Scalable Locality-Conscious Multithreaded Memory Allocation

Scott Schneider
Department of Computer Science
College of William and Mary
scotts@cs.wm.edu

Christos D. Antonopoulos
Department of Computer Science
College of William and Mary
cda@cs.wm.edu

Dimitrios S. Nikolopoulos
Department of Computer Science
College of William and Mary
dsn@cs.wm.edu

Abstract

We present Streamflow, a new multithreaded memory manager designed for low overhead, high-performance memory allocation while transparently favoring locality. Streamflow enables low overhead simultaneous allocation by multiple threads and adapts to sequential allocation at speeds comparable to that of custom sequential allocators. It favors the transparent exploitation of temporal and spatial object access locality, and reduces allocator-induced cache conflicts and false sharing, all using a unified design based on segregated heaps. Streamflow introduces an innovative design which uses only synchronization-free operations in the most common case of local allocations and deallocations, while requiring minimal, non-blocking synchronization in the less common case of remote deallocations. Spatial locality at the cache and page level is favored by eliminating small objects headers, reducing allocator-induced conflicts via contiguous allocation of page blocks in physical memory, reducing allocator-induced false sharing by using segregated heaps and achieving better TLB performance and fewer page faults via the use of superpages. Combining these locality optimizations with the drastic reduction of synchronization and latency overhead allows Streamflow to perform comparably with optimized sequential allocators and outperform—on a shared-memory system with four two-way SMT processors—four state-of-the-art multiprocessor allocators by sizeable margins in our experiments. The allocation-intensive sequential and parallel benchmarks used in our experiments represent a variety of behaviors, including mostly local object allocation-deallocation patterns and producer-consumer allocation-deallocation patterns.

Categories and Subject Descriptors D.4.2 [*Operating Systems*]: Storage Management—Allocation/deallocation strategies; D.3.3 [*Programming Languages*]: Language Constructs and Features—Dynamic storage management; D.4.1 [*Operating Systems*]: Process Management—Concurrency, Deadlocks, Synchronization, Threads; D.1.3 [*Programming Techniques*]: Concurrent Programming

General Terms Algorithms, Management, Performance

Keywords memory management, multithreading, shared memory, synchronization-free, non-blocking

ISMM'06 June 10–11, 2006, Ottawa, Ontario, Canada.

1. Introduction

Efficient dynamic memory allocation is essential for desktop, server and scientific applications [27]. As more of these applications use thread-level parallelism to exploit multiprocessors and emerging processors with multiple cores and threads, scalable multiprocessor memory allocation becomes of paramount importance.

Dynamic memory allocation can negatively affect performance by adding overhead during allocation and deallocation operations, and by exacerbating object access latency due to poor locality. Therefore, effective memory allocators need to be optimized for both low allocation overhead and good object access locality. Scalability and synchronization overhead reduction has been the central consideration in the context of thread-safe memory allocators [3, 18], while locality has been the focal point of the design of sequential memory allocators for more than a decade [11].

Multiprocessor allocators add synchronization overhead on the critical path of all allocations and deallocations. Synchronization is needed because a thread may need to access another thread's heap in order to remotely release an object to the owning thread. Since such operations may be initiated concurrently by multiple threads, synchronization is used to arbitrate thread accesses to the data structures used for managing the heaps. Therefore, local heaps need to be protected with locks or updated atomically with read-modify-write operations such as `cmp&swap`. The vast majority of thread-safe allocators use object headers [3, 9, 15, 18, 25], which facilitate object deallocation in local heaps but pollute the cache in codes that allocate many small objects.

Locality-conscious sequential allocators segregate objects of different sizes to different page blocks allocated from the operating system [7]. Objects are allocated by merely bumping a pointer and no additional information is stored with each object. In general, the allocation order of objects does not necessarily match their access pattern. However, contiguous allocation of small objects works well in practice because eliminating object headers helps avoid fragmentation and cache pollution.

Efficient, thread-safe memory allocators use local heaps to reduce contention between threads. The use of local heaps helps a multiprocessor allocator avoid false sharing, since threads tend to allocate and deallocate most of their objects locally [3]. At a lower level, page block allocation and recycling policies in thread-safe allocators are primarily concerned with fragmentation and blowup, without necessarily accounting for locality [3].

The design space of thread-safe allocators that achieve both good scalability and good data locality merits further investigation. It is natural to consider combining scalable synchronization mechanisms (such as lock-free management of heaps) with locality-conscious object allocation mechanisms (such as segregated heaps with headerless objects). Although the two design considerations of locality and scalability may seem orthogonal and complementary at first glance, combining them in a unified design is not merely an engineering effort. Several problems and trade-off's arise in an at-

tempt to integrate scalable concurrent allocation mechanisms with cache- and page-conscious object allocation mechanisms in a unified design. Addressing these problems is a central contribution of this paper. We show that both memory management overhead and locality exploitation in thread-safe memory allocators can be improved, compared to what is currently offered by state-of-the-art multiprocessor allocators. These design improvements and the associated performance benefits are also a key contribution of this paper.

We present Streamflow, a thread-safe allocator designed for both scalability and locality. Streamflow's design is a direct result of eliminating synchronization operations in the common case, while at the same time avoiding the memory blowup when strictly thread-local heaps are used in codes with producer-consumer allocation-freeing patterns. Local operations in Streamflow are *synchronization-free*. Not only do these operations proceed without thread contention due to locking shared data, but they also proceed *without the latency imposed by uncontested locks and atomic instructions.* The synchronization-free design of local heaps enables Streamflow to exploit established sequential allocation optimizations which are critical for locality, such as eliminating object headers for small objects and using bump-pointer allocation in page blocks comprising thread-local heaps.

Streamflow also includes an innovative remote object deallocation mechanism. Remote deallocations – namely deallocations of objects from threads different than the ones that initially allocated them – are decoupled from local allocations and deallocations by forwarding remotely freed objects to per-thread, non-blocking, lock-free lists. Streamflow's remote deallocation mechanism enables lazy object reclamation from the owning thread. As a result, most allocation and deallocation operations proceed without the cost of atomic instructions, and the infrequent operations that do require atomic instructions are non-blocking, lock-free and provably fast under various producer-consumer object allocation-freeing patterns.

Streamflow's design favors locality at multiple levels. Beyond reducing memory management overhead and latency, decoupling local and remote operations promotes temporal locality by allowing threads to favor locally recycled objects in their private heaps. The use of thread-local heaps reduces allocator-induced false sharing. Removing object headers improves spatial locality within cache lines and page blocks. The integration with a lower level custom page manager which utilizes superpages [19, 20] avoids allocator-induced cache conflicts via contiguous allocation of page blocks in physical memory, and reduces the activity of the OS page manager, the number of page faults and the rate of TLB misses. Combining these techniques produces a memory allocator that consistently outperforms other multithreaded allocators in experiments with up to 8 threads on a 4-processor system with Hyperthreaded Xeon processors. Streamflow, by design, also adapts well to sequential codes and performs competitively with optimized sequential and application-specific allocators.

