How to Focus on Memory Allocation Strategies

Alin Jula and Lawrence Rauchwerger
alinj@cs.tamu.edu , rwerger@cs.tamu.edu

Technical Report TR07-003
Parasol Lab
Department of Computer Science
Texas A&M University
College Station, TX 77843-3112

June 18, 2007

Abstract

In the first 30 years or so of the dynamic storage allocation history, people focused on allocation speed and memory fragmentation, for these were the dominant operations. While both goals still persist today, in the last decade the ever-increasing memory latency gap has been emerging as another dominant goal. To address this goal, new allocation strategies need to be further explored. Nonetheless, experimenting with new allocation strategies can be a daunting task as programmers lack appropriate support. To address this issue, we present a formalism which generalizes the memory management and allocation problem. This formalism allows programmers to focus on the allocation strategies rather than implementation. We implemented this formalism as a generic library. This library is to memory allocation what the C++ Standard Template Library is to programming. With this library, we build two new allocators which exploit data locality and reduce the overall execution time and the memory usage, when compared to state of the art allocators . Their implementation required several lines of code which stands as testimony to the efficacy of our formalism.

Note: Each bibliography entry has an embedded external `http` link to its content for easy exploration of the bibliography.

1 Motivation

As the memory wall gets taller, memory allocation can no longer afford to treat locality as a side effect. Improving locality should become a first rank citizen for memory allocation, along with speed and fragmentation. While some of the existent allocators exploit temporal locality, spatial locality has been marginalized. Neither temporal nor spatial locality have received the same attention in the memory allocation community as have allocation speed and memory fragmentation[33]. There are other techniques, such as compiler optimizations, software and hardware pre-fetching, which improve locality by reducing the memory access latency[6, 7, 13]. Unlike these techniques, memory allocation solves the problem at its source: data placement.

Nevertheless, improving locality through memory allocation is a hard problem. Unlike compilers where the locality improving phase happens off-line and thus its costs are less demanding, memory allocators have the daunting task of allocating fast, minimizing waste *and* improving locality, all at the same time and in an on-line scenario. However, the reduction in execution time provided by a locality improving allocator must outweigh the effort spent in the process. For example, allocating two blocks which are accessed together on different pages can yield a latency of a thousand cycles. These cycles can be spent by the memory allocator to allocate the blocks in the same page. But, if the allocator spends more than a thousand cycles, the cost of improving the locality outweighs its benefits and the locality improving is not profitable anymore. Having to account not only for the output but also for the process makes locality improving a hard problem.

It is thus paramount to have tools which assist in the exploration of these strategies. This article presents such tools, in a theoretical formalism and a generic software library named Allotheque, which allow users to focus on the allocation strategies. Allotheque is to memory allocation what the C++ Standard Template Library is to programming: it allows the users to focus on the allocation strategies rather than the implementation. We used Allotheque to develop two new allocators whose performance competes or surpasses the performance of the Lea allocator, one of the best overall memory allocators[22, 5]. Their implementation is less 50 lines of code. The code conciseness stands as testimony to Allotheque's effectiveness to develop new allocators. For example, the data structure of Kingsley's

```
1.typedef hash_table<size,list<void>,match_only, power2_segregation>   Kingsley;
2.typedef rb_tree<kbit<K>,list<void>,match> tk;
3.typedef hash_table<size,tk,match_next,multiple8_segregation > Defero;
```

Figure 1: Kingsley's and Defero's memory management schemes

allocator used in BSD 4.3[34, 23] can be implemented in a single line of code, Fig 1 line 1, while the data structure of our previous locality improving allocator, Defero[18], can be implemented in two lines of code, Fig 1, line 2-3.

This paper makes two contributions:

A formalism accompanied by its software library , named Allotheque, which allows users to focus on the allocation strategies rather then their implementation. The formalism presents a 3-dimensional space for describing a memory allocation problem. Its three independent dimensions are (1)the chunk's attributes, (2) the memory partition and (3) the allocation predicates.

The formalism's implementation, Allotheque, allows programmers to compose new allocators effectively. We surmise that the number of lines of C++ code required to implement an allocator is approximate to the number of lines of pseudo code used to describe the allocator's mechanism.

Two new allocators which improve program locality and reduce execution time and memory consumption. The two allocators, built with Allotheque, exploit the spatial locality helped by the hints provided to the allocator. We show that these new allocators are not only easy to develop, but they outperform state of the art allocators.

In the next section we describe related work and in section § 3 we introduce the 3-dimensional allocation space. In section§ 4 we explain the design and implementation of Allotheque and in section § 5 we present the algebraic formalism. The two new allocators are discussed in section § 6 and in section § 7 we report on their performance. We then conclude in section§ 8.

2 Related Work

There is a significant amount of work in dynamic memory allocation. Wilson et al. [33] provide an excellent survey which underlines the importance of focusing on allocation strategies rather than implementation when designing a memory allocator. As far as we know, our work is the first which generalizes the memory management and allocation problem. We next split the related work into two classes: 1) memory allocation libraries and 2) locality improvement.

Memory Allocation Libraries

Berger et al. [4] present a useful infrastructure for building memory allocators which uses overhead-free template mix-in technique to create a stack of layers. The programmer can use the provided layers, 20 composition and 3 system, to build new allocators. Their work is the closest to our software tool. While we share the same goal, we differ in the approach we take. We identify only three fundamental dimensions, namely chunk attributes, containers to store the attributes and allocation predicates to search them. This 3-dimensional space along with the theoretical formalism are sufficient to formulate any memory allocation problem. When these dimensions take concrete types, a new allocator mechanism and strategy gets instantiated.

Kiem-Phong Vo [31] introduces the idea of organizing the memory into separate regions, each with a discipline to get raw memory and a method to manage allocation. The author presents several allocation methods such as general purpose allocations, stack-like allocators and aids for memory debugging or profiling. This approach is complementary to ours, as the allocators for each region can be developed independently with Allotheque. The author also approaches the locality improving aspect by preserving the "wilderness", which is the topmost address allocated from the system. This mechanism reduces the fragmentation and indirectly increases data locality by minimizing the amount of used virtual pages, which was also confirmed in the studies performed by Grunwald et al. in [12].

Attardi et at. [2] present a garbage collection framework, Customisable Memory Management (CMM), which allows different heaps to be managed by different strategies. Unlike our work, their focus is on garbage collection and implicit memory management. However, their work is complementary to ours, as new allocation strategies can be integrated with their framework.

Locality Improving

Lattner and Adve [21] developed a compiler technique based on pointer and escape analysis to identify logical data structures in the program. Once a data structure is identified, its elements are allocated in a designated memory pool. Calder et al. [7] present a profile driven compiler technique for data placement (stack, global, heap and constants) in order to reduce data cache misses. Their technique creates a temporal relationship graph which is used in the next runs to arrange highly temporal data in order to reduce the cache conflicts. Huang et al. [16] used the Jikes RVM adaptive compiler to sample the methods used by the program at run-time, and when they become "hot", they are compiled. This approach increases data locality by profiling at garbage time. Shuf et al. [27] use a profile driven garbage collection technique to show that prolific type objects can benefit from allocating them together, since they tend to be related and short-lived. Chilimbi et al. [9] investigate placing adjacent blocks in the same cache line at allocation time. They present ccmorph and ccmalloc. Ccmorph rearranges trees in memory to reduce locality. This method is applicable only for static trees, which are allocated once and do not change over time. Ccmalloc places adjacent blocks in the same cache line, if possible. The strategy used to implement ccmalloc is not provided and the tool is not available[1]. We could not compare them against our work.

Barret and Zorn [3] show that 90% of all bytes allocated are short-lived. They present an algorithm for lifetime prediction for objects. Objects are connected to program call-sites, where they are created. Based on these call-sites, short-lived objects are placed in separate arenas, while long-lived objects are placed together. The call site is an attribute which can be described with our framework. Seidl and Zorn [26] present an algorithm for predicting heap object reference and lifetime behavior at the allocation time. Their profile-based approach considers a variety of different information sources present at the time of objects allocation, such as stack pointer,path point or stack contents, to predict the object's reference frequency and lifetime.

Gay and Aiken [11] present the region based memory management in which variables declared in the a syntactic scope are allocated in a contiguous memory region. Upon exiting a scope, its entire region is reclaimed. Region based allocation is a very simple and efficient allocation strategy which provides fast allocation and improves temporal locality. Yet, it is applicable only to variables which do not live longer than their declaration scope. However, Cherem and Rugina [8] extend the region based memory management to Java programs through a compiler transformation which translates the program into an equivalent output program with region-based memory management

There are other memory allocation schemes which attempt to improve locality by various techniques such as instance interleaving [30] or caching and coalescing [22].

3 Generic Memory Management and Allocation

This section presents the generalized memory management and allocation problem. We decompose the problem into three independent components: (i) chunk attributes, (ii) memory partitions to store them and (iii) allocation predicates to search them. We then show how these only three components assemble into any memory allocator.

[1]Personal communication with the author

3.1 Chunk Attributes

Each memory chunk has attributes which describes it, such as its size or the virtual page and the cache line to which the chunk belongs. Any chunk property is an attribute[2]. For example, the program call site used as a lifetime predictor by Barret et at. [3] or the stack pointer used by Seidl et at. [26] can be regarded as attributes.

