Composing High-Performance Memory Allocators

Emery D. Berger
Dept. of Computer Sciences
The University of Texas at Austin
Austin, TX 78712
emery@cs.utexas.edu

Benjamin G. Zorn
Microsoft Research
One Microsoft Way
Redmond, WA 98052
zorn@microsoft.com

Kathryn S. McKinley
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
mckinley@cs.umass.edu

ABSTRACT

Current general-purpose memory allocators do not provide sufficient speed or flexibility for modern high-performance applications. Highly-tuned general purpose allocators have per-operation costs around one hundred cycles, while the cost of an operation in a custom memory allocator can be just a handful of cycles. To achieve high performance, programmers often write custom memory allocators from scratch – a difficult and error-prone process.

In this paper, we present a flexible and efficient infrastructure for building memory allocators that is based on C++ templates and inheritance. This novel approach allows programmers to build custom and general-purpose allocators as "heap layers" that can be composed without incurring any additional runtime overhead or additional programming cost. We show that this infrastructure simplifies allocator construction and results in allocators that either match or improve the performance of heavily-tuned allocators written in C, including the Kingsley allocator and the GNU obstack library. We further show this infrastructure can be used to rapidly build a general-purpose allocator that has performance comparable to the Lea allocator, one of the best uniprocessor allocators available. We thus demonstrate a clean, easy-to-use allocator interface that seamlessly combines the power and efficiency of any number of general and custom allocators within a single application.

1. Introduction

Many general-purpose memory allocators implemented for C and C++ provide good runtime and low fragmentation for a wide range of applications [15, 17]. However, using specialized memory allocators that take advantage of application-specific behavior can dramatically improve application performance [4, 13, 26]. Hand-coded custom allocators are widely used: three of the twelve integer benchmarks included in SPEC2000 (`parser`, `gcc`, and `vpr` [22]) and several server applications, including the Apache web server [1] and Microsoft's SQL Server [10], use one or more custom allocators.

Custom allocators can take advantage of certain allocation patterns with extremely low-cost operations. For example, a programmer can use a *region* allocator to allocate a number of small objects with a known lifetime and then free them all at once [11, 19, 23]. This customized allocator returns individual objects from a range of memory (i.e., a region), and then deallocates the entire region. The per-operation cost for a region-based allocator is only a handful of cycles (advancing a pointer and bounds-checking), whereas for highly-tuned, general-purpose allocators, the per-operation cost is around one hundred cycles [11]. Other custom allocators can yield similar advantages over general-purpose allocators. Figure 1 shows the estimated impact of the custom allocator used in the SPEC benchmark `197.parser`, running on its test input. Replacing its custom allocator by the system allocator increases its runtime by over 60%.[1]

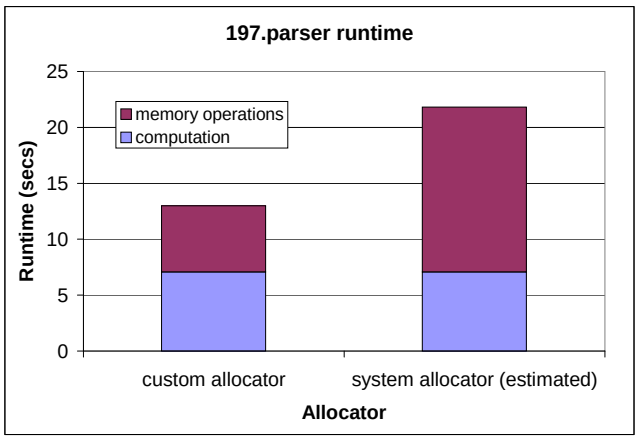


Figure 1: The impact of custom allocation on performance for `197.parser`.

To attain high performance, programmers often write their own ad hoc custom allocators as macros or monolithic functions in order to avoid function call overhead. This approach to improving application performance is enshrined among the best practices of

This work is supported by NSF grant EIA-9726401, NSF Infrastructure grant CDA-9502639, NSF ITR grant CCR-0085792, and DARPA grant 5-21425. Part of this work was done while the first author was at Microsoft Research. Emery Berger was also supported by a grant from Microsoft Corporation. Any opinions, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.

PLDI 2001

[1] We estimated this cost by measuring the time spent in allocation using `197.parser`'s custom allocator and computing a conservative estimate of allocation time with the system allocator (which cannot directly be substituted because of the semantics of the custom allocator). This and the other programs in this paper were compiled with Visual C++ 6.0 and run under Windows 2000 on a 366 MHz Pentium II system.

skilled computer programmers [8, 18]. Unfortunately, this kind of code is brittle and hard to maintain or reuse, and as the application evolves, it can be difficult to adapt the memory allocator as needs change (e.g., to a multithreaded environment). In addition, writing these allocators is both error-prone and difficult. Good memory allocators are complicated pieces of software that require a substantial engineering effort. Because of this complexity, combining a custom and general-purpose allocator to allow them to share free memory, for example, is currently almost impossible.

In this paper, we present a flexible and efficient infrastructure for building custom and general-purpose allocators called *heap layers*. This infrastructure is based on a combination of C++ templates and inheritance called *mixins* [7]. Mixins are classes whose superclass may be changed. Using mixins allows the programmer to code allocators as composable layers that a compiler can implement with efficient code. Unlike previous approaches, we show that this technique allows programmers to write highly modular and reusable code with no abstraction penalty. We describe a number of high-performance custom allocators that we built by mixing and matching heap layers. We show that these allocators match or improve performance when compared with their hand-tuned, monolithic C counterparts on a selection of C and C++ programs.

We further demonstrate that this infrastructure can be used effectively to build high-performance, general-purpose allocators. We evaluate two general-purpose allocators we developed using heap layers over a period of three weeks, and compare their performance with the Kingsley allocator, one of the fastest general-purpose allocators, and the Lea allocator, an allocator that is both fast and memory-efficient. While the current heap layers allocator does not achieve the fragmentation and performance of the Lea allocator, the Lea allocator is highly tuned and has undergone many revisions over a period of more than seven years [17].

The remainder of this paper is organized as follows. We discuss related work in Section 2. In Section 3, we describe how we use mixins to build heap layers and demonstrate how we can mix and match a few simple heap layers to build and combine allocators. We briefly discuss our experimental methodology in Section 4. In Section 5, we show how we implement some real-world custom allocators using heap layers and present performance results. Section 6 then describes two general-purpose allocators built with heap layers and compares their runtime and memory consumption to the Kingsley and Lea allocators. We describe some of the software engineering benefits of heap layers in Section 7, and in Section 8, we show how heap layers provide a convenient infrastructure for memory allocation experiments. We conclude in Section 9 with a discussion of future directions.

2. Related Work

In this section, we describe related work in memory allocation and memory management infrastructures. We discuss two representative general-purpose memory allocators and describe related work in custom memory allocation. We then compare heap layers to previous infrastructures for building memory managers.