This paper makes the following contributions:

- We present a new thread-safe dynamic memory manager which bridges the design space between allocators focused on locality and allocators focused on scalability. To our knowledge, this is the first time a memory allocator efficiently unifies locality considerations with multiprocessor scalability.
- We present a new method for eliminating (in the common case) and minimizing (in the uncommon case) synchronization overhead in multiprocessor memory allocators. Our method decouples remote and local free lists and uses a new non-blocking remote object deallocation mechanism. This technique preserves the desirable properties of a multiprocessor memory allocator, namely blowup avoidance and false sharing avoidance, without sacrificing the locality and low latency benefits of bump-pointer allocation.
- We present memory allocation and deallocation schemes that take into account cache-conscious layout of heaps, page- and TLB-locality. To our knowledge, Streamflow is the first multiprocessor allocator designed with multilevel and multigrain locality considerations.
- We demonstrate the performance advantages of our design using realistic sequential and multithreaded applications as well as synthesized benchmarks. Streamflow outperforms four widely used, state-of-the-art multiprocessor allocators in allocation-intensive parallel applications. It also performs comparably to optimized sequential allocators in allocation-intensive sequential applications. Streamflow exhibits solid performance improvements both in codes with mostly local object allocation-freeing patterns and codes with producer-consumer object allocation-freeing patterns. We have experimented with an SMP with four two-way SMT processors[1]. Such SMPs are popular as commercial server platforms, affordable high-performance computing platforms for scientific problems, and building blocks for large-scale supercomputers. Since Streamflow eliminates (in the common case) or significantly reduces (in the uncommon case) synchronization, the key scalability-limiting factor of multithreaded memory managers, we expect it to be scalable and efficient on larger shared-memory multiprocessors as well.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the major design principles, mechanisms and policies of Streamflow. Section 4 presents our experimental evaluation of Streamflow alongside other multiprocessor allocators and some optimized sequential allocators. In Section 5 we discuss some implications of the design of Streamflow and potential future improvements. Section 6 summarizes the paper.

2. Related Work

Streamflow includes elements adopted from efficient sequential memory allocators proposed in the past. Streamflow's segregated heap storage and BIBOP (big bag of pages)-style allocation derives from an allocation scheme originally proposed by Guy Steele in [24] and from the concept of independently managed memory zones which dates back to 1967 [21]. Segregated heap storage has since been used in numerous allocators, including the standard GNU C allocator in Linux [16], an older GNU allocator [11], Vmalloc [26], and more recent allocators such as Reaps [4] and Vam [7]. Modern allocators tend to adopt segregated heaps because they enable very fast allocation. Deallocation in segregated heap allocators is more intricate, because in order to comply with the semantics of `free()`, the allocator needs to be able to discover internally the size of each deallocated object, using the object pointer as its only input. Deallocation is simple if each object has a header pointing to the base of the heap segment from where the object was allocated. This technique is used, for example, in the GNU C allocator and in Reaps [4, 16]. However, object headers introduce fragmentation, pollute caches, and eventually penalize codes with many small object allocations. Therefore, locality-conscious allocators such as PHKmalloc [12] and Vam [7] eliminate object headers entirely for small objects and use tables of free lists to manage released space in segregated heaps. Elimination of headers is common practice in custom memory allocators [4], as well as semi-custom allocators

[1] This is the largest shared-memory system we have direct access to.

with alternate semantics for `free()`, such as region-based allocators [8].

Streamflow uses segregated object allocation in thread-private heaps, as in several other thread-safe allocators including Hoard [3], Maged Michael's lock-free memory allocator [18], Tcmalloc from Google's performance tools [10], LKmalloc [15], ptmalloc [9], and Vee and Hsu's allocator [25]. In particular, Streamflow uses strictly thread-local object allocation, both thread-local and remote deallocation and mechanisms for recycling free page blocks to avoid false sharing and memory blowup [3, 18].

Streamflow differs from earlier multithreaded memory allocators in several critical aspects: First, its design decouples local from remote object deallocation to allow local allocation and deallocation without any atomic instructions. Atomic instructions are used only sparingly for remote object deallocation and for recycling page blocks. Second, Streamflow eliminates object headers for small objects, thereby reducing cache pollution and improving spatial locality. Tcmalloc is the only thread-safe allocator we are aware of that uses the same technique, although Tcmalloc uses locks whenever memory has to be allocated from or returned to a global free memory objects pool. Third, Streamflow uses further optimizations for temporal locality, cache-conscious page block layout and better TLB performance. Fourth, unlike many other high performance allocators, Streamflow allows returning memory to the OS when the footprint of the application shrinks.

To our knowledge, Streamflow is the first user-level memory allocator to control the layout of page blocks in physical memory, using superpages as the means to achieve contiguous allocation of each page block in physical memory. It should be noted that superpages are a generic optimization tool and their scope extends beyond just memory allocators [6, 19]. However, since superpages (the size of which is set by the operating system) may subsume multiple page blocks (the size of which is set by the memory allocator) a multiprocessor memory allocator using superpages to achieve cache-conscious of page blocks has certain design choices as to how it manages free memory inside each superpage and how it divides superpages between page blocks from different threads. Streamflow's design includes some educated choices for effective management and utilization of superpages.

Several of the design goals of Streamflow, in particular its locality optimizations, can be achieved with allocators that utilize feedback from program profiles. For example, earlier work has shown that object lifetime predictors and reference traces can be used to customize small object allocation and object segregation [2, 22, 23]. Streamflow assumes no knowledge of object allocation and access profiles, although its design does not prevent the addition of profile-guided optimization.

3. Design of Streamflow

Streamflow primarily optimizes dynamic allocation of small objects, which is a common bottleneck in many sequential and multithreaded applications, including desktop, server and scientific applications. Streamflow optimizes dynamic allocation for low latency and scalability, as well as for temporal locality, spatial locality and cache-conscious layout of data. These optimizations are accomplished via the use of a decoupled local heap architecture, the elimination of object headers, the careful layout of heaps in contiguously allocated physical memory and the exploitation of large pages (superpages). At the same time, Streamflow provides mechanisms that facilitate both memory transfer between local heaps and returning memory to the system. As a result, it is not sensitive to pathologic memory usage patterns, such as producer-consumer ones, that could lead to high memory overhead and pressure.

Streamflow consists of two modules. Its front-end is a multithreaded memory allocator, which minimizes the overhead of memory requests by eliminating inter-thread synchronization and all associated atomic operations during common-case memory request patterns. Even in the infrequent cases when synchronization between threads is necessary, it is performed with a single, non-blocking atomic operation[2]. The front end also includes optimizations for spatial locality, temporal locality, and the avoidance of false-sharing.

The back-end of Streamflow is a locality-conscious page manager. This module manages contiguous page blocks, each of which is used by the front-end for the allocation of objects that belong to a given size class. The page manager allocates pages blocks within superpages to achieve contiguous layout of each page block in physical memory, thus reducing self-interference (within page blocks) and cross-interference (between page blocks) in the cache. The use of superpages can also improve the TLB performance and reduce page faults in applications with large memory footprints. Moreover, the Streamflow back-end facilitates the interchange of page blocks between threads, should the memory demand of each thread change during execution.

We describe the front-end multithreaded memory allocator in Section 3.1 and the back-end page manager in Section 3.2. The source code of Streamflow can be downloaded from `http://www.cs.wm.edu/streamflow` and can be used as a reference throughout this section.

3.1 Multithreaded Memory Allocator

3.1.1 Small Object Management

Objects in Streamflow are classified as small if their size does not exceed 2 KB (half a page in our experimental platform). The management of objects larger than 2 KB is described in section 3.1.2. In the following paragraphs we describe Streamflow's heap architecture, the techniques used to eliminate object headers, small object allocation and deallocation procedures and specialized support for recycling memory upon thread termination.

Local heaps: Each thread in Streamflow allocates memory from a local heap. The heap data structure, shown in Figure 1(a), is strictly private; only the owner thread can modify it. As a result, the vast majority of simultaneous memory management operations issued by multiple threads can be served simultaneously and independently, without synchronization. Synchronization is necessary only when the local heap does not have enough free memory available to fulfill a request, or during deallocations, when an object is freed by a thread other than the owner of the heap it was allocated from.

Local heaps facilitate the reduction of allocator-induced false-sharing between threads, since memory allocation requests by different threads are not interleaved in the same memory segment. This technique cannot, however, totally eliminate false-sharing in the presence of object migrations between threads [3].