Attributes are classified into two categories: 1) mandatory, such as size and 2) optional, such as call site. Size is the only mandatory attribute, for it affects the program's correctness. All the other attributes are optional, for they do not affect the program's correctness and are byproducts of the computer's design. They do affect the program's performance though and thus they can be used by the memory allocator to improve performance. In this article we use the optional allocation hints to improve on data locality. In addition to being classified as mandatory and optional, attributes can also be classified into 1) implicit and 2) explicit, based on how they can be inferred. For example, virtual page and cache line can be inferred implicitly from the chunk's address and thus they do not need to be stored in the chunk. Size on the other hand is an explicit attribute which cannot be inferred directly from the chunk, and thus it needs be stored[3]. The implicit and explicit classification is important because it decides whether the attribute gets stored with the chunk or not.

All attributes are instances of a generic and unified attribute concept. The chunks are characterized by a unified attribute which allows for a generic design of the memory management and allocation, as we will see in the next sections.

3.2 Memory Management with Partitions

We now use the set theory to partition the address space which is represented as a set of generic attributes. Partitioning the memory space groups chunks with the same attributes together. This grouping allows us to search for specific attributes. For example, the segregated lists approach partitions the chunks of the same size in a list. This partition makes the search for a specific size a constant time operation. Our goal here is to create a generic partition of the address space.

We use algebraic equivalence relations to partition the memory into equivalence classes. For example the equivalence relation 'modulo 2' partitions a set into two equivalent classes, each class holding elements of the same parity. Well known memory management approaches, such as the segregated lists, the buddy systems, or the virtual pagination, can also be expressed as equivalence relations. The reason we have chosen equivalence relations to partition the memory is because they have the property of maintaining their organization throughout the entire execution of the program, regardless of the allocation/deallocation pattern. Thus, equivalence partitions maintain their order at all times.

An equivalence relation, denoted by R, partitions the memory address space into equivalence classes, denoted by C_R. All chunks within the same equivalence class have the same attribute. We can now search this partition for a certain attribute, search whose complexity is $\Theta(\log C_R)$, assuming a partial order amongst the equivalence classes. Thus, the more equivalence classes, the longer the search.

The algebraic memory partition was an important design decision because it allows flexible memory

[2]There are attributes that are not relevant to memory allocation, such as the contents of a memory chunk. In this paper we refer only to attributes that are pertinent to the memory allocation problem.

[3]Size can be stored in each chunk as in the sequential-fits [20, 28] or once in a hash table for all chunks in the bucket as in the segregated approaches[10, 25].

management. For example, if we want to partition the managed memory into virtual pages, we select the address attribute for our chunks and use an equivalence relation R_{page} which creates an equivalence class for all chunks within a certain page. Thus, only the equivalence relation needs to be used to create a page aware memory partition, while the rest of the implementation remains unchanged. Next we describe how the memory management stores its partitions.

3.2.1 Storing Chunks in Containers

The equivalence classes are stored in a class container, denoted by *C-container*. A C-container should favor fast search operations designed to search for an equivalence class with a certain attribute. For example, trees and hash tables make good C-containers. The search requires the attribute to offer a partial order. For this reason, we require that each attribute be represented as an integral type, so that a partial order exists.

Each equivalence class has a collection of equivalent elements. All equivalent elements in the same class are stored in an element container, denoted by *E-container*. An E-container should favor insertion and deletion, operations which are expected to be dominant. For example, lists make good E-container. These guidelines should be used when one decides which C-container and E-container to use in the implementation of an allocator.

We now describe how to partition the space based on more than one attribute. Having the space partitioned based on multiple attributes allows for multi-attribute requests. For example, the space can be partitioned based on size and address. This partition allows for locating a memory chunk of a certain size and at a certain address. The returned memory chunk now satisfies two constraints. We identified two types of multi-attribute partitions: (i) *recursive* partitions where each equivalence class is further partitioned and (ii) *simultaneous* partitions where a chunk belongs to multiple independent partitions at the same time.

3.2.2 Recursive Partitions

In recursive partitioning, each equivalence class is further partitioned using a new attribute. This procedure, applied recursively, creates a chain of partitions, with the path to reach the last partition going through all previous partitions. For example, suppose we have a set of adddresses $S = \{0, 1, \ldots 5\}$ and the equivalence relation 'modulo 3', $R_{\%3}$, which creates three disjoint subsets, S_0, S_1 and S_2 as described below. Each subset can be further partitioned using a different equivalence relation. For example the parity relation $R_{odd-even}$, defined as two elements are equivalent iff they have the same parity, partitions each of the three subsets into two new $R_{\%3} \circ R_{odd-even}$ equivalence classes, $S_{i,j}, 0 \leq i \leq 2, j =' odd',' even'$. We obtain recursive partitions by composing equivalence relations, composition denoted by the symbol $\circ$.

$$S \xrightarrow{R_{\%3}} \begin{cases} S_0 = \{0,3\} \xrightarrow{R_{odd-even}} \begin{cases} S_{0,odd} = \{0\} \\ S_{0,even} = \{3\} \end{cases} \\ S_1 = \{1,4\} \xrightarrow{R_{odd-even}} \begin{cases} S_{1,odd} = \{4\} \\ S_{1,even} = \{1\} \end{cases} \\ S_2 = \{2,5\} \xrightarrow{R_{odd-even}} \begin{cases} S_{2,odd} = \{2\} \\ S_{2,even} = \{5\} \end{cases} \end{cases}$$

The recursive partition is not commutative. That is $R_1 \circ R_2 \neq R_2 \circ R_1$. In sections § 6.1 and § 6.3 we present allocators which compose the same relations, but in a different order, which leads to different allocation strategies.

3.2.3 Simultaneous Partitions

The second type of multi-attribute partition allows different partitions to coexist on the same set and at the same time. We refer to this scheme as *simultaneous* partitions. For example, the two equivalence relations, $R_{\%3}$ and $R_{odd-even}$ described above, divide S into two partitions which can exist simultaneously on the same set S. Every element of S simultaneously belongs to two equivalence classes.

$$\left.\begin{array}{l}\{0,3\} = S_0\\ \{1,4\} = S_1\\ \{2,5\} = S_2\end{array}\right\} \overset{R_{\%3}}{\longleftarrow} S \overset{R_{odd-even}}{\longrightarrow} \left\{\begin{array}{r} S_{odd} = \{1,3,5\}\\ S_{even} = \{0,2,4\}\end{array}\right.$$

In contrast to recursive partitioning, which requires the traversal of the recursion chain to reach a certain partition, the simultaneous partition allows direct access to any of the partitions. It allows memory requests based on different attributes. For example one can request an element based on its parity, and the next request can be based on its 'modulo-3'. In section § 3.4 we discuss how the simultaneous partitioning is used to implement splitting and coalescing in a memory allocator. Before that though, we discuss how these partitions are used in dynamic memory allocation.

3.3 Memory Allocation

The allocation process can now be generalized as a request of N attributes and can be formulated as: "return a memory chunk with the following simultaneous constraints: attribute$_i$ has value $attr_i$ for every 1$\leq$ i $\leq$ N". For example, a 3-attribute request might look like: "return a memory chunk of size 8 bytes, in the same virtual page as x, but in different cache set than x". When the partition type is considered furthermore, the allocation process shows different characteristics. In recursive partitioning the allocation process becomes a *prioritized* multi-attribute request. In the example presented in § 3.2.2, we searched for an integer whose modulo-3 was 2 and parity odd. We first selected S_2 and then selected $S_{2,odd}$. The second search took place in the S_2 subset and not the whole set S. Thus, the search was prioritized based on 'modulo-3' attribute. With three attributes, a prioritized allocation, of say (a_1, a_2, a_3), and with a partition $R_1 \circ R_2 \circ R_3$, first finds all chunks with the attribute a_1. Within this set of attributes, it then finds all chunks with the attribute a_2. Within this even more restricted set, it then finds the chunks with the attribute a_3. The same allocation request, (a_1, a_2, a_3), but with a different recursive partition $R_2 \circ R_3 \circ R_1$ might return different results. Hence, prioritized allocation, just like the recursive partition, is not commutative. We denote the prioritized relation with the symbol $\succ$, where $a \succ b$ means that a takes priority before b.

As for the simultaneous partition, the allocation process allows only one partition to be considered at one time. The search for the chunk's attribute takes place within the selected partition only. However, once the chunk is found in one partition, it is taken out of all the partitions. The strength of simultaneous partition is that it allows different attributes to be considered for different allocations. The deallocation process returns a chunk back into its equivalence classes, in both recursive and simultaneous partitions. The deallocation request must contain all the attributes of that chunk.