2.1 General-Purpose Allocation

The literature on general-purpose memory allocators is extensive [26]. Here we describe two memory allocators, the Kingsley allocator used in BSD 4.2 [26] and the Lea allocator [17]. In Section 6, we describe the implementation of these two allocators in heap layers. We chose to implement these allocators because they are in widespread use and are on opposite ends of the spectrum between maximizing speed and minimizing memory consumption.

The Kingsley allocator is a power-of-two segregated fits allocator: all allocation requests are rounded up to the next power of two. This rounding can lead to severe *internal fragmentation* (wasted space inside allocated objects), because in the worst case, it allocates twice as much memory as requested. Once an object is allocated for a given size, it can never be reused for another size: the allocator performs no *splitting* (breaking large objects into smaller ones) or *coalescing* (combining adjacent free objects). This algorithm is well known to be among the fastest memory allocators although it is among the worst in terms of fragmentation [15].

The Lea allocator is an approximate best-fit allocator that provides both high speed and low memory consumption. It forms the basis of the memory allocator included in the GNU C library [12]. The current version (2.7.0) is a hybrid allocator with different behavior based on object size. Small objects (less than 64 bytes) are allocated using exact-size quicklists (one linked list of freed objects for each multiple of 8 bytes). Requests for a medium-sized object (less than 128K) and certain other events trigger the Lea allocator to coalesce all of the objects in these quicklists in the hope that this reclaimed space can be reused for the medium-sized object. For medium-sized objects, the Lea allocator performs immediate coalescing and spliting and approximates best-fit. Large objects are allocated and freed using `mmap`. The Lea allocator is the best overall allocator (in terms of the combination of speed and memory usage) of which we are aware [15].

2.2 Special-Purpose Allocation

Most research on special-purpose (custom) allocation has focused on profile-based optimization of general-purpose allocation. Grunwald and Zorn's CustoMalloc builds memory allocators from allocation traces, optimizing the allocator based on the range of object sizes and their frequency of usage [13]. Other profile-based allocators use lifetime information to improve performance and reference information to improve locality [4, 20]. Regions, described in Section 1, have received recent attention as a method for improving locality [23]. Aiken and Gay have developed *safe* regions which delay region deletion when live objects are present [11]. Techniques for building other application-specific custom allocators have received extensive attention in the popular press [8, 18].

2.3 Memory Management Infrastructures

We know of only two previous infrastructures for building memory managers: *vmalloc*, by Vo, and *CMM*, by Attardi, Flagella, and Iglio. We describe the key differences between their systems and ours, focusing on the performance and flexibility advantages that heap layers provide.

The most successful customizable memory manager of which we are aware is the *vmalloc* allocator [25]. Vmalloc lets the programmer define multiple regions (distinct heaps) with different disciplines for each. The programmer performs customization by supplying user-defined functions and `structs` that manage memory. By chaining these together, vmalloc does provide the possibility of composing heaps. Unlike heap layers, each abstraction layer pays the penalty of a function call. This approach often prevents many useful optimizations, in particular method inlining. The vmalloc infrastructure limits the programmer to a small set of functions for memory allocation and deallocation; a programmer cannot add new functionality or new methods as we describe in Section 6.1. Unlike heap layers, vmalloc does not provide a way to delete heaps and reclaim all of their memory in one step. These limitations dramatically reduce vmalloc's usefulness as an extensible infrastructure.

Attardi, Flagella, and Iglio created an extensive C++-based system called the Customizable Memory Management (CMM) framework [2, 3]. Unlike our work, the primary focus of the CMM framework is garbage collection. The only non-garbage collected heaps provided by the framework are a single "traditional man-

ual allocation discipline" heap (whose policy the authors do not specify) called UncollectedHeap and a zone allocator called TempHeap. A programmer can create separate regions by subclassing the abstract class CmmHeap, which uses virtual methods to obtain and reclaim memory. For every memory allocation, deallocation, and crossing of an abstraction boundary, the programmer must thus pay the cost of one virtual method call, while in heap layers, there is no such penalty. As in vmalloc, this approach often prevents compiler optimizations across method boundaries. The virtual method approach also limits flexibility. In CMM, subclasses cannot implement functions not already provided by virtual methods in the base heap. Also, since class hierarchies are fixed, it is not possible to have one class (such as FreelistHeap, described in Section 3.1) with two different parent heaps in different contexts. In contrast, the mixin-based approach taken by heap layers allows both inheritance and reparenting, making heap layers more flexible and reusable.

The goal of heap layers is to provide the performance of existing custom and general-purpose allocators in a flexible, reusable framework that provides a foundation for programmers to build new allocators. We implement customized and general-purpose allocators using heap layers, demonstrating their flexibility and competitive performance.

3. Heap Layers

While programmers often write memory allocators as monolithic pieces of code, they tend to think of them as consisting of separate pieces. Most general-purpose allocators treat objects of different sizes differently. The Lea allocator uses one algorithm for small objects, another for medium-sized objects, and yet another for large objects. Conceptually at least, these heaps consist of a number of separate heaps that are combined in a hierarchy to form one big heap.

The standard way to build components like these in C++ uses virtual method calls at each abstraction boundary. The overhead caused by virtual method dispatch is significant when compared with the cost of memory allocation. This implementation style also greatly limits the opportunities for optimization since the compiler often cannot optimize across method boundaries. Building a class hierarchy through inheritance also fixes the relationships between classes in a single inheritance structure, making reuse difficult.

To address these concerns, we use *mixins* to build our heap layers. Mixins are classes whose superclass may be changed (they may be reparented) [7]. The C++ implementation of mixins (first described by VanHilst and Notkin [24]) consists of a templated class that subclasses its template argument:

```
template <class Super>
class Mixin : public Super {};
```

Mixins overcome the limitation of a single class hierarchy, enabling the reuse of classes in different hierarchies. For instance, we can use A in two different hierarchies, $A \rightarrow B$ and $A \rightarrow C$ (where the arrow means "inherits from"), by defining A as a mixin and composing the classes as follows:

```
class Composition1 : public A<B> {};
class Composition2 : public A<C> {};
```

A heap layer is a mixin that provides a `malloc` and `free` method and that follows certain coding guidelines. The `malloc` function returns a memory block of the specified size, and the `free` function deallocates the block. As long as the heap layer follows the guidelines we describe below, programmers can easily compose heap layers to build heaps. One layer can obtain memory from its parent by calling `SuperHeap::malloc()` and can return it with `SuperHeap::free()`. Heap layers also implement thin wrappers around system-provided memory allocation functions like `malloc`, `sbrk`, or `mmap`. We term these thin-wrapper layers *top heaps*, because they appear at the top of any hierarchy of heap layers.

We require that heap layers adhere to the following coding guidelines in order to ensure composability. First, `malloc` must correctly handle NULLs returned by `SuperHeap::malloc()` to allow an out-of-memory condition to propagate through a series of layers or to be handled by an exception-handling layer. Second, the layer's destructor must free any memory that is held by the layer. This action allows heaps composed of heap layers to be deleted in their entirety in one step.