Each thread-local heap consists of *page blocks*, shown in Figure 1(b). Page blocks are contiguous virtual memory areas. Each page block is used for the allocation of objects with sizes that fall into a specific range, which we call an *object class*. In Streamflow, each object class differs from the previous one by 4 bytes. This design provides for fine-grain object segregation and tends to improve spatial locality in codes that make heavy use of very small objects [7]. One or more page blocks, organized as a doubly linked list, can

[2] We use `cmp&swap(ptr, old_val, new_val)`, which atomically checks that the value in memory address `ptr` is `old_val` and changes it to `new_val`. If the value is not equal to `old_val` the operation fails. The operation may be replayed more than once if it fails. All modern processors offer `cmp&swap` for 32-bit and 64-bit operands, either as an instruction or as a high level primitive built from simpler instructions, such as load linked-store conditional.

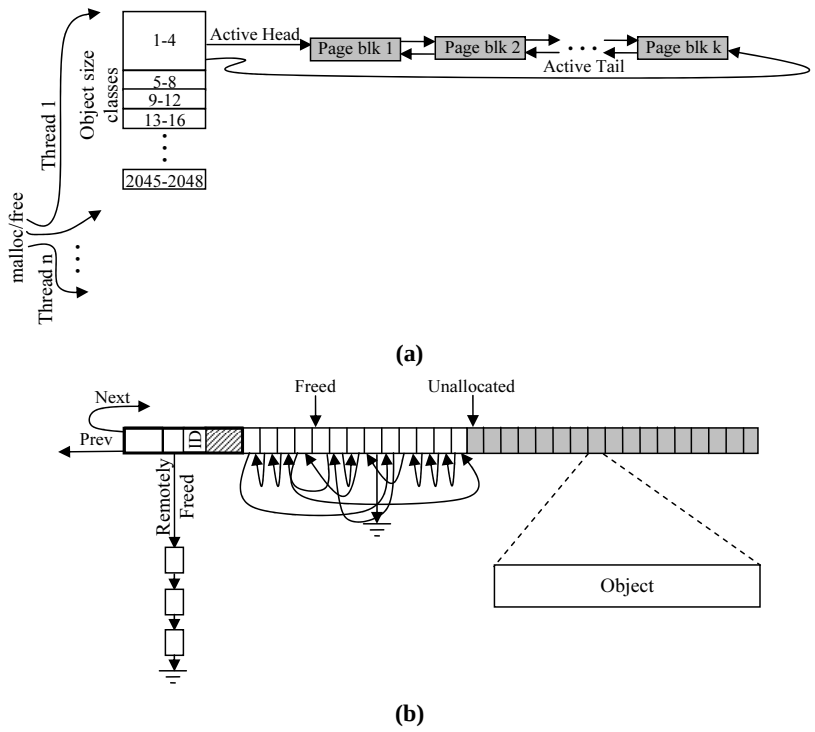

Figure 1. Streamflow front-end design. Figure (a) is an overview of a heap, and Figure (b) is the detail for a particular page block within that heap.

serve the same object class. A simple page block rotation strategy guarantees that if there is enough free memory for the allocation of a specific object class, a page block with available memory will be found at the head of the list. More specifically, when a page block becomes full, it is transferred to the end of the list. Similarly, when an object is freed by the owner of the heap, the page block it belongs to is placed at the head of the list, if it is not already there. The block rotation is a fast operation involving exactly seven pointer updates and no atomic instructions.

Page blocks are always page aligned. Their sizes vary, depending on the object class they serve. As a rule of thumb, each page block in Streamflow is large enough to accommodate 1024 objects, however minimum/maximum page block size limitations also apply. There is clearly a trade-off between the number of objects in each page block and the average amount of unused memory a page block may contain. The minimum page block size (16 KB in Streamflow) allows more than 1024 very small objects to be packed inside a single page block, given that the size of the resulting page blocks is also small and the additional memory consumption is not a concern. This amortizes costly heap expansion operations among more object allocations. On the other hand, the maximum page block size (256 KB in our implementation) limits the memory requirements for page blocks which serve relatively large object classes. Without a limit, page blocks for large objects could otherwise grow up to 2 MB. This limit reduces internal allocator fragmentation, which is the amount of memory reserved from the system, yet never used inside each page block. The resulting page block size is always rounded to the nearest power of two.

The beginning of each page block is occupied by the page block header. The header consists of all the data structures and bookkeeping information necessary for the management of the page block. It contains: i) Pointers for linking the page block to the doubly-linked list of page blocks for each object class, ii) Pointers to free memory inside the page block (`freed` and `unallocated`), iii) An identifier of the owner-thread of the page block (`id`), iv) The head of a LIFO list used for object deallocations to the page block by threads other than the owner-thread (`remotely_freed`), and v) bookkeeping information, such as the number of free objects in the page block and the size of each object. All the fields in the header, with the exception of `remotely_freed`, are accessed only by the page block owner-thread, thus accesses and modifications of these fields require no synchronization.

Headerless objects: When an object is freed, a memory allocator needs to discover whether the object is large or small as well as its size and—if the object is small—the exact page block it originated from. A common technique is to attach a header to each object and encode the necessary information in the header. Some architectures impose an 8-bytes alignment requirement for certain data types, or accesses to these data types suffer significant performance penalties. This limits the minimum memory required for headers to 8 bytes and the minimum object granularity supported by the allocator to 16 bytes (including the header). As a result, the use of headers introduces two serious side-effects: a) Significant space overhead, which can reach up to 300% (12 bytes of overhead for every 4-bytes object), and b) less objects can be packed in a single cache line or a single page, thus hurting spatial locality.

Streamflow eliminates headers from small objects using the BIBOP technique [24]. We introduce a global table with one byte for each virtual memory page in the system. Accesses to the table can simply be indexed by the page starting address. A single bit of each table cell characterizes objects allocated in the specific page as small or large. If the object is small, the remaining 7 bits are used to encode the disposition—in pages—of the header of the parent page block. This encoding is sufficient for realistic page block sizes (up to 512 KB, considering a page size of 4 KB). As soon as the header of the parent page block is available, the memory manager has all

the necessary information to proceed with the deallocation. In a 32-bit address space 1 MB is enough for the BIBOP table (768 KB in Linux, since 25% of the virtual address space is reserved for kernel memory). In 64-bit address spaces, multilevel trees or tries can be used instead [10], to encode information only for the segments of the address space that are actually used by the application. We are currently investigating these options in an ongoing effort to port Streamflow to a 64-bit system. The BIBOP technique allows the elimination of headers for small objects without introducing artificial segmentation of the virtual address space. The elimination of headers allows Streamflow to better exploit spatial locality opportunities. It also facilitates the support of arbitrarily small objects. In the current implementation the minimum object granularity is 4 bytes.

Object allocation: When a memory request is received, Streamflow directs it to the appropriate object size class in the local heap of the thread that initiated the request. In the common case, the first page block in the list of that size class will have available objects. There are two categories of available objects. Those that have already gone through one or more allocation/deallocation cycles populate the `freed` LIFO list and are preferred for consequent allocations. This design decision, combined with the LIFO organization of the list, favors temporal locality, since recently deallocated objects are reused as soon as possible. If the `freed` list is empty, Streamflow allocates one of the objects of the page block that have never been allocated before. The beginning of the memory area that accommodates such objects is pointed to by `unallocated`, which is a bump-pointer that is forwarded each time by one object.

Object deallocation: Object deallocations are usually initiated by the same thread that allocated the object. If this is the case, the object is simply inserted—without any synchronization—into the `freed` LIFO of the parent page block that it originated from. If after the deallocation the page block becomes empty, it is dealt with by the page block caching policy which is described later on.