Fig. 2 shows the generic allocation and deallocation as well as the traditional ones. Traditional

```
// Traditional allocation & deallocation requests
1. int *x1 = new int;    ...  delete x1;
2. void *x2 = malloc(8); ...free(x2);

// Generalized allocation & deallocation requests
3. int *x3 = allocate(Request<char*,int>(0,8));
4. deallocate(Request<char*,int>(x3,8));
```

Figure 2: Multi-attribute memory requests

allocation consider 'size' as the only attribute in a request[4], lines 1-2, Fig. 2. C++ standard extends this interface and allows the passing of an address pointer in its STL allocator interface[1]. The generic allocation considers multiple attributes in a request. In line 3-4, Fig. 2, the allocation considers two attributes, size and address.

The request implementation provides means of accessing its attributes based on type. In turn, this allows partitions to extract their attribute from the request in order to search for that specific attribute. This scheme decouples the containers from the generalized allocation. Section § 4.4 discusses this mechanism in more details.

3.3.1 Allocation Predicates

An allocation requests a chunk with specific attributes. We identified four components which create an allocation:

- the *search* traversal, such as linear search for lists and top-down search for trees
- the *start* position of the search traversal, such as the beginning or the middle of a list
- the *target* or the requested attribute
- the termination *predicate*, which decides when the search traversal stops

The first component, the search traversal, is dictated by the type of C-container, while the remaining three components - start, target and the terminate predicate - are independent of the container type. We grouped the last three components together in what we denote by *allocation predicate*. This allocation predicate tells the container where to start the search from, what to look for and when to stop looking for it. This is very similar to the STL $for\_each(begin, end, bind1st(binary\_predicate, target))$ algorithm[24, 1]. The search can start either from the beginning of the C-container or from the last visited element, which can be cached by the C-container. This decision is encapsulated in the allocation predicate, thus decoupling the search from the starting position. Table 1 shows the most common search traversals in the common containers. Table 2 shows the most common terminate predicates, such as equal or max, termination predicates which are used in the allocation predicates. When the search traversal and allocation predicate are put together, an allocation request is created. Note the decoupling between the traversal pattern and the termination condition.

[4]'malloc' requires the size explicitly, while 'new' requires a type, whose size is implicitly known

Container	Search Traversal	Description
List	linear	Beging to end
Tree	top-down	Top to bottom
Hash	lookup, linear	Try 'lookup', then start to end

Table 1: Search Traversals for Allocation.

Predicate	Description
true	return true
equal	if (attr == target) true else false
greater	if (attr > target) true else false
max	if ((attr - target)> max_diff)
	max = attr ; max_diff=attr-target
min	if ((attr - target)< min_diff)
	min = attr; min_diff = attr - target

Table 2: Termination Predicates

Most of the common allocation strategies in the literature can be described with this generic allocation predicate. Table 3 shows the most common strategies. For example, the 'first-fit' strategy is a linear traversal of a list from the beginning to end, together with the termination predicate of 'greater'. This search stops at the first chunk whose attribute is greater than the target. Or consider the 'worst-fit' strategy which selects the largest size in the list. This strategy can be constructed with a linear search and the 'max' terminate predicate. As an example of a search which does not start from the beginning of the container, consider the 'next-fit' strategy. This strategy is similar to 'first-fit' but it starts from where the last search left off. This can easily be implemented using an allocation predicate which signals the container to stores the last visited element and start the subsequent search from this element. More details about how this is implemented in section § 4.2.

Changing the search strategy is equivalent to replacing an allocation predicate. The modular separation between the containers and allocation strategies allows for experimenting with various schemes at the coding price of a *single* line of code. For example, the best-fit strategy "list<size, best-fit>" can be changed into a worst-fit strategy "list<size, worst-fit>" simply by replacing the allocation predicate.

Predicate	Components	Description
any	(begin-end)(true)	First.
first-fit	(begin-end)(greater)	First greater than target
next-fit	(last-end)(greater)	First greater than target
best-fit	(begin-end)(min)	The largest smaller fit
worst-fit	(begin-end)(max)	The largest chunk
match	(begin-end)(equal)	First equal to the target

Table 3: Allocation predicates

3.4 Splitting & Coalescing

We now talk about splitting and coalescing in light of simultaneous partition and allocation predicates. We show how adding these operations to an allocator is actually nothing more than adding another partition to the existent one.

Splitting and coalescing are common operations in memory allocation used in several allocation policies[20, 22, 33]. *Splitting* takes a large chunk and splits into into two smaller chunks: one of them is returned to the application and the other one is inserted back into the allocator's storage. *Coalescing* takes two adjacent chunks and merges them into one contiguous chunk. The idea of splitting and coalescing is to reduce the memory waste and the internal fragmentation. We will not discuss the benefits and strategies of splitting and coalescing as they have been well researched and documented[12, 20, 33]. We will discuss however how we represent them using the simultaneous partitioning scheme.

Consider the following equivalence relation $R_{adjacency}$: two chunks are equivalent if and only if they are adjacent. Two chunks are adjacent to each other if the address of one is equal to the address of the other one plus its size. There are two attributes involved in this relation: size and address. This relation $R_{adjacency}$ partitions the memory into contiguous chunks. A chunk's equivalence class is represented by either the chunk before it, in address order, or the chunk after it[4]. In this $R_{adjacency}$ partition, an allocation corresponds to splitting and a deallocation to coalescing. An allocation with the allocation predicate 'large-enough-to-split', implemented similarly to the ones described in table 2, finds a chunk large enough to both satisfy the request and insert the remainder back in the partition. A deallocation inserts the chunk back into its equivalence class. The insertion is a merge with the equivalence class's chunk. Thus both splitting and coalescing can be represented as allocation and deallocation in the partition created by $R_{adjacency}$ with 'large-enough-to-split' as the allocation predicate. They can be implemented in several ways, e.g. using either a bit flip as in binary buddy systems[19] or using boundary tags as described by Knuth in[20].

3.5 How to Build an Allocator with Allotheque

In this section we build the Kingsley allocator using Allotheque. The implementation requires a number of 18 lines of code, which is approximately the number of pseudo code lines necessary to describe this allocator. This shortness stands as testimony to the expressiveness of our generalized framework. This implementation is presented in Fig 3. The simple segregated storage mechanism is implemented in 2 lines of code, line 2-3. Line 2 declares a hash table with lists as buckets and with power-of-2 as the hash function. The actual hash vector is declared in line 3. This scheme was first published by Purdom, Stigler and Cheam in 1970 as a mix between the buddy system and first-fit[25]. Originally published with the name of 'segregated storage', this scheme was later referred to as 'simple segregated storage' to distinguish it from other segregated storage mechanisms[33]. It was later developed and integrated in BSD 4.2 by Chris Kingsley[34, 23]. The allocator groups all chunks whose size fall within powers of 2 in a single list. The allocator rounds every request to the nearest power of 2 and allocates the first chunk from the corresponding list. The Kingsley allocator does not perform *splitting* or *coalescing*. When the list corresponding to a certain range is emptied, more memory is allocated from the system. This is one of the fastest allocators available, although one of the worst in terms of fragmentation[17].

[4] or both when applied recursively

```
1. template<class SysAlloc>class KingsleyAlloc{

2. typedef hash_table<size, list<void>,match_only, hash_segregated_pow2> mem_partition;
3. static char* hash_tbl[32];

4. template<class Request,class AllocPred>
5. static char* allocate(Request& attrs){
6. round_up_size(attrs);
7. char* r= mem_partition::allocate<Request,AllocPred>(attrs,hash_tbl);
8. if (r==ErrorAttributeMissing) {
9.       size_t size=(size_t) reinterpret_cast<Size>(attrs);
10.      char* nc=SysAlloc::allocate(4096);
11.      for(int i=0;i<4096;i=i+size)
12.        deallocate(Attributes(nc+i,size));}
13.      r= mem_partition::allocate <Request,AllocPred>(attrs,hash_tbl);}
14.if (!ValidAddress(r)) throw Exception();
15.return r; }

16.template<class Request>
17.static void deallocate( Request& attrs) {
18.mem_partition::deallocate<Request>(attrs);}};
```

Figure 3: Kingsley allocator implementation in Allotheque

The allocation is implemented in 12 lines of code, lines 4-15, and the deallocation in 3 lines of code, line 16-18. In the allocation process, with the size rounded up to the nearest power of 2, line 6, the hash table searches for a chunk with the requested size attribute, line 7. This line 7 contains the recursive search algorithm described in section § 3.3. If there are no chunks with that attribute, line 8, the allocator asks for more memory from the SysAlloc class, which is in charge with transacting memory with the operating system, line 9-10. The new memory is inserted into the allocator's hash table structure, line 11-12, and the original search is repeated, line 13. If a chunk is not found, even after acquiring more memory from the system, the allocator terminates gracefully with an exception, line 14.

4 Design & Implementation

In this section we describe the design and implementation of our three components, namely the chunk attributes, the memory partition and the allocation predicate. We then describe how they interact with each other and show an example of how everything fits together. Allotheque is implemented using C++ templates and all of the class methods are static so there is no memory overhead for the instantiated objects.