3.1 Example: Composing a Per-Class Allocator

One common way of improving memory allocation performance is to allocate all objects from a highly-used class from a per-class pool of memory. Because all such objects are the same size, memory can be managed by a simple singly-linked freelist [16]. Programmers often implement these per-class allocators in C++ by overloading the `new` and `delete` operators for the class.

Below we show how we can combine two simple heap layers to implement per-class pools without changing the source code for the original class. We first define a utility class called PerClassHeap that allows a programmer to adapt a class to use any heap layer as its allocator:

```
template <class Object, class SuperHeap>
class PerClassHeap : public Object {
public:
  inline void * operator new (size_t sz) {
    return getHeap().malloc (sz);
  }
  inline void operator delete (void * ptr) {
    getHeap().free (ptr);
  }
private:
  static SuperHeap& getHeap (void) {
    static SuperHeap theHeap;
    return theHeap;
  }
};
```

We build on the above with a very simple heap layer called FreelistHeap. This layer implements a linked list of free objects of the same size. `malloc` removes one object from the freelist if one is available, and `free` places memory on the freelist for later reuse. This approach is a common idiom in allocators because it provides very fast allocates and frees. However, it is limited to handling only one size of object. The code for FreelistHeap appears in Figure 2 without the error checking included in the actual code to guarantee that all objects are the same size.

We can now combine PerClassHeap and FreelistHeap with mallocHeap (a thin layer over the system-supplied `malloc` and `free`) to make a subclass of *Foo* that uses per-class pools.

```
class FasterFoo :
public
  PerClassHeap<Foo, FreelistHeap<mallocHeap> >{};
```

3.2 A Library of Heap Layers

We have built a comprehensive library of heap layers that allows programmers to build a range of memory allocators with minimal effort by composing these ready-made layers. Figure 1 lists a number of these layers, which we group into the following categories:

Top heaps. A "top heap" is a heap layer that provides memory directly from the system and at least one appears at the top

```
template <class SuperHeap>
class FreelistHeap : public SuperHeap {
public:
  FreelistHeap (void)
    : myFreeList (NULL)
  {}
  ~FreelistHeap (void) {
    // Delete everything on the freelist.
    void * ptr = myFreeList;
    while (ptr != NULL) {
      void * oldptr = ptr;
      ptr = (void *) ((freeObject *) ptr)->next;
      SuperHeap::free (oldptr);
    }
  }
  inline void * malloc (size_t sz) {
    // Check the freelist first.
    void * ptr = myFreeList;
    if (ptr == NULL) {
      ptr = SuperHeap::malloc (sz);
    } else {
      myFreeList = myFreeList->next;
    }
    return ptr;
  }
  inline void free (void * ptr) {
    // Add this object to the freelist.
    ((freeObject *) ptr)->next = myFreeList;
    myFreeList = (freeObject *) ptr;
  }
private:
  class freeObject {
  public:
    freeObject * next;
  };
  freeObject * myFreeList;
};
```

Figure 2: The implementation of FreelistHeap.

of any hierarchy of heap layers. These thin wrappers over system-based memory allocators include mallocHeap, mmapHeap (virtual memory), and sbrkHeap (built using `sbrk()` for UNIX systems and an `sbrk()` emulation for Windows).

Building-block heaps. Programmers can use these simple heaps in combination with other heaps described below to implement more complex heaps. We provide an adapter called AdaptHeap that lets us embed a dictionary data structure inside freed objects so we can implement variants of FreelistHeap, including DLList, a FIFO-ordered, doubly-linked freelist that allows constant-time removal of objects from anywhere in the freelist. This heap supports efficient coalescing of adjacent objects belonging to different freelists into one object.

Combining heaps. These heaps combine a number of heaps to form one new heap. These include two segregated-fits layers, SegHeap and StrictSegHeap (described in Section 6.1), and HybridHeap, a heap that uses one heap for objects smaller than a given size and another for larger objects.

Utility layers. Utility layers include ANSIWrapper, which provides ANSI-C compliant behavior for `malloc` and `free` to allow a heap layer to replace the system-supplied allocator. A number of layers supply multithreaded support, including LockedHeap, which *code-locks* a heap for thread safety (acquires a lock, performs a `malloc` or `free`, and then releases the lock), and ThreadHeap and PHOThreadHeap, which implement finer-grained multithreaded support. Error handling is provided by ThrowExceptionHeap, which throws an exception when its superheap is out of memory. We also provide heap debugging support with DebugHeap, which tests for multiple frees and other common memory management errors.

A Library of Heap Layers	
Top Heaps	
mallocHeap	A thin layer over `malloc`
mmapHeap	A thin layer over the virtual memory manager
sbrkHeap	A thin layer over `sbrk` (contiguous memory)
Building-Block Heaps	
AdaptHeap	Adapts data structures for use as a heap
BoundedFreelistHeap	A freelist with a bound on length
ChunkHeap	Manages memory in chunks of a given size
CoalesceHeap	Performs coalescing and splitting
FreelistHeap	A freelist (caches freed objects)
Combining Heaps	
HybridHeap	Uses one heap for small objects and another for large objects
SegHeap	A general segregated fits allocator
StrictSegHeap	A strict segregated fits allocator
Utility Layers	
ANSIWrapper	Provides ANSI-malloc compliance
DebugHeap	Checks for a variety of allocation errors
LockedHeap	Code-locks a heap for thread safety
PerClassHeap	Use a heap as a per-class allocator
PHOThreadHeap	A private heaps with ownership allocator [6]
ProfileHeap	Collects and outputs fragmentation statistics
ThreadHeap	A pure private heaps allocator [6]
ThrowExceptionHeap	Throws an exception when the parent heap is out of memory
TraceHeap	Outputs a trace of allocations
UniqueHeap	A heap type that refers to one heap object
Object Representation	
CoalesceableHeap	Provides support for coalescing
SizeHeap	Records object sizes in a header
Special-Purpose Heaps	
ObstackHeap	A heap optimized for stack-like behavior and fast resizing
ZoneHeap	A zone ("region") allocator
XallocHeap	A heap optimized for stack-like behavior
General-Purpose Heaps	
KingsleyHeap	Fast but high fragmentation
LeaHeap	Not quite as fast but low fragmentation

Table 1: A library of heap layers, divided by category.

Object representation. SizeHeap maintains object size in a header just preceding the object. CoalesceableHeap does the same but also records whether each object is free in the header of the next object in order to facilitate coalescing.

Special-purpose heaps. We implemented a number of heaps optimized for managing objects with known lifetimes, including two heaps for stack-like behavior (ObstackHeap and XallocHeap, described in Sections 5.1 and 5.2) and a region-based allocator (ZoneHeap).

General-purpose heaps. We also implement two heap layers useful for general-purpose memory allocation: KingsleyHeap and LeaHeap, described in Sections 6.1 and 6.2.

