, which will be further improved in our future work.

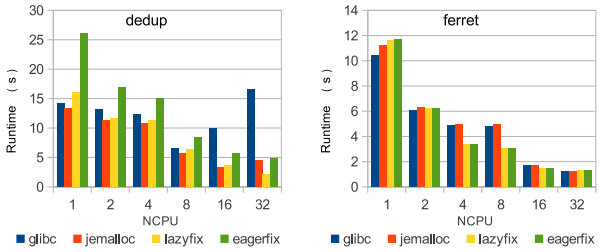
We next evaluate the performance of `bcast` within 8 threads.

***Performance Influenced by Total Message Size*** $M$**.** As shown in Fig. 8(b), `bcast` with $m$=4KB and $C$=512KB runs smoothly when total message size is below 8MB, and gradually slow down as $M$ increases, reflecting the more messages to be broadcasted, the longer the runtime. Since the message size is fixed to the page size, *i.e.,* 4KB, runtime of `bcast` with eager-fix is the same as that with lazy-fix.

***Performance Influenced by Message Size*** $m$**.** Fig. 8(c) lists the extension count ($n_e$) and fix count ($n_f$) of `bcast` with eager- or lazy-fix strategy, where $n_f$ under eager-fix is about 4X of that under lazy-fix if $m$=1KB. Although the fix or extension counts under eager-fix are not equal to those under lazy-fix, their runtimes shown in Fig. 8(d) are similar when other factors have the same configuration. But why does `bcast` with $M$=16MB and $C$=512KB run slower when $m$ is below 4KB and smoothly when $m$ exceeds 4KB? We think it is because moving page-aligned memory is faster than moving memory less than a page.

***Performance Influenced by Channel Size*** $C$**.** Fig. 8(e) presents the runtime of `bcast` with $m$=4KB and $M$=16MB in different channel sizes. It shows that although the smaller channel size causes more extension count, runtimes in different channel sizes have little difference. This indicates that we can configure smaller channel size to allow more channels for communication among more threads.

### 5.2 Performance of Pipeline Applications



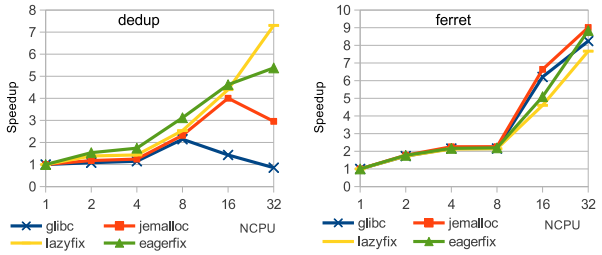

(a) (b)

**Figure 9.** Runtime overhead of various versions of dedup and ferret.

We next evaluate the performance of the two pipeline applications of various versions, including Pthreads versions built with glibc or jemalloc, and the ***sumC*** versions under eager- or lazy-fix strategy. The message size and channel size are fixed to 4KB and 512KB, respectively.

***Performance.*** Fig. 9(a) and (b) present the runtime overhead of various `dedup` and `ferret`. Runtimes of four `ferret` versions are similar on either 16 or 32 cores. The reason why two versions of our converted `ferret-d` run faster than two versions of Pthreads one at 4 or 8 cores is that the converted `ferret-d` merges the middle four stages into one bigger stage, avoiding communication within the four stages.

For `dedup`, the eagerfix version is the slowest one at all kinds of core counts; the jemalloc version has better performance than the lazyfix one on no more than 16 cores; but the lazyfix one has the best performance on 32 cores, nearly 2.05X faster than the jemalloc counterpart.




(a) (b)

**Figure 10.** Parallel speedup over its own single-CPU performance of various dedup and ferret.

***Scalability.*** Fig. 10(a) and (b) show each benchmark's speedup relative to its own single-CPU execution. For `ferret`, all four versions scale well, where the jemalloc one scales the best on 16 cores and the lazyfix one scales a little worse than the eagerfix one on 16 and 32 cores.

For `dedup`, both two versions of the Pthreads one scale worse when increasing from 16 cores to 32 cores, and two versions of the converted `dedup-d` have better scalability, where the lazyfix one is better than the eagerfix one.

***Message Volume through Channels.*** To understand why the lazyfix `dedup-d` runs better than the eagerfix one, we count messages through channels in both versions.

(a) All channels through which the maximum message size is 2.01 MB

| NCPU | fix | pages | | nchans | messages | | extension | |
|---|---|---|---|---|---|---|---|---|
| | | $n_p$ | E/L | | $n_m$ | total(MB) | $n_e$ | E/L |
| 1, 2, 4 | eager | 674151 | 4.02 | 5 | 564458 | 451 | 10612 | 4.03 |
| | lazy | 167537 | | | 564059 | 450 | 2636 | |
| 8 | eager | 674158 | 4.02 | 12 | 564488 | 452 | 10602 | 4.03 |
| | lazy | 167542 | | | 564064 | 450 | 2628 | |
| 16 | eager | 674178 | 4.02 | 32 | 564477 | 451 | 10592 | 4.06 |
| | lazy | 167553 | | | 564100 | 450 | 2606 | |
| 32 | eager | 674263 | 4.02 | 117 | 564621 | 452 | 10502 | 4.17 |
| | lazy | 167599 | | | 564223 | 450 | 2516 | |