4.1 Containers

Each container keeps its free chunks organized based on a certain attribute. The container has four

```
1. template<class Attribute, class AllocPred, class NestedContainer, int NestContOffset=0>
2. class container{
...
3. template<class Request, class NewAllocPred=AllocPred>
4. static char* allocate(char** header, Request& attrs)};

5. template<class Request>
6. static void deallocate(char** header, Request& attrs)
```

Figure 4: Container's interface

parameters which are represented as template parameters: (i) chunk's attribute, (ii) default allocation predicate, (iii) nested container type, and (iv) the offset within a chunk where the nested container's data structure starts. The C-container searches its equivalence classes, using the default allocation predicate or the one provided in the allocate method, line 3, Fig. 4. When the equivalence class is found, the allocation is called recursively on the nested container responsible for that equivalence class. Since chunks can belong to two nested containers, the NextContOffset offset, line 1, carries the displacement from the beginning of the chunk where the data structure of the nested container is stored. The chunk implementation is described in more detail in the next section.

We implemented five containers in our framework: singly linked lists, doubly linked lists, a generic hash table and several hash functions, a red-black tree and the identity container which ends the recursion. Section § 6 uses them to build new allocators. This container collection is by no means comprehensive, but rather forms the basic platform for building new allocators. For example, Allotheque does not currently have an implementation for cartesian tree, container which is organized based on two attributes, one for the height and one for the width [29].

Recursive Partition is implemented using C++ nested templates. Suppose we have three recursively nested containers, each with a different atrribute, $C_0 < Attribute_0, C_1 < Attribute_1, C_2 < Attribute_2, identity >>>$. The first container is a C-container. The middle container is both a C-container for its own partition and an E-containers for its parent's partition respectively. The third is an E-container.

Each **Simultaneous Partition** is implemented separately within each chunk, which stores all the partitions' data structures. For three simultaneous partitions we have the following partition declaration:
$C_0 < Attribute_0, E_0, AP_0, 0 >$ *partition A*
$C_1 < Attribute_1, E_1, AP_1, size_1 >$ *partition B*
$C_2 < Attribute_2, E_1, AP_2, size_0 + size_1 >$ *partition C*

4.2 Chunks

The chunk's structure for recursive and simultaneous partitioning is depicted in Fig. 4.2. For recursive partitioning, Fig. 4.2 top, the first item stored in a chunk is the container's structure. This can be a 'next' pointer for the linked lists or 'left' and 'right' pointers for the binary trees. The second item stored in a chunk is the attribute, if explicit. Otherwise the implicit attributes need not be stored since they can be inferred directly. The third item stored in a chunk is a pointer to the next container in

the partition recursion, if any. And finally, the fourth item a chunk stores is a pointer to the nested container's last allocated from equivalence class, if the allocation predicate informs the container to do so. The recursive partition requires a minimum chunk size of $min_{size} = max\{P_i, 0 \leq i \leq N\}$, where P_i

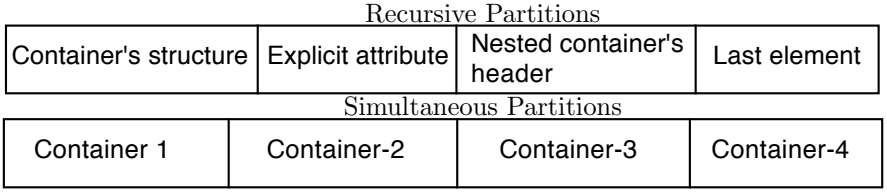


Figure 5: Chunk's structure for recursive and simultaneous partitions

is the size required to store the chunk in the i^{th} container. Thus, containers that have a high memory overhead, such as hash tables, are not appropriate for use in a recursive partition, since their overhead is propagated to every chunk. Except when the container is the first in the recursion chain, in which case the overhead appears only once.

As for the simultaneous multi-partitions, Fig. 4.2 bottom, each chunk carries the internal structure of every partition. Hence, the number of simultaneous partitions is limited by the size of the chunk. For example, if a partition requires two pointers, the number of partitions for chunks of size 16 bytes is limited to two. The minimum chunk's size to represent N simultaneous multi-partitions is: $min_{size} = \sum_{i=0}^{N} size_i$, where $size_i$ is the size required to store the i^{th} container's internal structure. This allows each partition's overhead to be paid only once, not for every nested equivalence class as in the recursive partition. It also makes the simultaneous multi-partition more desirable for larger chunks and less desirable for smaller ones.

4.3 Attributes

Each attribute is designed as a separate class which contains type information about whether the attribute is explicit or implicit and size required to represent this attribute. An attribute class has two operations: (i) read and (ii) write. Fig. 4.3 shows the implementation of 'cache-set' attribute for a 2

```
struct CacheSetAttribute {
  typedef implicit_attribute_tag attribute_tag;
  //! Number of bytes it required
  enum {Attribute_Span = 0};
  //! Read the cache-set of the memory chunk.
  static size_t read(const char* p)
  {return (((size_t)(p) & 0x0003FFC0))>>6;}
  //! Write the cache-set
  static void write(char* p, size_t attr) {}  };
```

Figure 6: Cache Set Attribute

MB L2 cache with 64-byte cache line. The method 'read' extracts cache set index from the address bits

which determine the cache set while 'write' is empty because implicit attributes cannot be written by their definition, see § 3.1.

4.4 Allocation Search and Predicate

The search algorithm for an allocation receives a collection of attributes in the form of a generalized memory request. Fig. 7 shows this generic search algorithm, line 2-8, along with the implementation of the 'match' allocation predicate, line 9-14. A container extracts its attribute from the request and searches for an equivalence class with the same attribute. The extraction is implemented as a type conversion operator, present in each type of request, line 4. This technique allows for decoupling of memory management and memory allocation, which can be developed independent of each other. The

```
1. template<class Attribute,class DefaultAP,class NestedAllocator> class container {
...
2. template<class Request, class AllocPred=DefaultAP>
3. char* Container<Attribute>::allocate(Request rqt, char** header) {
4. Attribute *req_attr=reinterpret_cast<Attribute*>(rqt);
5. AllocPred ap(req_attr);
6. start= begin(AllocPred::state_tag());
7. while (ap.check(start)) ++start;
8. return NestedAllocator:: template allocate<Request>(rqt,start);}
//-------------------------------
9.  struct MatchAllocPred{
10. typedef stateless_tag state_tag;
11. char* target;
12. static int check(char*  obj)   {
13.    return  obj==target ?  0 : 1 ; }
14.  Match(char* t): target(t) {} };
```

Figure 7: Generalized Allocation Algorithm and Allocation Predicate

search algorithm starts with the beginning of the container, unless the allocation predicate signals that the search should start with the cached equivalence class, line 6. The search algorithm iterates the container until either reaches the end of it or the allocation predicate signals that the requested attribute was found, line 7. The allocation search is recursively called on the equivalence class found until the recursive hits the identity container, which simply returns the chunk found. The allocation predicate is implemented as a separate class. Lines 9-14 in Fig. 7 show the implementation of 'match' allocation predicate.

As for deallocation, each chunk is returned to its corresponding partitions. Our framework also provides a *range deallocation* operation which deallocates a range of elements with the same attribute at the same time. This operation is very useful for STL containers because when the destructor is invoked or when the container is cleared, all elements of the container are deallocated in bulk. The range deallocation takes advantage of the fact that container's elements are likely to have the same attribute. Thus, instead of deallocating a chunk at the time and traversing the whole series of partitions for each chunk, a range of elements can be deallocated at one time, if they have the same attribute. The range

deallocation is provide with a range of iterators. The range is traversed exactly once and the destructor is invoked for each element.

4.5 System Memory Allocation

Allotheque provides three ways to allocate memory from the operating system: (i) malloc, (ii) mmap and (iii) sbrk & mmap - sbrk as long as possible and mmap thereafter. The latter strategy aims at improving locality and it is used in the Lea allocator[22], and it is also the strategy we used in all of our experiments. Our allocators request one virtual page at a time, which on our system has a size of 4KB. Each allocator can select its own system allocation strategy during its instantiation. A similar approach is provided by HeapLayers [4].

5 Formalism - Unified Theory of Memory Management and Allocation

We now formalize the memory management and allocation problem using equivalence classes, partitions, allocation predicates and attributes. We argue that any memory allocation problem can be formulated using this formalism.

Let $R_1, R_2, \ldots R_n$ be n equivalence relations defined on the memory space $M = i, 0 \leq i \leq 2^{64}$. The partitions created by R_i are stored in a container C_i. Each relation R_i regards a memory chunk by an attribute A_i. For n relations we have $A_1, A_2, \ldots A_n$ attributes which need not be unique. For every relation R_i whose partition is stored in container C_i there is an allocation predicate which is responsible for the allocation strategy. There are $P_1, P_2, \ldots P_n$ allocation predicates, one corresponding for each partition.