4. Experimental Methodology

We wrote these heap layers in C++ and implemented them as a series of include files. We then used these heap layers to replace a number of custom and general-purpose allocators. For C++ programs, we used these heap layers directly (e.g., `kHeap.free(p)`). When replacing custom allocators in C programs, we wrapped the heap layers with a C API. When replacing the general-purpose allocators, we redefined `malloc` and `free` and the C++ operators `new` and `delete` to refer to the desired allocator. Programs were compiled with Visual C++ 6.0 and run on a 366 MHz Pentium II system with 128MB of RAM and a 256K L2 cache, under Windows 2000 build 2195. All runtimes are the average of three runs; variation was minimal.

5. Building Special-Purpose Allocators

In this section, we investigate the performance implications of building custom allocators using heap layers. Specifically, we evaluate the performance of two applications (`197.parser` and `176.gcc` from the SPEC2000 benchmark suite) that make extensive use of custom allocators. We compare the performance of the original carefully-tuned allocators against versions of the allocators that we wrote with heap layers.[2]

5.1 197.parser

The `197.parser` benchmark is a natural-language parser for English written by Sleator and Temperley. It uses a custom allocator the authors call *xalloc* which is optimized for stack-like behavior. This allocator uses a fixed-size region of memory (in this case, 30MB) and always allocates after the last block that is still in use by bumping a pointer. Freeing a block marks it as free, and if it is the last block, the allocator resets the pointer back to the new last block in use. Xalloc can free the entire heap quickly by setting the pointer to the start of the memory region. This allocator is a good example of appropriate use of a custom allocator. As in most custom allocation strategies, it is not appropriate for general-purpose memory allocation. For instance, if an application never frees the last block in use, this algorithm would exhibit unbounded memory consumption.

We replaced xalloc with a new heap layer, XallocHeap. This layer, which we layer on top of MmapHeap, is the same as the original allocator, except that we replaced a number of macros by inline static functions. We did not replace the general-purpose allocator which uses the Windows 2000 heap. We ran `197.parser` against the SPEC test input to measure the overhead that heap layers added. Figure 3 presents these results. We were quite surprised to find that using layers actually slightly *reduced* runtime (by just over 1%), although this reduction is barely visible in the graph. The source of this small improvement is due to the increased opportunity for code reorganization that layers provide. When using layers, the compiler can schedule code with much greater flexibility. Since each layer is a direct procedure call, the compiler can decide what pieces of the layered code are most appropriate to inline at each point in the program. The monolithic implementations of `xalloc`/`xfree` in the original can only be inlined in their entirety. Table 2 shows that the executable sizes for the original benchmark are the smallest when the allocation functions are not declared inline and the largest when they are inlined, while the version with XallocHeap lies in between (the compiler inlined the allocation functions with XallocHeap regardless of our use of `inline`). Inspecting the assembly output reveals that the compiler made more fine-grained decisions on what code to inline and thus achieved a better trade-off between program size and optimization opportunities to yield improved performance.

[2]We did not include `175.vpr`, the other benchmark in SPEC2000 that uses custom allocators, because its custom memory allocation API exposes too much internal allocator state to allow us to cleanly replace the allocator.

`197.parser` variant	Executable size
original	211,286
original (inlined)	266,342
XallocHeap	249,958
XallocHeap (inlined)	249,958

Table 2: Executable sizes for variants of `197.parser`.

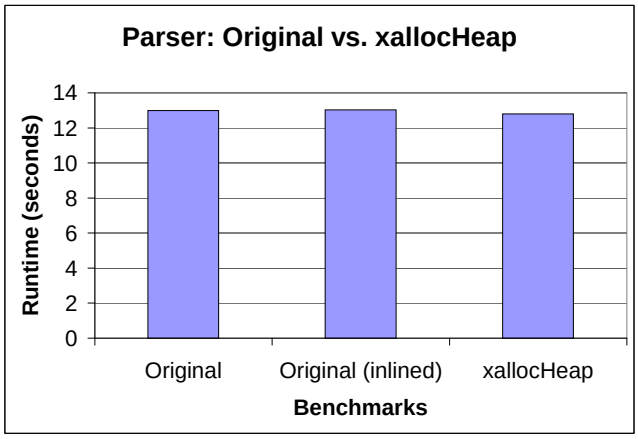


Figure 3: Runtime comparison of the original `197.parser` custom allocator and xallocHeap.

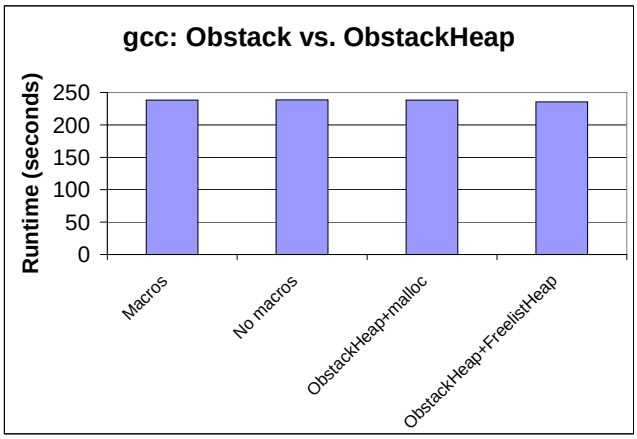
5.2 176.gcc

Gcc uses *obstacks*, a well-known custom memory allocation library [26]. Obstacks also are designed to take advantage of stack-like behavior, but in a more radical way than xalloc. Obstacks consist of a number of large memory "chunks" that are linked together. Allocation of a block bumps a pointer in the current chunk, and if there is not enough room in a given chunk, the obstack allocator obtains a new chunk from the system. Freeing an object deallocates all memory allocated after that object. Obstacks also support a `grow()` operation. The programmer can increase the size of the current block, and if this block becomes too large for the current chunk, the obstack allocator copies the current object to a new, larger chunk.

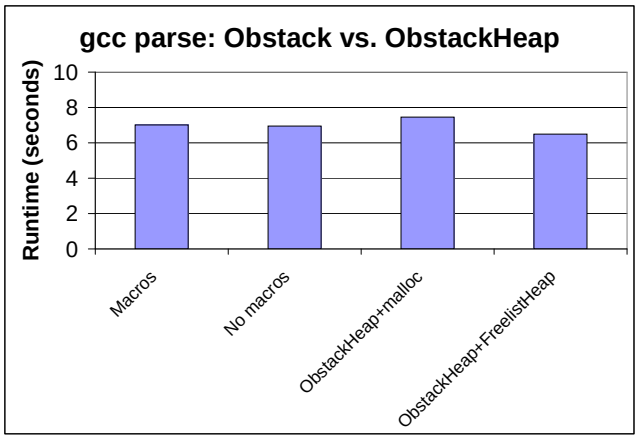
Gcc uses obstacks in a variety of phases during compilation. The parsing phase in particular uses obstacks extensively. In this phase, gcc uses the obstack grow operation for symbol allocation in order to avoid a fixed limit on symbol size. When entering each lexical scope, the parser allocates objects on obstacks. When leaving scope, it frees all of the objects allocated within that scope by freeing the first object it allocated.