Remote object deallocations are deallocations of an object from a thread other than the one that allocated it. Remote deallocations need to be treated differently, since only the owner-thread of each page block can modify the `freed` LIFO. In this case, the object is inserted to the `remotely_freed` LIFO list of the parent page block. The insertion to the list is performed via a 64-bit atomic `cmp&swap` operation which simultaneously updates the `remotely_freed` LIFO head and checks the owner identifier (`id`) of the parent page block to ensure that the page block is actually owned by a thread[3]. Objects inserted into the `remotely_freed` LIFO will be eventually transferred to the `freed` LIFO by the owner-thread of the page block. The decoupling of local and remote operations is a key design point which drastically improves the latency and scalability of Streamflow by eliminating atomic instructions from the critical path of the most frequent operations. Furthermore, Streamflow uses the minimum number of atomic instructions for thread-safe remote object deallocations.

When a memory request can not be served by the page block at the head of the appropriate object size class because the page block is full, the owner-thread checks the `remotely_freed` LIFO for objects freed earlier to the page block by remote threads. If such objects exist, they are all removed with a single atomic `cmp&swap` operation and transferred to the `freed` list. The memory request then proceeds exactly as the common case memory request described earlier. The lazy reclamation policy of remotely freed objects, combined with the page blocks rotation strategy, guarantees that remotely freed memory objects will eventually be reused. However, their reuse will be delayed until it is absolutely necessary: when the parent page block runs out of free memory. This strategy minimizes the number of atomic operations required for accessing the `remotely_freed` list. If, however, the page block at the head of the object size class is full and its `remotely_freed` list does not contain any objects, the page block is rotated to the end of the list and a new page block is fetched from the cache or requested from the page manager.

Thread termination: Whenever a thread terminates, Streamflow ensures that the free memory of partially free or locally cached page blocks in its heap will be made available to the other threads. Empty and partially full page blocks are handled by the caching policy described below. If one of the page blocks appears to be full, its `remotely_freed` list is checked for remotely freed objects. If the list is not empty, the objects are removed—with a single, atomic `cmp&swap` operation—and transferred to the `freed` list. Following, the page block is managed by the caching policy as a completely or partially free block. If this is not the case, the thread declares the page block as "orphaned," by setting the `id` of its owner to `NULL`. Any orphaned page block can be "adopted" and attached to the heap of the first thread that deallocates an object originating from it observes that the page block is orphaned.

The `id` is set to `NULL` with an atomic 64-bit `cmp&swap` operation, which simultaneously verifies that the `remotely_freed` list remains empty. Should the instruction fail, one or more objects have been freed into the `remotely_freed` LIFO after the last check, so the page block is no longer full. The atomic operation eliminates the possibility of declaring a page block as orphaned after all its objects have been returned to the `remotely_freed` list. The free memory of such a page block would never be reused, since no thread would ever have the opportunity to observe it as orphaned.

Page block caching: Page block caching is the boundary that separates the multithreaded memory allocator front-end and the page manager back-end. When the allocator needs a new page block, it first checks a thread-local cache, then the global caches, and if no cached page blocks of the correct size are found, it passes a request on to the page manager. The local caches are synchronization-free LIFO lists, and the global cache is a lock-free LIFO list. The caching layer is the last level at which Streamflow applies lock-free, non-blocking synchronization. Its purpose is to relieve strain on the page manager.

Page blocks in local caches are organized according to their size. Due to the minimum size, maximum size, and power of two size limitations for page blocks, multiple object classes use page blocks of the same size. Orphaned page blocks whose original owner thread has terminated are placed on a global list, which must preserve the page block's object class, since there are still some live objects allocated from this page block. Completely free page blocks can be placed on a global free cache upon thread termination, or when a thread releases a page block and the local cache is overpopulated. In order to maintain low virtual memory usage, our implementation constrains the population of the local and global caches to one and zero page blocks respectively. Orphaned page blocks can always be stored in the global list of orphaned blocks, independent of the list's population.

Discussion: From the discussion so far it is clear that Streamflow performs the vast majority of memory allocation/deallocation operations without introducing synchronization. Synchronization between threads is only required in the infrequent cases of: i) remote object deallocations, ii) batch reclamation of remotely freed objects, iii) declaration of a page block as orphaned, iv) adoption of an orphaned page block, and v) page block returns to or requests from the page manager. Even in these cases, with the exception of

[3] Given that `id` and `remotely_freed` need to be updated by a single 64-bit atomic operation, they are always placed into 64 consecutive bits in the page block header.

(v), the synchronization is performed using a single non-blocking atomic operation (`cmp&swap`).

3.1.2 Large Object Management

The management of large objects is significantly simpler than that of small objects. Large object requests are forwarded directly to the operating system. After memory is allocated from the system, the BIBOP table is updated to indicate that the corresponding virtual pages accommodate a large object. Finally, the object is prefixed with a header that contains the object size and the object is returned to the application.

Similarly, if the BIBOP table lookup during a deallocation identifies an object as large, its header is recovered from the 8 bytes right before the object's base address. As soon as the object size is determined, the memory occupied by the object is returned to the operating system.

3.2 Page Manager

The page manager implements page block allocations and deallocations as needed by the multithreaded memory allocator in the front-end of Streamflow. Its functionality is threefold: i) It allocates and deallocates physical memory from/to the operating system, using superpages as the unit of allocation. ii) It allocates page blocks and manages space within superpages to achieve contiguous allocation of each page block in physical memory. iii) It optimizes the placement of multiple page blocks within superpages to avoid cache conflicts within and between page blocks residing in the same superpage.

Most modern processors provide support for multiple page sizes. For example, Intel's Itanium 2 provides eleven page sizes between 4 KB and 4 GB, Alpha processors provide four page sizes between 8 KB and 4MB, while the IBM Power4/Power5, Intel Xeon and UltraSPARC processors provide two page sizes, a small page size of 4 or 8 KB and a large page size of 4 MB. We use the term superpages to refer to pages of size larger than the smallest page size on a given architecture.

Superpages enable the coverage of large regions of the virtual address space with a small number of pages and TLB entries. Therefore, they can improve performance by reducing paging and TLB misses. The use of superpages can be particularly beneficial on simultaneous multithreading (SMT) processors, where more than one threads share a common TLB and the typically few TLB entries become a contested resource. More importantly, superpages enable contiguous allocation of large regions of the virtual address space in physical memory. Contiguous allocation of large blocks of virtual memory often improves cache performance on processors with large, physically indexed L2 caches, by eliminating or reducing interference in the cache within and between page blocks.

Streamflow's page manager is implemented on top of Linux 2.6, which provides support for superpages via a virtual filesystem. Streamflow allocates superpages by creating files in the virtual filesystem, and mapping these files in whole or in part to virtual memory. An allocated superpage is uniquely identified by the virtual file which backs the page and its disposition within this file.

The page manager associates a "header" data structure with each superpage. The collection of superpage headers holds all necessary information for the management of different superpages, as well as for management of space inside each superpage. Superpage headers reside in page blocks which are dynamically allocated from the operating system. The management of page blocks that store superpage headers is almost identical to the management of page blocks used for small objects (described in section 3.1.1). The main difference is that, since a page block with superpage headers is global and accesses to it are protected by a global page manager lock, functionality related to remotely freed objects, orphaned page blocks and page block adoption is not necessary. Moreover, page blocks with superpage headers do not need to be freed, since the space they occupy is negligible. As a result, the data structures used for the management of page blocks with superpage headers do not need to be replicated with each page block.

Each superpage header includes the disposition of the superpage in the virtual file which backs the superpage. The page manager returns superpages to the operating system when all page blocks within a superpage are freed. Whenever a superpage is returned to the operating system (via `munmap()`), its header is recycled to the `freed` LIFO list of superpage headers, however the disposition of the superpage is preserved in the header. As a result, the page manager can easily identify the dispositions of unmapped superpages inside a file, just by reusing recycled headers. The superpage headers also include `prev` and `next` pointers for linking superpages in lists, a pointer to the base virtual address of the superpage (`sp_base`), the size (as a power of 2) of the largest contiguous free memory block inside the superpage (`largest_free_order`), as well as some bitmaps necessary for managing space inside the superpage.