A recursive partition is denoted as $R_{1,2,\ldots,n} = R_1 \circ R_2 \circ \ldots Rn$ and stored in recursive containers $C_1 < C_2 < \ldots C_n < R_n, A_n >> \ldots >$. An allocation request for $(a_1, a_2, \ldots a_n)$ with the allocation predicates $P_1, P_2, \ldots P_n$ returns a chunk $= (ra_1, ra_2, \ldots ra_n)$, with $a_1 \stackrel{P_1}{=} ra_1 \succ a_2 \stackrel{P_2}{=} ra_2 \succ \ldots \succ a_n \stackrel{P_n}{=} ra_n$, where $x \stackrel{P}{=} y$ means that binary predicate P with x and y evaluates to true and $\succ$ denotes a prioritized condition. A simultaneous partition is denoted as $R_{1,2,\ldots,n} = R_1 \wedge R_2 \wedge \ldots Rn$ and stored in independent containers $C_1 < R_1, A_1 > \wedge C_1 < R_2, A_2 > \ldots \wedge C_n < R_n, A_n >$. An allocation for (a_i) with the allocation predicate P_i returns a chunk $=(ra_i)$, with $a_i \stackrel{P_i}{=} ra_i$ for any $0 \leq i \leq n$. A deallocation request returns a chunk $= (a_1, a_2, \ldots a_n)$ back into the corresponding partitions. In recursive partition, the deallocation follows the recursive partition chain in returning a chunk to its partitions. In simultaneous, the deallocation returns the chunks to each partition, in any order.

We implemented this formalism into Allotheque, with which we built several new allocators. Existing allocation strategies, such as sequential fits, segregated fits, buddy systems - binary, double, Fibbonacci - or region based, can also be formulated and implemented using Allotheque.

Focus on Strategies

The goal of Allotheque to allow the focus to be on the *strategies* of memory allocation, rather than its implementation. Wilson et al. argue in their thorough dynamic storage allocation survey that "higher-level strategic issues are still more important, but have not given much attention"[33]. Allotheque allows experimentation with new dynamic memory allocation strategies in a very short amount of time. We

surmise that the number of lines of code required to build a new allocator is approximately the same as the number of lines of pseudo-code required to describe the allocator's strategy.

6 Allocation Strategies to Improve Locality

In this section we present two new allocators and another allocator which extends on our previous work[18]. These allocators use various attributes such as virtual page, K-bit and SK-bit, which we will describe shortly. While all the allocators we present have different policies and mechanisms, they share a unifying theme: *to improve locality*. Their implementation does not exceed 50 lines of code, which is as compact as their pseudo code description. We report on their performance in the next section.

Existing allocators can also be implemented using our library in a few lines of code. In § 3.5 we implemented Kingsley's allocator in 18 lines of code. The QuickFit allocator, described by Weinstock in [32], can easily be derived from Kingsley's implementation, by modifying 2 lines of code which contain the size segregation and allocation predicates. While Kingsley's allocator uses the power of 2 size segregation, QuickFit uses a generic size segregation, originally up to 32, which improves on the internal fragmentation, which Kingsley's suffers from. Just like the Kingskey allocator, QuickFit does not coalesce or split chunks, but anticipates that a allocated size will be requested in the future. We use the same segregated approach in all of our allocators. We also use a similar size segregation, namely multiple of 8 bytes, up to 128 bytes. We implemented this size segregation as an equivalence relation, $R_{segregated\ size}$, which partitions the space into 17 groups, 16 size class for sizes less than 128 bytes, grouped 8 bytes apart, and one group for all sizes greater than 128 bytes. This size segregation, also used by Lea allocator - up to 512 bytes - has been shown to work well in comparison with other allocation mechanisms by Johnstone et at. [17].

6.1 Velox

We explore an *allocation strategy which prioritizes location over speed*. We use a recursive partition which first partitions the space based on address and then on size. We name this allocator Velox[5]. Its strategy is to quickly locate the target vicinity and find a chunk of the requested size in that vicinity.

Partition

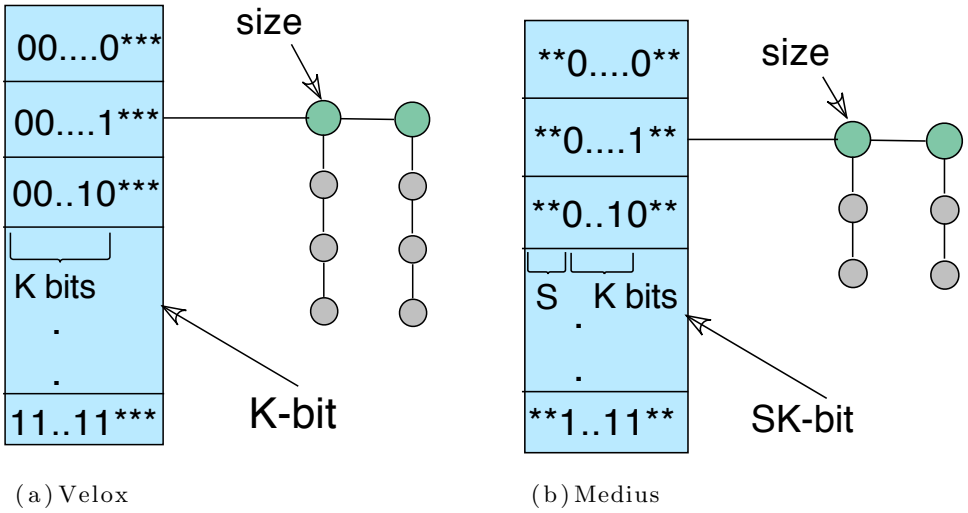
Velox partitions the space using the K-bit relation into K-classes and stores them in a hash table. K-bit is defined as: two memory addresses are equivalent iff their first K higher-order bits are the same. The K-bit partitions the memory in 2^K groups, each with a maximum 2^{32-K} elements. We first used K-bit to implement the Defero allocator, a locality improving allocator[18]. A hash bucket stores all the addresses with the same most significant K bits. The hash table ensures fast access to any K-class. Each K-class gets farther partitioned based on the size attribute. We use the simple size relation, R_{size}, to partition each K-class. The size classes are stored as a list . Each size class contains all the chunks with that corresponding size, which are also stored in a list attached, to the size class node. So essentially we have a hash of lists of lists.

Notice that within a K-class, we have an exact size segregation which is stored in a list rather than a traditional hash table. The list was chosen over the hash table because the latter imposes a memory overhead for each K-class, which would increases the memory fragmentation. Velox relies of the fact

[5] in Latin *velox* means "quick"

the number of sizes within a K-class is rather small, which makes the search in the list sufficiently fast. Fig 8(a) depicts this structure and shows its implementation, lines 1-2. This recursive memory partition can be formally described as $R_{Velox} = R_{K-bit} \circ R_{size}$.

```
1. typedef list<size,list<void>,match> list_size;
2. hash_table<k_bit,list_size,match_first,hf_identity_sparse> velox_partition;
3. hash_table<ks_bit,list_size,match_last_first,hf_identity> medius_partition;
```



(a) Velox (b) Medius

Figure 8: Velox's and Medius' partitions - Implementation and Structure

Allocation Strategy: Prioritize Location

An allocation request for Velox contains a size and a target address. The allocation finds the target's K-class and allocates from within that vicinity. This strategy increases spatial data locality. Within the target's K-class, the size list is searched for the right size. When this size class is found, an address from its list is returned. The returned chunk is guaranteed to be within 2^{32-K} distance from the target, denoted as K-close, and to be within the requested size class. If there are no chunks available in the target's K-class, either because the K-class is empty or the right size is not found, then there are two options: (i) find a bigger chunk and split it within the same K-class or (ii) find a chunk of the right size in a different K-class. We explored the second option. If the right size was not found within a K-class, the last used K-class is selected. This strategy exploits temporal locality. If the last K-class fails too, lowest address available is selected, enforcing the wilderness preservation[31]. This process iterates the available K-classes, starting with the first K-class, until the right sized is found. This strategy is implemented using the 'match_last_first' allocation predicate, Fig 8,line 2. If one wants to bypass the last used K-class, they can use the 'match_first' allocation predicate instead.

The problem with this approach is that programs use only certain regions in the address space, which makes the hash table storing the K-classes in Velox rather sparse. To overcome this issue, we use a sparse implementation for the hash table, which links the contiguous entries in the linked list. This is implemented as a layer over the hash table, and it is provided to the hash table upon instantiation, see

Allocs	Memory partition (attr.) , storage (cont.) and allocation predicate (ap)								
	Level 1			Level 2			Level 3		
	cont.	attr.	ap	cont.	attr.	ap	cont.	attr.	ap
Velox	hash table	K-bit	match_first	list	Size	match	list	void	any
Medius	hash table	SK-bit	match_last_first	list	Size	match	list	void	any
Defero	hash table	Segregated Size	match_only	tree	K-bit	match, first next	list	void	any

Table 4: New allocators' partitions and allocation predicates.

Fig 8 line 2. This layer provides navigation tools within the hash table, such as 'next' and 'previous' operations, which are also used by the allocation predicates.

Memory overhead and chunk size

Velox trades off a larger memory overhead for a faster address search time. Storing the K-classes in a hash vector requires a fixed overhead of 2^K pointers. For $K = 16$ and 4-byte pointers the number of entries is 64K and the overhead is 256KB. For smaller applications this overhead might be significant. However, today's applications use a large amount of memory. For these applications, the overhead might represent a very small percentage of the overall used memory. For example, consider the Olden benchmarks. Except for 'power' , for which Velox's 256KB hash table represents a memory overhead of 30%, for the rest of the benchmarks this overhead represented between 0.11 % and 2.7%, computed as 256KB out of the maximum memory used by an application, see table 5.