Obstacks have been heavily optimized over a number of years and make extensive use of macros. We implemented ObstackHeap in heap layers and provided C-based wrapper functions that implement the obstack API. This effort required about one week and consists of 280 lines of code (around 100 are to implement the API wrappers). By contrast, the GNU obstack library consists of around 480 lines of code and was refined over a period of at least six years.

We ran gcc on one of the reference inputs (scilab.i) and compared two versions of the original gcc with two versions of gcc with ObstackHeap: the original macro-based code, the original with function calls instead of macros, the ObstackHeap version layered on top of mallocHeap, and an ObstackHeap version that uses a Free-



(a) Complete execution of gcc.



(b) gcc's parse phase only.

Figure 4: Runtime comparison of gcc with the original obstack and ObstackHeap.

Memory-Intensive Benchmarks		
Benchmark	Description	Input
cfrac	factors numbers	a 36-digit number
espresso	optimizer for PLAs	*test2*
lindsay	hypercube simulator	*script.mine*
LRUsim	a locality analyzer	an 800MB trace
Perl	Perl interpreter	*perfect.in*
roboop	Robotics simulator	included benchmark

Table 3: Memory-intensive benchmarks used in this paper.

listHeap to optimize allocation and freeing of the default chunk size and mallocHeap for larger chunks:

```
class ObstackType :
  public ObstackHeap<4096,
    HybridHeap<4096 + 8,  // Obstack overhead
               FreelistHeap<mallocHeap>,
               mallocHeap> {};
```

As with `197.parser`, we did not replace the general-purpose allocator. Figure 4(a) shows the total execution time for each of these cases, while Figure 4(b) shows only the parse phase. Layering ObstackHeap on top of FreelistHeap results in an 8% improvement over the original in the parse phase, although its improvement over the original for the full execution of gcc is minimal (just over 1%). This result provides further evidence that custom allocators composed quickly with heap layers can perform comparably to carefully tuned hand-written allocators.

6. Building General-Purpose Allocators

In this section, we consider the performance implications of building general-purpose allocators using heap layers. Specifically, we compare the performance of the Kingsley and Lea allocators [17] to allocators with very similar architectures created by composing heap layers. Our goal is to understand whether the performance costs of heap layers prevent the approach from being viable for building general-purpose allocators. We map the designs of these allocators to heap layers and then compare the runtime and memory consumption of the original allocators to our heap layer implementations, KingsleyHeap and LeaHeap.

To evaluate allocator runtime performance and fragmentation, we use a number of memory-intensive programs, most of which were described by Zorn and Wilson [14, 15] and shown in Table 3: *cfrac* factors arbitrary-length integers, *espresso* is an optimizer for programmable logic arrays, *lindsay* is a hypercube simulator, *LRUsim* analyzes locality in reference traces, *perl* is the Perl interpreter included in SPEC2000 (`253.perlbmk`), and *roboop* is a robotics simulator. As Table 4 shows, these programs exercise memory allocator performance in both speed and memory efficiency. This table also includes the number of objects allocated and their average size. The programs' footprints range from just 16K (for *roboop*) to over 1.5MB (for *LRUsim*). For all of the programs except *lindsay* and *LRUsim*, the ratio of total memory allocated to the maximum amount of memory in use is large. The programs' rates of memory allocation and deallocation (memory operations per second) range from under one hundred to almost two million per second. Except for *LRUsim*, memory operations account for a significant portion of the runtime of these programs.

6.1 The Kingsley Allocator

We first show how we can build KingsleyHeap, a complete general-purpose allocator using the FreelistHeap layer described in Section 3.1 composed with one new heap layer. We show that KingsleyHeap, built using heap layers, performs as well as the Kingsley allocator.

The Kingsley allocator needs to know the sizes of allocated objects so it can place them on the appropriate free list. An object's size is often kept in metadata just before the object itself, but it can be represented in other ways. We can abstract away object representation by relying on a `getSize()` method that must be implemented by a superheap. SizeHeap is a layer that records object size in a header immediately preceding the object.

```
template <class SuperHeap>
class SizeHeap : public SuperHeap {
public:
  inline void * malloc (size_t sz) {
    // Add room for a size field.
    freeObject * ptr = (freeObject *)
      SuperHeap::malloc (sz + sizeof(freeObject));
    // Store the requested size.
    ptr->sz = sz;
    return (void *) (ptr + 1);
  }
  inline void free (void * ptr) {
    SuperHeap::free ((freeObject *) ptr - 1);
  }
```

Memory-Intensive Benchmark Statistics						
Benchmark	Objects	Total memory	Max in use	Average size	Memory ops	Memory ops/sec
cfrac	10,890,166	222,745,704	176,960	20.45	21,780,289	1,207,862
espresso	4,477,737	1,130,107,232	389,152	252.38	8,955,367	218,276
lindsay	108,862	7,418,120	1,510,840	68.14	217,678	72,300
LRUsim	39,139	1,592,992	1,581,552	40.70	78,181	94
perl	8,548,435	162,451,960	293,928	19.00	17,091,308	257,809
roboop	9,268,221	332,058,248	16,376	35.83	18,536,397	1,701,786

Table 4: Statistics for the memory-intensive benchmarks used in this paper. We divide by runtime with the Lea allocator to obtain memory operations per second.

```
  inline static size_t getSize (void * ptr) {
    return ((freeObject *) ptr - 1)->sz;
  }
private:
  union freeObject {
    size_t sz;
    double _dummy; // for alignment.
  };
};
```

StrictSegHeap provides a general interface for implementing strict segregated fits allocation. Segregated fits allocators divide objects into a number of *size classes*, which are ranges of object sizes. Memory requests for a given size are satisfied directly from the "bin" corresponding to the requested size class. The heap returns deallocated memory to the appropriate bin. StrictSegHeap's arguments include the number of bins, a function that maps object size to size class and size class to maximum size, the heap type for each bin, and the parent heap (for bigger objects). The implementation of StrictSegHeap is 32 lines of C++ code. The class definition appears in Figure 5.

We now build KingsleyHeap using these layers. First, we implement the helper functions that support power-of-two size classes:

```
inline int pow2getSizeClass (size_t sz) {
  int c = 0;
  sz = sz - 1;
  while (sz > 7) {
    sz >>= 1;
    c++;
  }
  return c;
}
inline size_t pow2getClassMaxsize (int i) {
  return 1 << (i+3);
}
```

By combining these heap layers, we can now define KingsleyHeap. We implement KingsleyHeap as a StrictSegHeap with 29 bins and power-of-two size classes (supporting an object size of up to $2^{32}-1$ bytes). Each size class is implemented using a FreelistHeap that gets memory from SbrkHeap (a thin layer over `sbrk()`).

```
class KingsleyHeap :
public StrictSegHeap<29, pow2getSizeClass,
         pow2getClassMaxSize,
         SizeHeap<FreelistHeap<SbrkHeap> >,
         SizeHeap<FreelistHeap<SbrkHeap> > > {};
```