Allocated superpages are organized in a hash table, indexed with the size (as a power of 2) of the requested page block. Using this hash table, the page manager can easily search for "*best-fit*" superpages, namely superpages where the largest contiguous free block is as close as possible to the size of the requested block.

Streamflow's page manager allocates memory within each superpage using a buddy allocator [13, 14]. The buddy allocator tends to reduce memory fragmentation inside each superpage, being at the same time faster than first-, next-, and best-fit allocators.

4. Evaluation

4.1 Experimental Setting

We evaluated Streamflow on a 4-processor Dell PowerEdge 6650 server, with Hyperthreaded Intel Xeon processors clocked at 2.0 GHz. Hyperthreaded Intel processors can execute up to 2 threads simultaneously. Each processor has a 4-way associative 8 KB L1 data cache, a 12 KB instruction trace cache, a 512 KB 8-way associative L2 cache and an external 1 MB L3 cache. The system has 2 GB of RAM and runs Suse Linux 9.1 with the 2.6.13.4 kernel and glibc 2-3.3.

To compare the performance of Streamflow against other multithreaded memory allocators, we evaluated the performance of Hoard (version 3.3.0) [3], Tcmalloc from Google's performance tools (version 0.4) [10], our 32-bit implementation of Maged Michael's lock-free allocator[18] and the thread-safe allocator of glibc in Linux, which is based on Doug Lea's memory allocator [16] with extensions for thread safety implemented by Wolfram Gloger [9]. Hoard, Tcmalloc and glibc use local heaps with multiple object size classes. Hoard, Tcmalloc and Michael's allocator use a minimum object granularity of 8 bytes. The glibc allocator uses a minimum object granularity of 16 bytes. Tcmalloc is the only multithreaded allocator besides Streamflow that does not use object headers for small objects. We also compared the performance of Streamflow for a sequential application with that of glibc and Vam [7]. Interestingly enough, the glibc allocator uses a more efficient non-thread-safe implementation of `malloc()` and `free()` if it detects that the code is not multithreaded. Streamflow, on the other hand, adapts to sequential codes as a consequence of its design, which completely offloads synchronization from the critical path of sequential allocation. Vam is an optimized, strictly sequential memory allocator that targets the improvement of application locality at both the cache- and the page-level. Vam uses fine-grain object size classes, headerless objects and reap style allocation.

Benchmark	Description	Input	#Objects ($\geq$ 2KB)	#Objects ($<$ 2KB)	Avg. object size ($<$ 2KB)	Remote Frees (%)
197.parser	SPECINT2000 English parser (sequential)	reference	79760	787M	14b	N/A
Recycle	Synthetic benchmark, with variable object recycling frequency	10^7 objs./thread 8b objects, *rate*=1000	8	10^7/thread	8b	0.0
Larson	Multithreaded server simulator	5-seconds run	0	2.72M	7b	100
Consume	Producer-consumer benchmark	6000 objs/block 5000 iterations	8	240M	4b	100
Knary	Hood implementation of a parallel tree building benchmark	(11,5,0,0)	16	61M	40b	0.00056
Barnes	Hood implementation of the Barnes-Hut algorithm	131072 bodies d=0.025, 10 its.	17	2.33M	30b	0.79
MPCDM	Multithreaded mesh generation application	10M triangles	338	30M	35b	1.4

Table 1. Benchmarks used to evaluate Streamflow.

Table 1 summarizes the benchmarks we used to evaluate Streamflow. The table includes the total number of small objects (smaller than 2KB) allocated in each benchmark, the number of large objects, the average size of small objects, and the percentage of remotely freed objects in the multithreaded benchmarks. It is evident that the vast majority of objects are deallocated by the same thread that allocated them, even in codes in which objects are accessed by multiple threads. The only exceptions are the *Consume* and *Larson* benchmarks which are explicitly designed to stress the allocators under extreme remote memory deallocation and reclamation conditions. This property puts Streamflow at an advantage against other allocators, thanks to its design that removes the synchronization overhead from almost all allocations and deallocations.

The results of our experiments are summarized in Figures 2 and 3. We report execution times for *197.parser*, *Recycle*, *Consume*, *Knary*, *Barnes* and *MPCDM*. *Larson* runs for a fixed time interval, so we report its throughput, measured in memory management operations per microsecond. For the sequential *197.parser* we report the execution time of both the optimized non-thread-safe, and the thread-safe glibc allocator implementation. For *197.parser*, we also report execution times for the custom, hard-coded allocator of the benchmark and Vam. Note that we bypass the custom allocator of the benchmark when we measure the performance of Streamflow and the other general-purpose allocators. This means that all allocations and deallocations are directed to the `malloc()` and `free()` calls of the general-purpose allocators rather to the `xalloc()` and `xfree()` calls of the custom allocator of the benchmark. For all benchmarks we also report the performance of the four thread-safe allocators (Streamflow, Michael's, Hoard and Tcmalloc). Specifically for Streamflow, we provide three data points: one for a base implementation which performs decoupling of local allocation from remote deallocation but no further locality optimizations (labeled "Streamflow headers" in the charts); a second from an improved implementation which eliminates object headers, in addition to decoupling, to improve spatial locality and cache usage (labeled "Streamflow wo headers" in the charts); and a third from a complete implementation which includes the superpage manager and page block layout optimizations (labeled "Streamflow superpages" in the charts).

4.2 Results and Analysis

4.2.1 Sequential Benchmarks

197.parser: This benchmark is an English language parser benchmark from SPECINT2000 which includes a custom memory allocator. This custom allocator works well for objects allocated and deallocated in a stack-like fashion. *197.parser* spends more than 40% of its execution time in memory allocation and stresses the efficiency of object placement in memory as well as the ability of thread-safe allocators to provide fast sequential allocation. Most thread-safe allocators suffer in this aspect because they impose synchronization overhead, due to unnecessary execution of atomic instructions, even though there is no contention between threads in a sequential program. *197.parser* is also representative of many applications that use custom allocators for higher performance [4, 7].

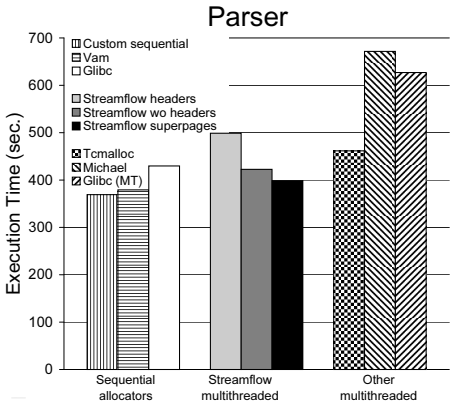

Figure 2. Execution time (lower is better) attained by different allocators for *197.parser*.

The performance of Streamflow with all locality optimizations is within 8% of the performance of the custom allocator in *197.parser* and within 5% of the performance of Vam. Both the custom allocator and Vam apply bump-pointer memory allocation. Moreover, the custom allocator changes the semantics of `free()` to provide the memory object size to the deallocation function. Streamflow performs 7% faster than the optimized, sequential glibc allocator.

Streamflow outperforms the thread-safe glibc allocator by 36% and Michael's allocator by 40%, due to the fact that the latter two suffer the unnecessarily high—although contention free—overhead of atomic instructions. *197.parser* crashes when executed with Hoard, and as a result a comparison with Hoard is not possible.

The elimination of object headers results in a 15% performance improvement over the base implementation of Streamflow. Headerless objects contribute to better spatial locality at both the cache- and page-level. Minor page faults, for example, drop from 3.8M to 2.6M. Streamflow without headers is 8.5% faster than Tcmalloc, which also uses headerless objects. The cost for a common-case, uncontested memory allocation is 200 cycles for Streamflow, compared with 214 cycles for Tcmalloc.

The use of superpages yields an additional 5% execution time improvement over the base Streamflow implementation. It also almost completely eliminates minor page faults (just 713, down

from 2.6M) and reduces TLB misses by 2%. The larger page size allows the coverage of the address space with significantly fewer pages and, thus, reduces the pressure to the OS page manager and the TLB.