The size of the minimum chunk which can be stored in Velox is 12 bytes. The hash table does not require any chunk size restriction, but the list which stores the size requires 12 bytes, 4 bytes storing the size of the chunk, 4 bytes storing the head of a linked list which has all the chunks of that specific size and 4 bytes storing the 'next' link. Next, we'll explore a strategy which minimizes the size of hash table at the expense of locality.

6.2 Medius

In this section, we increase the locality precision, but we lose the locality guarantee. Velox uses the first K bits of an address to index into a K-class. If the heap grows from low to high in address space, then most of the significant bits in the K bits might be the same[6]. If we disregard the most significant bits of an address we can reduce the hash table size. We name this strategy, Medius[7].

Partition

Medius is similar to Velox, but instead of considering the most significant K bits of an address, it considers the K bits of an address, but starting with the S bit. Thus, only the S,S+1, ...S+K bits are considered. We named this equivalence relation $R_{Medius-SK}$. Two addresses are $R_{Medius-SK}$ equivalent iff their $S^{th} - (S+K)^{th}$ bits are the same. Medius partitions the space into 2^{K-S} equivalence classes, which we name SK-classes. This recursive memory partition can be formally described as $R_{Medius} = R_{SK-bit} \circ R_{size}$. While Velox guarantees that two addresses in the same K-class are K-close, Medius guarantees that two addresses in the same SK-class are either (S+K)-close or they are at

[6]If the heap grows from high to low, then the least significant bits in the K bits might be the same.

[7]in Latin *Medius* means "middle"

least 2^{32-S} *apart* from each other. Thus, Medius risks more to improve locality and reduce overhead. Fig. 8(b) depicts Medius' partition, while Fig. 8 line 3 shows the implementation of Medius' structure in just one line of code.

Allocation Strategy: Prioritize Location with More Precision but No Guarantee

Two addresses from an SK-class are either (S+K)-close or at least 2^{32-S} apart from each other. The former helps us improve locality. The latter does not. Medius hopes that the active memory window managed by the allocator is within a 2^{32-S} range. If this is true, then Medius can help improve locality with high precision by guaranteeing that two addresses within the same SK-class are (S+K)-close. Otherwise, Medius relies on the temporal locality of the application to store the close chunks together. Medius' hash table is no longer sparse and thus we do not use a sparse layer for the hash table anymore, see Fig. 8 line 3.

Memory Overhead

Medius and Velox have both the same memory overhead and the same minimum required chunk size, which is 12 bytes.

6.3 Defero

In this section we extend on our previous work. In [18] we presented Defero, an allocator which prioritizes size over locality. The closeness metric was determined by the most significant K bits, which determines both the locality goodness and the cost of doing so. Fig 9(b) shows Defero's structure and lines 4-5 shows its implementation. Defero's partition can be formally expressed as $R_{Defero} = R_{segregated\ size} \circ R_{K-bit}$. Notice that Defero's partition is Velox's partition commuted, which are fundamentally different. This comes as an example of the non commutative property of recursive partitioning presented in § 3.2.2. In [18] we presented Defero and explored 'First' and 'Match' allocation predicates in the partition created by K-bit. Our library allowed us to add the 'next' allocation predicate for the K-bit partition. Adding 'next' informed the hash table to store an additional pointer for each hash entry, to the last used K-class. This is practically caching the last used K-class for each size class. An allocation visits this pointer first. If the allocation predicates are successful, a tree traversal is avoided, thus increasing the allocation speed. Storing these pointers in the hash table requires minimal overhead. It increased the size of the hash table from 16 entries to 32 entries.

Table 4 summarizes the three new allocators we have presented, alongside Defero with 'next' allocation predicate.

Others

We experimented with other allocation policies and strategies. For example, we tried to store the size of both Medius and Velox in a tree for faster searching, but we observed that the list was faster. The number of size classes within a K or SK class was usually small. We also ran across prioritized allocations that resulted in incorrect allocation. For example, Velox's partition when used with ('match', 'match') couple of allocation predicates attempts to locate a target K-class and find a specific size within that class. When a K-class is 'matched', a right size is searched within this class. If there is no splitting and the right size does not exist in that K-class, the allocator determines that the allocation failed and allocates more memory and then tries again. The second time will fail again since the newly acquired memory will unlikely be inserted in the 'match'-ed K-class. This process raises

```
1. typedef rb_tree<kbit,list<void>,next> tree_k;
2. hash_table<size,tree_k,match_next,segregated>  defero;
```

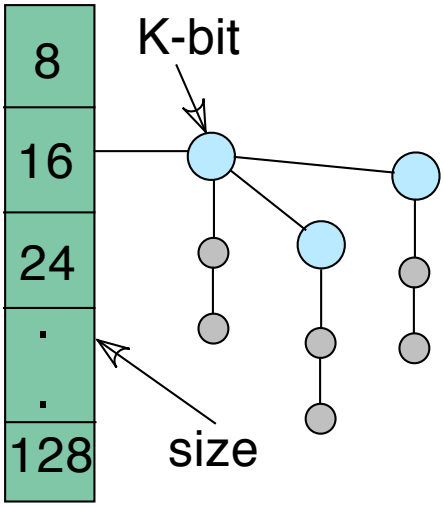


Figure 9: Defero's Partition -Implementation and Structure

Benchmark	Total objects	Max objects in use	Aver. size	Total memory	Max Mem. in use	Total/Max	Input Args
Olden							
bh	188,940	48,638	89	16,821,268	5,597,108	3.0	32,768 1
bisort	2,097,151	2,097,151	12	25,165,812	25,165,812	1.0	3,000,000 1
em3d	560,006	560,006	410	229,600,016	229,600,016	1.0	70,000 100 75 1
health	1,041,607	690,418	13	13,909,444	9,695,176	1.4	5 3,000 1
mst	4,196,352	4,196,352	13	54,525,952	54,525,952	1.0	2,048 1
perimeter	2,204,357	2,204,357	28	61,721,996	61,721,996	1.0	12 1
power	22,401	22,401	38	846,544	846,544	1.0	-
treeadd	4,194,303	4,194,303	12	50,331,636	50,331,636	1.0	22 1
tsp	1,048,575	1,048,575	36	37,748,700	37,748,700	1.0	1,000,000 1
voronoi	1,289,090	524,246	64	83,107,392	34,157,376	2.4	100,000 1
Other							
debruijn	22,394,880	497,664	12	268,738,560	5,971,968	45.0	20736 12 5
md	1,067,490	3,541	20	21,349,800	70,820	301.4	12 2.6 5

Table 5: Benchmarks Statistics.

an exception and interrupts the application. Using a 'match_first' instead for the hash table fixes the bug. Commuting the partitions also fixes the bug. In $R_{segregated\ size} \circ R_{K-bit}$, the first allocation fails the first time, but the second allocation succeeds. Thus, not all composition combinations lead to valid configuration, and one must proceed with care in composing new meaningful allocators.

Benchmark	Velox	Medius	Defero	DL
Olden				
bh	4.80	4.80	0.12	4.64
bisort	1.15	1.15	33.33	33.35
em3d	0.11	0.11	0.25	1.58
health	2.78	2.78	14.28	42.75
mst	0.57	0.57	30.76	30.80
perimeter	0.62	0.62	14.28	14.28
power	32.10	32.10	2.11	20.44
treeadd	0.62	0.62	33.33	33.34
tsp	1.38	1.38	11.55	11.12
voronoi	0.78	0.78	0.02	85.95
Other				
debruijn	4.53	4.53	33.33	33.40
md	5.13	5.13	33.24	33.69
Average				
	4.54	4.54	17.21	28.77

Table 6: Allocators' Fragmentation Percentage

Benchmark	Max. Mem.	Velox	Medius	Defero	DL
	Req.	Max Alloc	Max Alloc	Max Alloc	Max Alloc
Olden					
bh	5,597,108	5,866,148	5,866,148	5,604,072	5,857,280
bisort	25,165,812	25,456,640	25,456,640	33,554,500	33,558,528
em3d	229,600,016	229,874,304	229,874,304	230,165,188	233,250,816
health	9,695,176	9,965,568	9,965,568	11,079,748	13,840,384
mst	54,525,952	54,841,344	5,4841,344	71,303,236	71,323,648
perimeter	61,721,996	62,107,648	62,107,648	70,541,380	70,541,312
power	846,544	1,118,352	1,118,352	864,468	1,019,904
treeadd	50,331,636	50,647,040	50,647,040	67,108,932	67,112,960
tsp	37,748,700	38,273,024	38,273,024	42111044	41,947,136
voronoi	34,157,376	34,426,416	34,426,416	34,164,340	63,516,672
Other					
debruijn	5,971,968	6,242,304	6,242,304	7,962,692	7,966,720
md	5,318,132	5,591,040	5,591,040	7,086,148	7,110,656

Table 7: Allocators' Fragmentation in Bytes

7 Experimental Results

7.1 Setup

We ran our experiments on a Linux Intel Xeon CPU 3.00GHz processor with 1024 KB L2 cache and g++ 4.1.2 with O3 optimizations. The reported counters are averaged across three runs. The Polaris compiler ran on an Intel Pentium 4 CPU 2.80GHz processor with 512 KB L2 cache and GNU gcc 3.2.2 and O2 optimizations. We compared our allocators with the original allocators used in the benchmarks. We also compared them with version 2.8.1 of Lea allocator (DL) , which is considered the best overall memory allocator [22]. Berger et al. [5] show that it competes with custom memory allocators, and sometimes even outperforms them.