A C++ programmer now uses this heap by declaring it as an object and directly using the `malloc` and `free` calls.

```
KingsleyHeap kHeap;
char * ch = (char *) kHeap.malloc (20);
kHeap.free (ch);
```

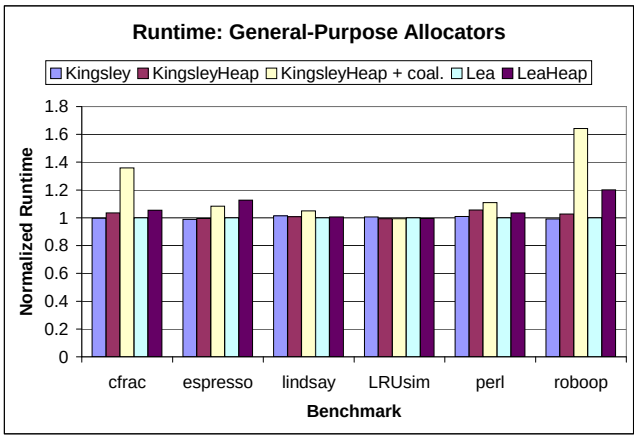
```
template <int NumBins,
          int (*getSizeClass) (size_t),
          size_t (*getClassMaxSize) (int),
          class LittleHeap,
          class BigHeap>
class StrictSegHeap : public BigHeap {
public:
  inline void * malloc (size_t sz) {
    void * ptr;
    int sizeClass = getSizeClass (sz);
    if (sizeClass >= NumBins) {
      // This request was for a "big" object.
      ptr = BigHeap::malloc (sz);
    } else {
      size_t ssz = getClassMaxSize(sizeClass);
      ptr = myLittleHeap[sizeClass].malloc (ssz);
    }
    return ptr;
  }
  inline void free (void * ptr) {
    size_t objectSize = getSize(ptr);
    int objectSizeClass
      = getSizeClass (objectSize);
    if (objectSizeClass >= NumBins) {
      BigHeap::free (ptr);
    } else {
      while (getClassMaxSize(objectSizeClass)
             > objectSize) {
        objectSizeClass--;
      }
      myLittleHeap[objectSizeClass].free (ptr);
    }
  }
private:
 LittleHeap myLittleHeap[NumBins];
};
```

Figure 5: The implementation of StrictSegHeap.

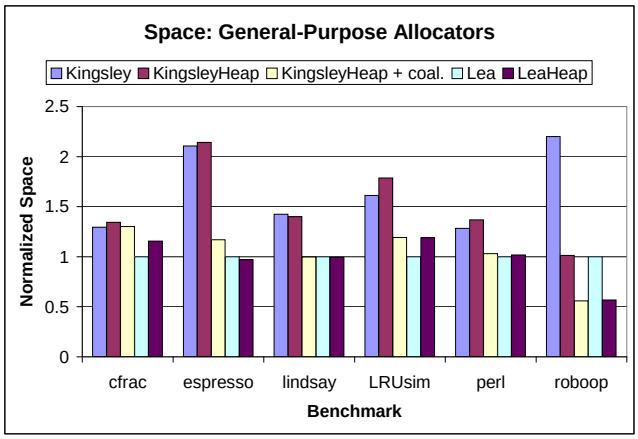
6.2 The Lea Allocator

Version 2.7.0 of the Lea allocator is a hybrid allocator with different behavior for different object sizes. For small objects (≤ 64 bytes), the allocator uses quick lists; for large objects ($\geq 128K$ bytes), it uses virtual memory (`mmap`), and for medium-sized objects, it performs approximate best-fit allocation [17]. The strategies it employs are somewhat intricate but it is possible to decompose these into a hierarchy of layers.

Figure 7 shows the heap layers representation of LeaHeap, which is closely modeled after the Lea allocator. The shaded area represents LeaHeap, while the Sbrk and Mmap heaps depicted at the top are parameters. At the bottom of the diagram, object requests are managed by a SelectMmapHeap, which routes large size requests to be eventually handled by the Mmap parameter. Smaller requests are routed to ThresholdHeap, which both routes size requests to a small and medium heap and in certain instances (e.g., when a



(a) Runtime normalized to the Lea allocator.



(b) Space (memory consumption) normalized to the Lea allocator.

Figure 6: Runtime and space comparison of the original Kingsley and Lea allocators and their heap layers counterparts.

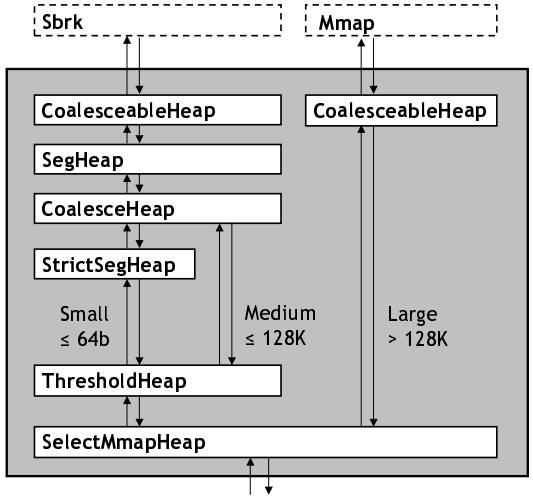


Figure 7: A diagram of LeaHeap's architecture.

sufficiently large object is requested), frees all of the objects held in the small heap. We implemented coalescing and splitting using two layers. CoalesceHeap performs splitting and coalescing, while CoalesceableHeap provides object headers and methods that support coalescing. SegHeap is a more general version of StrictSegHeap described in Section 6.1 that searches through all of its heaps for available memory. Not shown in the picture are AdaptHeap and DLList. AdaptHeap lets us embed a dictionary data structure within freed objects, and for LeaHeap, we use DLList, which implements a FIFO doubly-linked list. While LeaHeap is not a complete implementation of the Lea allocator (which includes other heuristics to further reduce fragmentation), it is a faithful model that implements most of its important features, including the hierarchy described here.

We built LeaHeap in a total of three weeks. We were able to reuse a number of layers, including SbrkHeap, MmapHeap, and SegHeap. The layers that implement coalescing (CoalesceHeap and CoalesceableHeap) are especially useful and can be reused to build other coalescing allocators, as we show in Section 8. The new layers constitute around 500 lines of code, not counting comments or white space, while the Lea allocator is over 2,000 lines of code. LeaHeap is also more flexible than the original Lea allocator. For instance, a programmer can use multiple instances of LeaHeaps to manage distinct ranges of memory, something that is not possible with the original. Similarly, we can make these heaps thread-safe when needed by wrapping them with a LockedHeap layer. Because of this flexibility of heap layers, we can easily include *both* a thread-safe and non-thread-safe version of the same allocator in the same application.