4.2.2 Multithreaded Benchmarks

Recycle: This is a custom synthetic microbenchmark that stresses the ability of multithreaded allocators to efficiently perform simultaneous memory management operations by multiple threads, for objects that are created and destroyed locally by each thread. Each thread allocates a total of 10^7 objects, the size of which is selected randomly from a given range. The benchmark can simulate different memory reuse patterns. Each thread deallocates all the objects it has allocated after every *rate* allocations, *rate* being a user-provided parameter. *Recycle* is not expected to scale with more processors (in terms of execution time reduction), since its workload is also scaled with the number of threads.

Streamflow outperforms glibc by 41%, Hoard by 59%, Michael's lock-free allocator by 48% and Tcmalloc by 14% in the sequential execution of *Recycle*. In multithreaded executions Streamflow performs significantly better than allocators that employ synchronization, such as glibc (8%–39%, avg. 25%), Hoard (11%–54%, avg. 29%) and Michael's allocator (10%–43%, avg. 22%). Its performance is similar to Tcmalloc, which uses thread-local, synchronization-free memory caches.

Allocators that put synchronization on the critical path of every operation—even Michael's allocator, which uses lock-free, non-blocking synchronization—suffer from synchronization latency even during thread-local management operations, although these operations could be performed independently by each thread. Streamflow performs thread-local memory allocation and deallocation without atomic instructions. *Recycle* is only sensitive to allocation latency and scalability. Since allocated objects are not accessed in the code, and the memory footprint of the benchmark is rather small, it is insensitive to locality optimizations. As a result, the performance of the three versions of Streamflow is practically indistinguishable.

Larson: This is a benchmark which simulates a multithreaded server [15]. Objects allocated from a given thread are released by another thread. The thread that performs the deallocation is usually spawned after the termination of the thread that performed the allocation. We experimentally found that this is the case in 96.8% of all deallocations. *Larson*, thus, stresses the page block adoption functionality of Streamflow. The allocating thread is still alive in 3.2% of the deallocations, which activates the remote memory object deallocation and reclamation modules of Streamflow. The code runs for a fixed time interval and reports the attained throughput in terms of memory management operations. The number of threads spawned is proportional to the speed of memory allocation and deallocation[4]. *Larson* is sensitive to multithreaded allocation latency and scalability, as well as to the performance of the memory recycling mechanisms of the allocator upon thread termination.

Streamflow, with all locality optimizations enabled, outperforms Hoard by 56% on one thread and by more than 5.5 times on 8 threads (2.5 times on avgerage). The improvements over glibc range from 27% to 7.73 times (4.53 times on avg.). Streamflow achieves 1.9 to 2.6 times higher throughput than Tcmalloc (67% on avg.). Tcmalloc outperforms Streamflow only in one case, when *Larson* is executed sequentially. Finally, Streamflow proves 78% to 5.6 times (3.5 times on avg.) more efficient than Michael's lock-free allocator. Note that Streamflow scales almost linearly with the number of threads in *Larson*, a particularly desirable property for real-world multithreaded server applications. The main reason for Streamflow performance is the efficient page block adoption strategy upon thread termination. A page block is adopted—with a non-blocking atomic operation—by the first thread that deallocates an object originating from it. Further deallocations to the page block from that thread are treated as local ones. Thus, they do not suffer even the minimal overhead of lock-free enqueueing to the `remotely_freed` queue of the parent page block.

Objects in *Larson* are merely allocated and deallocated, being accessed only once, immediately after their allocation. However, the elimination of object headers enhances the spatial locality, especially at the page-level. Minor page faults are reduced (for the 8 threads execution) from 51K to 3.2K, resulting in a throughput improvement of 15%.

Consume: This is a synthetic microbenchmark from the Hoard distribution. It simulates produced-consumer applications, in which memory objects are allocated from one thread and are used and deallocated by other threads. The producers and consumers live simultaneously in the system. A single producer thread allocates n blocks of memory, each of which is then deallocated by one of the n different consumer threads. Memory allocations for a block can be performed simultaneously with deallocation of memory objects from other blocks. The number of threads, the size of the blocks and the number of allocation/deallocation rounds are specified by the user. *Consume* stresses the efficiency of remotely freeing memory through non-blocking atomic operations and the efficiency of lazy memory reclamation in Streamflow. The single producer thread is the main performance bottleneck of the application. As a result, the execution time of *Consume* is expected to increase almost linearly with the number of consumer threads. Moreover, since memory objects are simply allocated and deallocated, locality optimizations cannot be expected to have any effect.

Streamflow performs 25% to 1.3 times (avg. 77%) better than glibc and 60% to 8.7 times (avg. 5.1 times) better than Hoard. It also outperforms Michael's lock-free allocator by 74% to 2.7 times (avg. 1.8 times) and Tcmalloc by 4% to 3.7 times (avg. 2 times).

Multithreaded allocators based on locks, such as glibc and Hoard must acquire and release at least one lock per deallocation operation. Tcmalloc and Michael's allocator minimize the effects of producer-consumer memory usage patterns by using thread-local caches and atomic, lock-free operations respectively. In the case of Streamflow, each remote memory object deallocation is performed at the cost of a single, non-blocking, atomic operation. Moreover, the lazy memory reclamation strategy amortizes the cost of reclaiming the freed memory to that of a single atomic operation for all the objects in the `remotely_freed` queue of the page block.

Knary: Hood benchmark which builds trees of arbitrary depth and fan-out degree and associates a user-defined amount of work per tree node [1]. *Knary* is sensitive to allocation latency and scalability, but not to locality, because the work performed per generated tree node is typically small. It stresses the ability of allocators to serve simultaneous, mostly thread-local memory allocation and deallocation requests by multiple threads.

Streamflow outperforms glibc by 70% to 1.5 times (avg. 88%). It also provides significant performance improvements over Hoard, Tcmalloc and Michael's allocator (64%, 73% and 76% on average respectively). Tcmalloc performs similarly with Streamflow only when *Knary* is executed sequentially.

The performance improvements can be attributed directly to the design of Streamflow, which minimizes—and in most cases eliminates— synchronization between threads performing simultaneous memory management operations.

[4] We set *Larson* execution time to a relatively small value in order to limit the number of spawned threads below the threshold which triggers the fork-bomb protection mechanism in the operating system.