The defaults for Velox were $K = 16$ for the hash table size, match-last-first allocation predicate and First allocation predicate for the size tree container. Medius has the same $K = 16$ default hash table size and $S = 4$ as the default starting bit. With this configuration we emphasized locality over memory overhead, by making the SK-classes coincide with virtual pages. For Defero we report results with 'next' allocation predicate and for $K = 20$, which also corresponds to the virtual page size on both systems.

7.2 Benchmarks

We experimented with Olden benchmark suite, which contains 10 applications. We also experimented with a molecular dynamics, a network simulation and a Fotran compiler. All the applications are written in C++. For applications which use STL, namely the latter three, the integration our of allocators was minimal: we just recompiled the code using our modified version of C++ STL. This STL version uses our allocators with its containers, such as list and tree, which automatically provide the hint address to the allocators. We describe this mechanism in [18]. For the Olden benchmarks we added the hint address for allocation by hand, following the same two simple rules that our modified STL uses: (i) for trees we allocated a child close to its parent and (ii) for lists we allocated an element close to its predecessor.

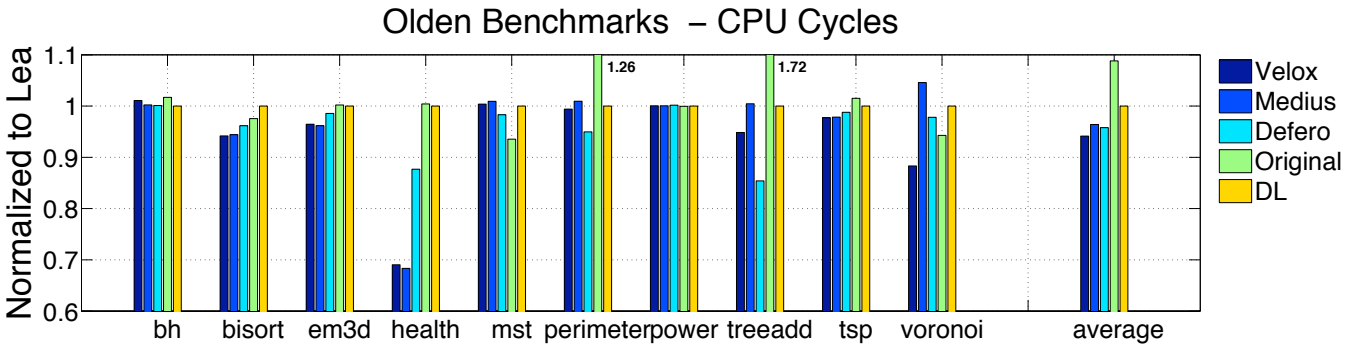


Figure 10: Olden Benchmarks - Normalized Execution Time to the Lea Allocator

All of our allocators were designed to follow the C++ Standard allocator interface [1] rather the the traditional malloc/free interface. Thus, the size of the chunk does not need to be stored in the chunk.

This is because the allocator infers the size from the type of the allocated element, type which is present in the allocator's interface. This fact gives our allocator an advantage when the size of the objects is in between segregated size classes and a rounding is needed.

Table 5 presents the benchmarks memory statistics together with the inputs. Table 6 shows the memory fragmentation of our allocators and the Lea allocator, calculated as the ratio between the additional memory used by the allocator at the point of maximum memory consumption and the maximum requested memory. Table 7 shows the same fragmentation in absolute numbers. Defero and Lea allocators have similar fragmentations, except for 'voronoi' which allocates 64 byte objects, required to by aligned by 64. Our allocators align a 64 byte request since they do not store the chunk size in its header and because they use mmap and sbrk to acquire memory from the system, in multiple of virtual pages. Velox and Medius have lower fragmentations, since they do not segregate but rather use exact sizes. Notice that the fragmentation for Velox and Medius is below 5% for all except one benchmark, 'power'. This fragmentation is well below the average for both Defero and DL allocators. Using less memory might also help improve locality.

Olden Benchmarks

The Olden benchmarks have 10,000 lines of code all together [15]. Fig 10 shows the execution time of the Olden benchmarks normalized to Lea allocator. For 'bh', 'health' and 'voronoi', Velox was equal, 30% and respectively 11% faster than the Lea allocator. For the remaining applications, Velox was between 0% and 5% faster than the Lea allocator. For applications that need a fast allocator and where a 'natural' order of allocation can fit into the temporal locality of the application, the benefits of improving locality might be outweighed by the allocator's speed. On average, Velox was 7% faster than Lea allocator, followed closely by Medius and Defero. The Lea allocator was faster than the original allocator by 10%. Figure 11 shows 5 hardware counters for 'voronoi' benchmark, namely L1 cache

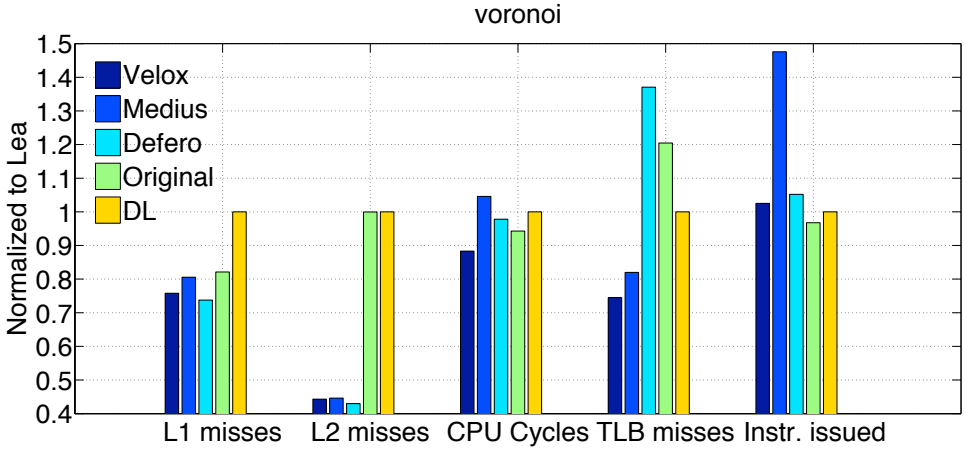


Figure 11: Voronoi's Hardware Counters

misses, L2 cache misses, CPU cycles, TLB misses and Instructions Issued. Both Velox and Medius outperform all the other allocators for locality counters, L1, L2 and TLB misses, reducing the TLB misses by more the 50%. However, the number of instructions executed by these allocators is larger than that of the Lea allocator's or the original allocator. This shows that the cost of improving locality is outweighed by the locality improvement.

Polaris is a Fortran restructuring compiler with approx. 600,000 lines of C++ code [14]. At its core, the most used container is a list similar to an STL list which stores various semantic entities, such as statements, expressions, constants, etc. We used our allocators as the underlying memory allocator for these lists. We manually modified the list's insert and erase operations to include our allocators' interface. The algorithms that we used in Polaris are compiler passes that perform program transformations, such as forward substitution, removing multiple subprogram entries, fixing type mismatches, etc. Figure 12 shows the execution time of compiling three Fortran benchmarks from the SPEC2000 suite. Defero with 'next' outperformed Defero with 'first' or 'match', improving on our previous work. It also outperformed Lea allocator by 7% on average. For the other allocators the results were mixed. On 'swim', Velox and Medius improved the execution time by 13-15% compared with the Lea allocator, while on others they were slower. On average, Velox was equal and Medius was slower than the Lea allocator. The results for Polaris depended heavily on the inputs. The only allocator which got a

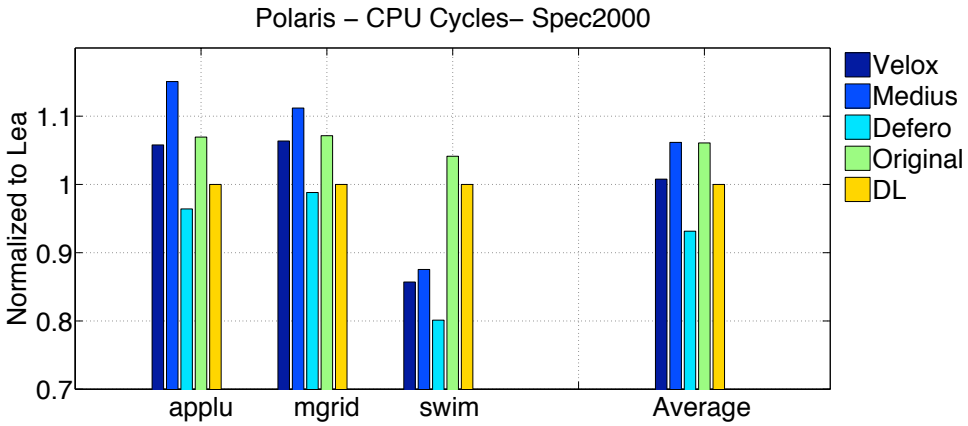


Figure 12: Compilation Time of SPEC 2000 Inputs with Polaris.

consistent improvement over Lea allocator was Defero with 'next'. The same trends exhibit for other input files such as SPEC89 and Perfect benchmarks.