6.3 Experimental Results

We ran the benchmarks in Table 3 with the Kingsley allocator, KingsleyHeap, KingsleyHeap plus coalescing (which we discuss in Section 8), the Lea allocator, and LeaHeap. In Figure 6(a) we present a comparison of the runtimes of our benchmark applications normalized to the original Lea allocator (we present the data used for this graph in Table 5). The average increase in runtime for KingsleyHeap over the Kingsley allocator is just below 2%. For the two extremely allocation-intensive benchmarks, *cfrac* and *roboop*, the increase in runtime is just over 3%, demonstrating that the overhead of heap layers has minimal impact. Despite being cleanly decomposed into a number of layers, KingsleyHeap performs nearly as well as the original hand-coded Kingsley allocator. Runtime of LeaHeap is between $1/2$% faster and 20% slower than the Lea allocator (an average of 7% slower).

Figure 6(b) shows memory consumption for the same benchmarks normalized to the Lea allocator (we present the data used for this graph in Table 6). We define memory consumption as the high-water mark of memory requested from the operating system. For the Kingsley and Lea allocators, we used the amount reported by these programs; for the heap layers allocators, we directly measured the amount requested by both SbrkHeap and MmapHeap. KingsleyHeap's memory consumption is between 54% less and 11% more (on average 5.5% less), while LeaHeap's memory consumption is between 44% less and 19% more (on average 2% less) than the Lea allocator. The outlier is *roboop*, which has an extremely small footprint (just 16K) that exaggerates the memory efficiency of the heap layers allocators. Excluding *roboop*, the average increase in memory consumption for KingsleyHeap is 4% and for LeaHeap is 6.5%.

This investigation provides several insights. First, we have demonstrated that the heap layers framework is sufficiently robust that

Runtime for General-Purpose Allocators					
Benchmark	Kingsley	KingsleyHeap	KingsleyHeap + coal.	Lea	LeaHeap
cfrac	19.02	19.75	25.94	19.09	20.14
espresso	40.66	40.91	44.56	41.12	46.33
lindsay	3.05	3.04	3.16	3.01	3.03
LRUsim	836.67	827.10	826.44	831.98	828.36
perl	66.94	70.01	73.61	66.32	68.60
roboop	10.81	11.19	17.89	10.89	13.08

Table 5: Runtime (in seconds) for the general-purpose allocators described in this paper.

Memory Consumption for General-Purpose Allocators					
Benchmark	Kingsley	KingsleyHeap	KingsleyHeap + coal.	Lea	LeaHeap
cfrac	270,336	280,640	271,944	208,896	241,272
espresso	974,848	992,032	541,696	462,848	448,808
lindsay	2,158,592	2,120,752	1,510,688	1,515,520	1,506,720
LRUsim	2,555,904	2,832,272	1,887,512	1,585,152	1,887,440
perl	425,984	454,024	342,344	331,776	337,408
roboop	45,056	20,760	11,440	20,480	11,616

Table 6: Memory consumption (in bytes) for the general-purpose allocators described in this paper.

even quite sophisticated allocator implementations can be developed using it. Furthermore, we have shown that we can quickly (in a matter of weeks) assemble an allocator that is structurally similar to one of the best general-purpose allocators available. In addition, while we have spent little time tuning our current implementation, its performance and fragmentation are comparable to the original allocator.

7. Software Engineering Benefits

Our experience with building and using heap layers has been quite positive. Some of the software engineering advantages of using mixins to build software layers (e.g., heap layers) have been discussed previously, especially focusing on ease of refinement [5, 9, 21]. We found that using heap layers as a means of stepwise refinement greatly simplified allocator construction. We also found the following additional benefits of using layers.

Because we can generally use any single layer to replace an allocator, we are often able to test and debug layers in isolation, making building allocators a much more reliable process. By adding and removing layers, we can find buggy layers by process of elimination. To further assist in layer debugging, we built a simple DebugHeap layer (shown in Figure 8) that checks for a variety of memory allocation errors, including invalid and multiple `frees`. During development, we insert this layer between pairs of layers as a sanity check. DebugHeap is also useful as a layer for finding errors in client applications. By using it with our heap layers allocators, we discovered a number of serious allocation errors (multiple `frees`) in *p2c*, a program we had previously planned to use as a benchmark.

The additional error-checking that heap layers enable, combined with compiler elimination of layer overhead, encourage the division of allocators into many layers. When porting our first version of the LeaHeap to Solaris, we found that one of our layers, CoalesceSegHeap, contained a bug. This heap layer provided the functionality of SegHeap as well as coalescing, splitting and adding headers to allocated objects. By breaking out coalescing and header management into different layers (CoalesceHeap and CoalesceableHeap) and interposing DebugHeap, we were able to find the bug quickly. The new layers had the additional benefit of allowing us to apply coalescing to other heaps, as we do in the next section.

8. Heap Layers as an Experimental Infrastructure

Because heap layers simplify the creation of memory allocators, we can use them to perform a wide range of memory allocation experiments that previously would have required a substantial programming effort. In this section, we describe one such experiment that demonstrates the use of heap layers as an experimental infrastructure.

As Figures 6(a) and 6(b) demonstrate, the Kingsley allocator is fast but suffers from excessive memory consumption. Wilson and Johnstone attribute this effect to the Kingsley allocator's lack of coalescing or splitting that precludes reuse of objects for different-sized requests [15]. A natural question is to what extent adding coalescing remedies this problem and what impact it has on performance. Using heap layers, we just add coalescing and splitting with the layers we developed for LeaHeap.

We ran our benchmarks with this coalescing Kingsley heap and report runtime and performance numbers in Figures 6(a) and 6(b) as "KingsleyHeap + coal." Coalescing has a dramatic effect on memory consumption, bringing KingsleyHeap fairly close to the Lea allocator. Coalescing decreases memory consumption by an average of 50% (as little as 3% and as much as 80%). For most of the programs, the added cost of coalescing has little impact, but on the extremely allocation-intensive benchmarks (*cfrac* and *roboop*), this cost is significant. This experiment demonstrates that coalescing achieves effective memory utilization, even for an allocator with high internal fragmentation. It also shows that the performance impact of immediate coalescing is significant for allocation-intensive programs, in contrast to the Lea allocator which defers coalescing to certain circumstances, as described in Section 2.

9. Conclusion and Future Work

Dynamic memory management continues to be a critical part of many important applications for which performance is crucial. Programmers, in an effort to avoid the overhead of general-purpose allocation algorithms, write their own custom allocation implementations in an effort to increase performance further. Because both general-purpose and special-purpose allocators are monolithic in design, very little code reuse occurs between implementations of either kind of allocator.