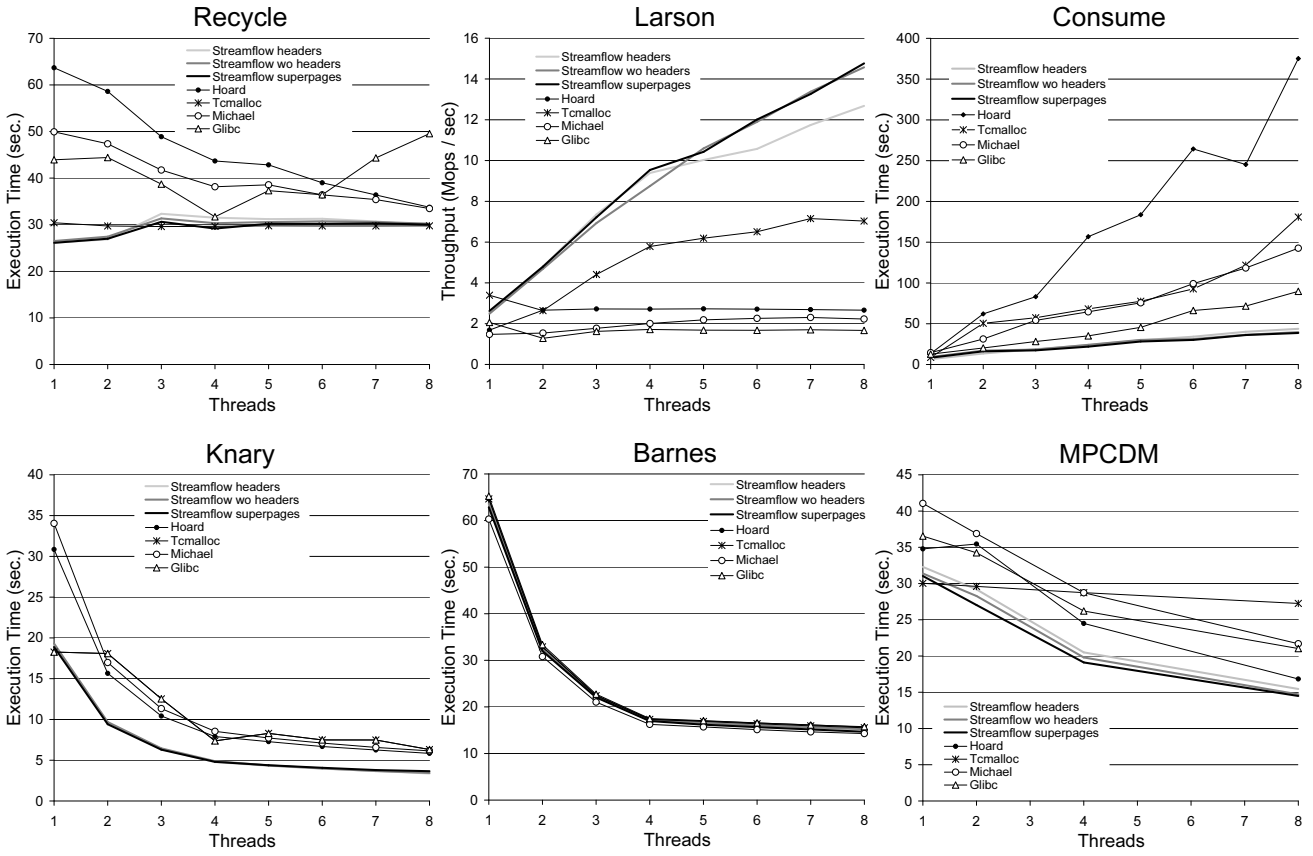

Figure 3. Execution time (lower is better) or throughput (higher is better) attained by different allocators.

Barnes: Hood implementation of the N-body Barnes-Hut force calculation algorithm [1]. *Barnes* has only limited sensitivity to allocation latency, particularly during the first iteration (time step) of the code, in which the main application data structures are created and initialized. The benchmark provides limited opportunities for exploiting spatial and temporal locality.

Streamflow improves the execution time of *Barnes* 4.9% on average over glibc, 3.6% over Tcmalloc and 4.3% over Hoard. *Barnes*, however, is the only application in which Michael's lock-free allocator outperforms Streamflow by 2.6% to 4.6% (3.6% on avg.).

The low intensity of memory management operations limits the performance improvements that can be attained by using different memory allocators. It should be noted, though, that the use of superpages by Streamflow, yields a 2% performance improvement, and a 13% reduction of minor page faults (from 9.1K to 7.9K).

MPCDM: This is a guaranteed-quality multithreaded mesh generator based on the Delaunay method [5]. For realistic problem sizes, it allocates hundreds of millions of small objects (35 bytes on average) which represent triangles and points in a mesh. The algorithm deletes triangles that do not satisfy quality criteria set by the user, as well as some of their neighbors. The resulting empty area is then re-triangulated. After the re-triangulation it consists of at least as many triangles as those deleted. As a result the memory footprint of the application increases monotonically. The application offers opportunities for temporal and spatial locality optimizations, stresses allocator memory reuse, and is sensitive to memory operation latency and allocator scalability. *MPCDM*'s scalability is limited by the frequent synchronization between its threads. It can serve as a case study of the extent of benefits that can be attained by efficient memory allocators, in the presence of other bottlenecks, unrelated to memory management.

The fully optimized Streamflow implementation outperforms glibc by 18% to 45% (32% on avg.). The improvements against Hoard and Michael's lock-free allocator range between 12% and 50% (22% and 42% on average respectively). Streamflow also performs up to 88% better than Tcmalloc (36% on average). It is clear that Streamflow can benefit complex scientific applications with intense memory management requirements, such as *MPCDM*. Especially in the presence of frequent, application-induced synchronization operations, Streamflow's mostly synchronization-free design practically eliminates additional, allocator induced contention between threads.

The elimination of headers allows more small objects to be placed inside a single memory page. It thus favors spatial locality at the page-level, reducing minor page faults by 49% (from 247K to 127K) in the 8-threads execution. This is reflected in a 4% performance improvement over the base Streamflow implementation. The use of superpages has similar effects. Minor page faults are limited to just 888 and performance improves by 6% compared with the base implementation.

4.2.3 Memory overhead

An important metric for the quality a multithreaded memory allocator is the memory overhead it introduces, quantified by the amount of virtual memory reserved by the allocator for a given stream of memory requests by the application. Table 2 shows the maximum virtual memory footprint of the seven benchmarks when executed with all allocators. The memory usage of each application was mea-

	Streamflow w/o headers	Streamflow w/ superpages	Hoard	Tcmalloc	Michael	Glibc
197.parser	17	19	–	18	27	16
Recycle	20	23	19	19	19	28
Larson	1487	1598	289	685	235	196
(stacks)	*1473*	*1581*	*273*	*671*	*218*	*174*
Consume	23	28	21	22	23	25
Knary	28	31	25	26	27	32
Barnes	45	47	46	41	43	46
MPCDM	530	538	1032	530	630	610

Table 2. Maximum virtual memory footprint of the benchmarks (in MB) when executed with different multithreaded allocators. Multithreaded benchmarks are executed with 8 threads. In the case of *Larson* we also report the memory required just for thread stacks in each case. *197.parser* crashes when executed with Hoard, so no footprint value is reported.

sured by querying the `/proc` filesystem[5] for each process' total virtual memory consumption every tenth of a second for the lifetime of the application. The value reported is the maximum observed for each application / memory allocator pair.

With the exception of *Larson*, Streamflow achieves virtual memory footprints smaller than glibc and comparable to the other allocators. *Larson* continuously generates threads that perform a constant number of allocation and deallocation operations, spawn new threads, and then terminate without ever being joined. Since *Larson* runs for a fixed time period, the number of threads spawned by the application is proportional to the achieved rate of allocation and deallocation operations. Tracing system calls performed by the application revealed that before each thread generation, 513 memory pages (2052 KB) are allocated for the thread's stack. The system calls trace also revealed that—as expected—since threads are never joined, their stacks are never freed and reused. As a result, the virtual memory footprint of the application is dominated by thread stacks. In fact, the virtual memory footprint grows monotonically during the execution life of the application, with a rate that is linearly related to the throughput of memory management operations achieved by each multithreaded memory manager. Table 2 reports—in the case of Larson—the total maximum virtual memory footprint of the application, as well as the maximum virtual memory footprint of thread stacks. The latter is calculated by multiplying the total number of threads generated by the application by 2052 KB (the stack size).

It is worth noting that Streamflow performs well even with *Consume*, which is specifically designed to stress multithreaded allocators that use thread-local heaps. Allocators which use strictly thread-local heaps are sensitive to memory blowup under producer-consumer memory usage patterns.

5. Discussion and Future Directions

Streamflow uses superpages as a tool to avoid cache conflicts through the allocation of page blocks directly in physical memory. The use of superpages also provides the necessary infrastructure to investigate cache-color aware placement of page blocks and demonstrate the potential of multilevel locality optimizations within a scalable memory allocator. However, imposing the use of superpages in all programs has certain disadvantages. Some of the most important ones are severe fragmentation for small programs and unjustified memory pressure, which may occur in a multiprogrammed system in which some of the programs make extensive use of superpages but utilize little space within each page. One way to address these problems is to leverage operating system support for dynamic superpage management [19].