Molecular Dynamics. The code has 1,800 lines of C++ code which fully utilizes STL containers, such as lists, trees and STL algorithms such as for_each, sort. The code computes the molecular interactions between particles in a time step algorithm. Fig 13 shows the execution time normalized to the Lea allocator. Medius and Defero with 'next' were close to the Lea allocator, while Velox was 10% slower. All allocators were faster than the original.

Debruijn -Network Simulation. The application is a micro-kernel which has 600 lines of C++ code and utilizes STL. The network simulation application uses a Debruijn graph [8] to simulate network behavior under random circumstances, such as node failures. Fig 13 shows the normalized execution time relative to the original allocator. Medius was 4% faster than Lea allocator, while Velox and Defero were 20% and 30% slower.

Summary

Across all benchmarks, 10 Olden benchmarks, a molecular dynamics code, a network simulation and

[8] A Debruijn graph has $O(N)$ vertices and approximately $O(\log N)$ number of edges for each vertex

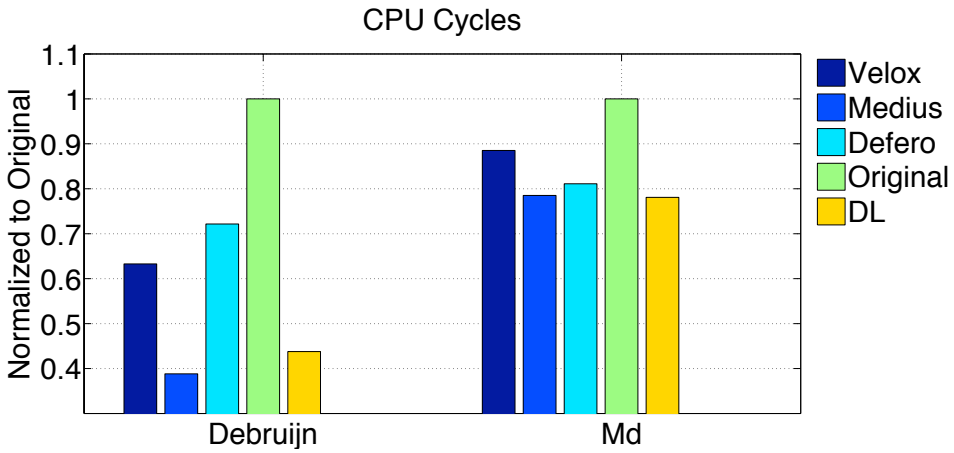


Figure 13: Debruijn and Molecular Dynamics - Execution Time

a Fortran compiler, there was no clear winner in terms of execution time. Velox improved the execution time by 7% on the Olden benchmarks compared with the Lea allocator, Defero by 8% on Polaris and Medius by 4% on Debruijn. However, in terms of memory usage, both Velox and Medius used 23% less memory on average than the Lea allocator, while Defero 11%. The memory reduction also contributes to the locality improvement.

8 Conclusions

We presented a theoretical formalism for general memory management and allocation, accompanied by its software library, Allotheque. Both tools allow programmers to focus on the allocation strategies rather than implementation. In fact, Allotheque is to memory allocation what the C++ Standard Template Library is to programming. We used Allotheque to build two new locality improving allocation strategies and another allocator as an extension to our previous work. Their implementation required several lines of code, which was as compact as the allocators' description in pseudo code. Our allocators improved or equaled the execution time when compared with the Lea allocator, while all of our new allocators reduced the memory usage by an average of 23%.

Each title in the bibliography below has an embedded `http` link to its corresponding file.

References

[1] International Standard ISO/IEC 14882. *Programming Languages – C++*. First edition, 1998.

[2] Giuseppe Atardi, Tito Flagella, and Pietro. Iglio. A customizable memory management framework. In Usenix - Advanced Computing Systems Association, editor, *Proceedings of the USENIX C++ Conference*, Cambridge, Massachusetts, USA., 1994.

[3] David A. Barrett and Benjamin G. Zorn. Using lifetime predictors to improve memory allocation performance. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 187–196, New York, NY, USA, 1993. ACM Press.

[4] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Composing high-performance memory allocators. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 114–124, New York, NY, USA, 2001. ACM Press.

[5] Emery D. Berger, Benjamin G. Zorn, and Kathryn S. McKinley. Reconsidering custom memory allocation. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 1–12, New York, NY, USA, 2002. ACM Press.

[6] Gerald Bozman. The software lookaside buffler reduces search overhead with linked lists. *Commun. ACM*, 27(3):222–227, 1984.

[7] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 139–149, New York, NY, USA, 1998. ACM Press.

[8] Sigmund Cherem and Radu Rugina. Region analysis and transformation for Java programs. In *ISMM '04: Proceedings of the 4th international symposium on Memory management*, pages 85–96, New York, NY, USA, 2004. ACM Press.

[9] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pages 1–12, New York, NY, USA, 1999. ACM Press.

[10] W. T. Comfort. Multiword list items. *Commun. ACM*, 7(6):357–362, 1964.

[11] David Gay and Alex Aiken. Memory management with explicit regions. In *PLDI '98: Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, pages 313–323, New York, NY, USA, 1998. ACM Press.

[12] Dirk Grunwald, Benjamin Zorn, and Robert Henderson. Improving the cache locality of memory allocation. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, pages 177–186, New York, NY, USA, 1993. ACM Press.

[13] Josefin Hallberg, Tuva Palm, and Mats Brorsson. Cache-Conscious Allocation of Pointer-Based Data Structures Revisited with HW/SW Prefetching, 2003.

[14] University of Illinois - Urbana Champaign. Polaris - Compiler Optimization. Texas A&M University.

[15] Princeton University. Olden Benchmarks.

[16] Xianglong Huang, Stephen M. Blackburn, Kathryn S. McKinley, J Eliot B. Moss, Zhenlin Wang, and Perry Cheng. The garbage collection advantage: improving program locality. In *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 69–80, New York, NY, USA, 2004. ACM Press.

[17] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? In *ISMM '98: Proceedings of the 1st international symposium on Memory management*, pages 26–36, New York, NY, USA, 1998. ACM Press.

[18] Alin Jula and Lawrence Rauchwerger. Custom memory allocation for free. In *The 19th Workshop on Languages and Compilers for Parallel Computing*, New Orleans, Louisiana, USA, 2006. Springer-Verlag Lecture Notes in Computer Science.

[19] Kenneth C. Knowlton. A fast storage allocator. *Commun. ACM*, 8(10):623–624, 1965.

[20] Donald E. Knuth. *The art of computer programming, volume 1 (3rd ed.): fundamental algorithms*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1997.

[21] Chris Lattner and Vikram Adve. Automatic pool allocation: improving performance by controlling data structure layout in the heap. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 129–142, New York, NY, USA, 2005. ACM Press.

[22] Doug Lea. A memory allocator. *The C++ Report*, 1989.

[23] Machael J. Karels Marshall Kirk McKusick. General Purpose Memmory Allocator for the 43.BDS UNIX Kernel. In *Proceedings of the San Francisco USENIX Conference*, pages 295–303, June, 1988.

[24] David R. Musser and Atul Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[25] P. W. Purodm, S. M. Stigler, and Tat-Ong Cheam. Statistical Investigation of Three Storage Allocation Algorithms. Technical Report 98, Computer Science, University of Wisconsin - Madison, 1970.

[26] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 12–23, New York, NY, USA, 1998. ACM Press.

[27] Yefim Shuf, Manish Gupta, Hubertus Franke, Andrew Appel, and Jaswinder Pal Singh. Creating and preserving locality of java applications at allocation and garbage collection times. In *OOPSLA '02: Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 13–25, New York, NY, USA, 2002. ACM Press.

[28] Thomas A. Standish. *Data Structure Techniques*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[29] C. J. Stephenson. New methods for dynamic storage allocation (Fast Fits). In *SOSP '83: Proceedings of the ninth ACM symposium on Operating systems principles*, pages 30–32, New York, NY, USA, 1983. ACM Press.

[30] D. N. Truong, F. Bodin, and A. Seznec. Improving Cache Behavior of Dynamically Allocated Data Structures. In *PACT '98: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, page 322, Washington, DC, USA, 1998. IEEE Computer Society.

[31] Kiem-Phong Vo. Vmalloc: A general and efficient memory allocator. *Software Practice and Experience*, pages 357–374, 1996. 26(3).

[32] Charles B . Weinstock and William A. Wulf. QuickFit: An Efficient Algorithm for Heap Storage Allocation. *ACM SIGPLAN Notices*, pages 141–144, 1988. 23(10).

[33] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *IWMM '95: Proceedings of the International Workshop on Memory Management*, pages 1–116, London, UK, 1995. Springer-Verlag.

[34] www.freebds.org. *Documentation of BDS 4.4*.