In this paper, we describe a framework in which custom and general purpose allocators can be effectively constructed from compos-

```
template <class SuperHeap>
class DebugHeap : public SuperHeap {
private:
  // A freed object has a special (invalid) size.
  enum { FREED = -1 };
  // "Error messages", used in asserts.
  enum { MALLOC_RETURNED_ALLOCATED_OBJECT = 0,
         FREE_CALLED_ON_INVALID_OBJECT = 0,
         FREE_CALLED_TWICE_ON_SAME_OBJECT = 0 };
public:
  inline void * malloc (size_t sz) {
    void * ptr = SuperHeap::malloc (sz);
    if (ptr == NULL)
      return NULL;
    // Fill the space with a known value.
    memset (ptr, 'A', sz);
    mapType::iterator i = allocated.find (ptr);
    if (i == allocated.end()) {
      allocated.insert (pair<void *, int>(ptr, sz));
    } else {
      if ((*i).second != FREED) {
        assert (MALLOC_RETURNED_ALLOCATED_OBJECT);
      } else {
       (*i).second = sz;
      }
    }
    return ptr;
  }
  inline void free (void * ptr) {
    mapType::iterator i = allocated.find (ptr);
    if (i == allocated.end()) {
      assert (FREE_CALLED_ON_INVALID_OBJECT);
      return;
    }
    if ((*i).second == FREED) {
      assert (FREE_CALLED_TWICE_ON_SAME_OBJECT);
      return;
    }
    // Fill the space with a known value.
    memset (ptr, 'F', (*i).second);
    (*i).second = FREED;
    SuperHeap::free (ptr);
  }
private:
  typedef map<void *, int> mapType;
  // A map of tuples (obj address, size).
  mapType allocated;
};
```

Figure 8: The implementation of DebugHeap.

able, reusable parts. Our framework, heap layers, uses C++ templates and inheritance to allow high-performance allocators, both general and special purpose, to be rapidly created. Even though heap layers introduce many layers of abstraction into an implementation, building allocators using heap layers can actually match or improve the performance of monolithic allocators. This non-intuitive result occurs, as we show, because heap layers expand the flexibility of compiler-directed inlining.

Based on our design, we have implemented a library of reusable heap layers: layers specifically designed to combine heaps, layers that provide heap utilities such as locking and debugging, and layers that support application-specific semantics such as region allocation and stack-structured allocation. We also demonstrate how these layers can be easily combined to create special and general purpose allocators.

To evaluate the cost of building allocators using heap layers, we present a performance comparison of two custom allocators found in SPEC2000 programs (`197.parser` and `176.gcc`) against an equivalent implementation based on heap layers. In both cases, we show that the use of heap layers improves performance slightly over the original implementation. This demonstrates the surprising result that the software engineering benefits described above have no performance penalty for these programs. We also compare the performance of a general-purpose allocator based on heap layers against the performance of the Lea allocator, widely considered to be among the best uniprocessor allocators available. While the allocator based on heap layers currently requires more CPU time (7% on average), we anticipate that this difference will shrink as we spend more time tuning our implementation. Furthermore, because our implementation is based on layers, we can immediately provide an efficient scalable version of our allocator for multithreaded programs comparable to Hoard [6], whereas the Lea allocator requires significant effort to rewrite for this case.

Our results suggest a number of additional research directions. First, because heap layers are so easy to combine and compose, they provide an excellent infrastructure for doing comparative performance studies. Questions like the cache effect of size tags, or the locality effects of internal or external fragmentation can be studied easily using heap layers. Second, we anticipate growing our library of standard layers to increase the flexibility with which high-performing allocators can be composed. At the same time, we believe that we can build better general-purpose allocators by using heap layers. Finally, we are interested in expanding the application of mixin technology beyond memory allocators. The potential that C++ templates allow abstraction and composition at no performance cost opens up a number of possibilities to redesign and refactor other performance-critical infrastructures.

10. Acknowledgements

Thanks to Michael Parkes for many enjoyable discussions about allocator design and construction and to Per-Åke Larson for pointing out the importance of custom allocators in server applications, to Yannis Smaragdakis for first bringing the mixin technique to our attention, and to Don Batory, Rich Cardone, Doug Lea, Michael VanHilst, and the PLDI reviewers for their helpful comments.

The heap layers infrastructure and accompanying benchmark programs may be downloaded from `http://www.cs.utexas.edu/users/emery/research`.

11. References

[1] Apache Foundation. Apache web server. http://www.apache.org.

[2] G. Attardi and T. Flagella. A customizable memory management framework. In *Proceedings of the USENIX C++ Conference*, Cambridge, Massachussetts, 1994.

[3] Giuseppe Attardi, Tito Flagella, and Pietro Iglio. A customizable memory management framework for C++. In *Software Practice & Experience*, number 28(11), pages 1143–1183. Wiley, 1998.

[4] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 187–196, Albuquerque, New Mexico, June 1993.

[5] Don Batory, Clay Johnson, Bob MacDonald, and Dale von Heeder. Achieving extensibility through product-lines and domain-specific languages: A case study. In *Proceedings of the International Conference on Software Reuse*, Vienna, Austria, 2000.

[6] Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. In *International Conference on*

Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX), pages 117–128, Cambridge, MA, November 2000.

[7] Gilad Bracha and William Cook. Mixin-based inheritance. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming: Systems, Languages, and Applications (OOPSLA) / Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 303–311, Ottawa, Canada, 1990. ACM Press.

[8] Dov Bulka and David Mayhew. *Efficient C++*. Addison-Wesley, 2001.

[9] Richard Cardone and Calvin Lin. Comparing frameworks and layered refinement. In *Proceedings of the 23rd International Conference on Software Engineering (ICSE)*, May 2001.

[10] Trishul Chilimbi. Private communication. May 2000.

[11] David Gay and Alex Aiken. Memory management with explicit regions. In *Proceedings of the 1998 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 313 – 323, Montreal, Canada, June 1998.

[12] Wolfram Gloger. Dynamic memory allocator implementations in linux system libraries. http://www.dent.med.uni-muenchen.de/˜ wmglo/malloc-slides.html.

[13] Dirk Grunwald and Benjamin Zorn. CustoMalloc: Efficient synthesized memory allocators. In *Software Practice & Experience*, number 23(8), pages 851–869. Wiley, August 1993.

[14] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *Proceedings of the 1993 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 177–186, New York, NY, June 1993.

[15] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: Solved? In *International Symposium on Memory Management*, Vancouver, B.C., Canada, 1998.

[16] Murali R. Krishnan. Heap: Pleasures and pains. Microsoft Developer Newsletter, February 1999.

[17] Doug Lea. A memory allocator. http://g.oswego.edu/dl/html/malloc.html.

[18] Scott Meyers. *More Effective C++*. Addison-Wesley, 1997.

[19] D. T. Ross. The AED free storage package. *Communications of the ACM*, 10(8):481–492, 1967.

[20] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII)*, pages 12–23, October 1998.

[21] Yannis Smaragdakis and Don Batory. Implementing layered design with mixin layers. In Eric Jul, editor, *Proceedings of the European Conference on Object-Oriented Programming (ECOOP '98)*, pages 550–570, Brussels, Belgium, 1998.

[22] Standard Performance Evaluation Corporation. SPEC2000. http://www.spec.org.

[23] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Information and Computation*, 1997.

[24] Michael VanHilst and David Notkin. Using role components to implement collaboration-based designs. In *Proceedings of OOPSLA 1996*, pages 359–369, October 1996.

[25] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. In *Software Practice & Experience*, number 26, pages 1–18. Wiley, 1996.

[26] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic storage allocation: A survey and critical review. *Lecture Notes in Computer Science*, 986, 1995.