Although Streamflow provides support for relinquishing page blocks back to the operating system, it does not do so adaptively, as a reaction to memory pressure [7]. Extending Streamflow with mechanisms and policies to detect memory pressure and proactively release memory to prevent thrashing is left as future work.

Streamflow was designed under the assumption that dynamic feedback such as actual object sizes and lifetimes is not available to the allocator [11, 22]. Such profiles enable customizations, such as reap-style object deallocation of short-lived objects [11], or object segregation based on access frequency and length of object lifetimes [22]. In general, profiling information has not been used so far in multiprocessor memory allocators and it is a path we would like to explore in the near future. Profiling may prove useful for customizing Streamflow's allocation and deallocation policies for exploiting more aggressively specific types of locality, such as locality in streams of accesses to objects from different classes.

As multicore and simultaneous multithreading processors become commonplace, it is important to consider the implications of these processors on multithreaded memory allocation. Some of the related considerations were discussed in [17]. The main challenge for a locality-conscious allocator for chip multiprocessors is making good use of a large shared on-chip cache. The fact that threads can share data through a cache requires the allocator to customize its page block management policies so that page blocks belonging to different threads that run on the same processor are allocated contiguously and conflict-free, if possible. Streamflow's design enables this optimization, pending the addition of feedback from the operating system so that the allocator becomes aware of the placement of threads on execution cores at runtime.

6. Conclusions

Multiprocessor memory allocators have so far capitalized on scalability. Optimized, sequential allocators, on the other hand, place emphasis on locality. In this paper we have presented Streamflow, a high-performance, low-overhead thread-safe memory allocator also designed to favor locality at several levels of the memory hierarchy.

Streamflow's design decouples local and remote operations in order to eliminate synchronization for most memory allocation operations, while still avoiding memory blowup which strictly local-heaps suffer from. In order to further reduce latency, all synchronization operations are non-blocking and lock-free. This scalable and locality-conscious design enables Streamflow to perform comparably to optimized sequential allocators, yet be usually significantly faster than other multiprocessor allocators. These properties are consolidated in a unified segregated heap design. Streamflow also improves cache-, TLB-, and page-level locality via careful layout of heaps in memory, careful reuse of freed objects and the exploitation of superpages. Put together, these properties make Streamflow an attractive unified framework for sequential and parallel memory allocation and a useful tool for taming the ever-increasing memory latencies in codes that rely heavily on dynamic memory allocation.

The design space for locality-conscious multiprocessor memory allocators is vast. Streamflow represents a realistic point in this design space and a step in the direction of composing adaptive memory allocators with sufficient self-customization capabilities for multiple design goals, such as locality and parallelism.

[5] `/proc` is a virtual filesystem available on most UNIX-like operating systems that exposes information from the OS kernel to user-level at runtime.

Acknowledgments

This work is supported by the National Science Foundation (Grants CCR-0346867 and ACI-0312980) the U.S. Department of Energy (Grant DE-FG02-05ER2568), and an equipment grant from the College of William and Mary.

References

[1] N. Arora, R. Blumofe, and C. Greg-Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. In *Proc. of the 10th ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 1998.

[2] D. Barrett and B. Zorn. Using Lifetime Predictors to Improve Memory Allocation Performance. In *Proc. of the 1993 ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 187–196, June 1993.

[3] E. Berger, K. Mckinley, R. Blumofe, and P. Wilson. Hoard: A Scalable Memory Allocator for Multithreaded Applications. In *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 117–128, Cambridge, MA, November 2000.

[4] E. Berger, B. Zorn, and K. McKinley. Reconsidering Custom Memory Allocation. In *Proc. of the 17th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applpications*, pages 1–12, Seattle, WA, November 2002.

[5] Filip Blagojevic. Optimizing Irregular Adaptive Application on Multi-Threaded Processors: The Case of Medium-Grain Parallel Delaunay Mesh Generation. Master's thesis, The College of William and Mary, Williamsburg, VA, U.S.A., December 2005.

[6] C. Cascaval, E. Duesterwald, P. Sweeney, and R. Wisniewski. Multiple Page Size Modeling and Optimization. In *Proc. of the 14th International Conference on Parallel Architectures and Compilation Techniques*, pages 339–349, Saint Louis, MO, September 2005.

[7] Y. Feng and E. Berger. A Locality-Improving Dynamic Memory Allocator. In *Proceedings of the Third Annual ACM SIGPLAN Workshop on Memory Systems Performance*, Chicago, IL, June 2005.

[8] D. Gay and A. Aiken. Memory Management with Explicit Regions. In *Proc. of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 313–323, Montreal, Canada, June 1998.

[9] Wolfram Gloger. Dynamic Memory Allocator Implementations in Linux System Libraries. http://www.dent.med.uni-muenchen.de/ wmglo/malloc-slides.html.

[10] Google. Google Performance Tools. http://goog-perftools.sourceforge.net/.

[11] D. Grunwald, B. Zorn, and R. Henderson. Improving the Cache Locality of Memory Allocation. In *Proc. of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 177–186, Albuquerque, NM, June 1993.

[12] P. Kamp. Malloc(3) Revisted. http://phk.freebsd.dk/pubs/malloc.pdf.

[13] K. C. Knowlton. A Fast Storage Allocator. *Communications of the ACM*, 8(10):623–625, 1965.

[14] D. E. Knuth. *Dynamic Storage Allocation. In The Art of Computer Programming*, volume 1. Addison-Wesley, 1968.

[15] P. Larson and M. Krishnan. Memory Allocation for Long-Running Server Applications. In *Proceedings of the First International Symposium on Memory Management*, pages 176–185, Vancouver, BC, October 1998.

[16] D. Lea. A Memory Allocator. http://gee.cs.oswego.edu/dl/html/malloc.html.

[17] L. McDowell, S. Eggers, and S. Gribble. Improving Server Software Support for Simultaneous Multithreaded Processors. In *Proc. of the 2003 ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37–48, San Diego, CA, June 2003.

[18] M. Michael. Scalable Lock-free Dynamic Memory Allocation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 35–46, Washington, DC, June 2004.

[19] J. Navarro, S. Iyer, and A. Cox. Practical, Transparent Operating System Support for Superpages. In *Proc. of the Fifth Symposium on Operating Systems Design and Implementation*, pages 89–104, Boston, MA, December 2002.

[20] T. Romer, W. Ohlrich, A. Karlin, and B. Bershad. Reducing TLB and Memory Overhead using Online Superpage Promotion. In *Proc. of the 22nd International Symposium on Computer Architecture*, pages 176–187, Santa Margherita Ligure, Italy, June 1995.

[21] D. Ross. The AED Free Storage Package. *Communications of the ACM*, 10(8):481–492, 1967.

[22] M. Seidl and B. Zorn. Segregating Heap Objects by Reference Behavior and Lifetime. In *Proc. of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, San Jose, CA, October 1998.

[23] Y. Shuf, M. Gupta, R. Bordawekar, and J. Pal Singh. Exploiting Prolific Types for Memory Management and Optimizations. In *Proc. of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Lanugages*, pages 295–306, Portland, OR, January 2002.

[24] G. Steele. Data Representation in PDP-10 MACLISP. Technical Report AI Lab Memo 421, MIT, 1977.

[25] V. Vee and W. Hsu. A Scalable and Efficient Storage Allocator on Shared Memory Multiprocessors. In *Proceedings of the 1999 International Symposium on Parallel Architectures, Algorithms and Networks*, pages 230–235, Perth, Australia, June 1999.

[26] K. Vo. Vmalloc: A General and Efficient Memory Allocator. *Software Practice and Experience*, 26(3):357–374, 1996.

[27] P. Wilson, M. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Proc. of the International Workshop on Memory Management, LNCS Vol. 986*, pages 1–116, Kinross, UK, September 1995.