(b) Some channels through which the maximum message size is 8-Byte

| NCPU | fix | pages | | nchans | messages | | extension | |
|---|---|---|---|---|---|---|---|---|
| | | $n_p$ | E/L | | $n_m$ | total(MB) | $n_e$ | E/L |
| 1,2,4 | eager | 215667 | 4.37 | 2 | 188124 | 181 | 2069 | 2.73 |
| | lazy | 49372 | | | 187985 | 186 | 759 | |
| 8 | eager | 215671 | 4.37 | 6 | 188116 | 186 | 2066 | 2.73 |
| | lazy | 49374 | | | 187985 | 186 | 757 | |
| 16 | eager | 215685 | 4.37 | 20 | 188152 | 186 | 2062 | 2.76 |
| | lazy | 49383 | | | 188008 | 185 | 746 | |
| 32 | eager | 215755 | 4.37 | 90 | 188249 | 186 | 2022 | 2.85 |
| | lazy | 49420 | | | 188109 | 186 | 709 | |

**Table 4.** Messages through channels.

Table 4 (a) lists number of EMEM pages ($n_p$) used, number of channels used (nchans), number of total messages ($n_m$) total size of messages, and number of extensions

happened ($n_e$) when running `dedup-d` in either lazy-fix or eager-fix on different number of CPU cores. Column 4 lists $n_p$ used in lazy-fix devided by that in eager-fix, and column 8 lists $n_e$ happened in lazy-fix devided by that in eager-fix. The maximum message size in both two programs is 2.01MB. From the table, we observe that there is about 450MB messages through channels no matter what the core count is. We also find there are some channels transferring short messages of no more than 8 bytes. We extract the counts on this case from the whole counts, listed in Table 4 (b). The amount of short messages of no more than 8-byte occupies more than 25% of total messages. We also find $n_p$ in eager-fix is 4X of thath in lazy-fix, leading to more overhead on EMEM management in eager-fix.

## 6. Related Work

There have been several attempts at providing higher-level of abstraction for expressing pipeline parallelism. Stream programming languages (e.g., StreamIt [5, 13]) enable explicit syntax for data, task and pipeline parallelism, but remain a challenge to compilation and optimization. Task-based libraries such as TBB, Cilk Plus for C/C++[1] and TPL Dataflow in .NET[2] offer attractive alternatives for legacy code. TBB includes specialized pipeline constructs for expressing restricted pipelines [11]. Cilk Plus can only express a *serial-parallel-serial* 3-stage pipeline by clever use of spawn-sync and reducer constructs [8]. TPL Dataflow includes specialized dataflow constructs for general producer/consumer relationships, but only supports .NET platform. They all rely on programmers to avoid race conditions and other errors caused by nondeterminism.

***sumC*** currently does not contain higher-level constructs such as StreamIt's Pipeline, SplitJoin, and FeedbackLoop constructs for composing pipeline stages into a communicating network, but offers multicast channel abstraction for automatic management and optimization of communication, and can be used to greatly simplify tasks of a streaming language compiler or a scheduler.

## 7. Conclusions

We have presented ***sumC***, a scalable unbounded multicast channel, to support efficient pipeline parallelism. Our conversion on existed pipeline Pthreads programming using ***sumC*** indicates the ease-to-use programmability. Our evaluation further shows the scalability and reliablity of ***sumC*** for pipeline parallelism. As to poor performance on multicast across processors, our future work will provide NUMA-aware scalable multicast channels.

[1] http://www.cilkplus.org/

[2] http://www.nuget.org/packages/Microsoft.Tpl.Dataflow

## References


[1] C. Bienia et al. The PARSEC benchmark suite: Characterization and architectural implications. In *17th PACT*, pages 72–81, October 2008.

[2] J. Evans. A scalable concurrent malloc (3) implementation for FreeBSD. In *BSDCan Conference*, Ottawa, Canada, 2006.

[3] B. Ford et al. Microkernels meet recursive virtual machines. In *2nd OSDI*, pages 137–151, 1996.

[4] W. Gloger. Dynamic memory allocator implementations in linux system libraries, May 2006. URL `http://www.malloc.de/en/index.html`.

[5] M. I. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *12th ASPLOS*, pages 151–162, New York, NY, USA, 2006. ACM. ISBN 1-59593-451-0.

[6] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *22nd SPAA*, pages 355–364, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0079-7. . URL `http://doi.acm.org/10.1145/1810479.1810540`.

[7] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *16th PPoPP*, pages 223–234, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0119-0. . URL `http://doi.acm.org/10.1145/1941553.1941585`.

[8] M. McCool, J. Reinders, and A. D. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Elsevier, 2012.

[9] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *15th PODC*, pages 267–275, New York, NY, USA, 1996. ACM. ISBN 0-89791-800-2. . URL `http://doi.acm.org/10.1145/248052.248106`.

[10] A. Morrison and Y. Afek. Fast concurrent queues for x86 processors. In *18th PPoPP*, pages 103–112, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1922-5. . URL `http://doi.acm.org/10.1145/2442516.2442527`.

[11] E. C. Reed, N. Chen, and R. E. Johnson. Expressing pipeline parallelism using TBB constructs: A case study on what works and what doesn't. In *SPLASH '11*, pages 133–138, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-1183-0.

[12] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in MPICH. *HPDC'2005*, 19:49–66, 2005.

[13] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *11th CC*, pages 179–196, London, UK, 2002. Springer-Verlag. ISBN 3-540-43369-4.

[14] P. Tsigas and Y. Zhang. A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In *13th SPAA*, pages 134–143, New York, NY, USA, 2001. ACM. ISBN 1-58113-409-6. . URL `http://doi.acm.org/10.1145/378580.378611`.