Custom Memory Allocation for Free
Improving Data Locality with Container-Centric Memory Allocation

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

Texas A&M University , College Station, Texas

Abstract. We propose a novel *container-centric* memory allocation scheme. In this scheme, the container's semantics guide the memory allocation, which results in data locality improvement and execution time reduction. The container-centric allocation provides the benefits of custom memory allocation, with the portability advantage. Applications need not change a single line of code, but rather change the underlying container library. Container-centric memory allocation increases data locality and reduces execution time, at no cost.
At compile time, the container's semantics provide knowledge which is evaluated at run-time, and then used for more efficient memory clustering. Our approach enables an application to use different allocation policies for different types of containers, or even different instantiations of the same type of container. We have integrated our memory allocator, named *Defero*, with the C++ Standard Template Library (STL) containers for automatic usage. We have used Defero in applications such as compiler infrastructure, molecular dynamics, network simulation, as well as on micro-kernels.

1 Introduction

Memory allocation is performed from various information-rich contexts, such as containers, libraries, and domain specific applications. Yet, traditional allocation schemes do not use the semantic information present in these rich contexts, but rather allocate memory based only on the *size* of the request. For example, to solve the spatial data locality for dynamically allocated memory, we need to analyze the memory based on its *location*, not its size. Data locality cannot be solved based on size only. Thus, we need better memory allocation to improve data locality.

This paper studies how memory allocation automatically benefits from the knowledge present in the STL containers, and how this knowledge gets communicated from the containers to the memory allocator.

Data locality has been given a great deal of attention by numerous people over the years. In fact, compiler techniques offer a plethora of optimizations, such as tiling, register allocation, and field reorganization, which increase data locality for regular data structures [Wolf 87,Ding 99,Chil 99b]. However, compiler analysis is less effective for dynamic data structures. The intrinsic dynamic property of these structures inhibits the analysis. This is because the information necessary to perform the optimization is not available at compile time, but it becomes available only at run time as the program executes.

We present a novel and efficient container-centric memory allocator, named *Defero*[1], which allows a container to guide the allocation of its elements. This guidance is supported by the semantic-rich context in which a new element is inserted, by suggesting where to allocate each individual element. The communication between containers and the memory allocator allows Defero to automatically increase data locality for containers.

We integrated Defero to work automatically with the C++ Standard Template Library (STL) containers [Inte 98]. The integration yielded improvements in data locality for applications that use STL. Applications benefit from the performance improvement provided by Defero while maintaining their portability, without having to design a custom memory allocator. Thus, applications can get the best of both worlds: the improved performance of custom memory allocation and the hassle-free portability of general memory allocation.

This paper makes the following contributions:

– ***A container-centric memory allocation with a simple interface*** which guides the allocation process. It allows users or compilers to easily specify various memory allocation policies at the container instantiation level.
– ***Allocation based on multiple attributes.*** Defero allocates based on 'size' and 'location' as an additional allocation goal. The policy of allocating a new object *close*(in memory space) to a related object (hinted by the user or compiler) results in improved memory reference locality. This approach also minimizes the external fragmentation.
– ***An adjustable memory scheme, K-Bit***, which allows data locality to be controlled with a simple number. We also present a novel allocation predicate, named Path, which increases data locality for balanced trees.

We present experimental results from various areas, such as molecular dynamics, network simulation, and compilers, as well as micro-kernels.

The remainder of the paper is organized as follows. Defero's design and implementation, along with examples are presented in section § 2. Several allocation policies are discussed in section § 3 , while section § 4 reasons upon the criteria used to select a certain allocation scheme. Defero's automatic interaction with containers, as well as its integration in libraries is described in section § 5. Experimental results are described in section § 6, and section § 7 discusses related work. Conclusions are presented in section § 8.

2 Defero Memory Allocator

In this section we discuss Defero's design principles, implementation and examples.

Memory allocation schemes fall into two main categories. On one end of the spectrum, a large number of memory allocation schemes use a *general* memory allocation policy for all applications. These policies are memory allocation centric, in which the

[1] In Latin, *defero* means *"to hand over, carry down, communicate, offer, refer"*

focus is on memory allocation alone. They are rigid and do not adapt to the application's needs, and therefore are not optimal. On the other end of the spectrum, *custom* allocation schemes are tailored to specific applications. These schemes are application-centric, in which the focus is on that specific application's memory allocation alone. They have a tremendous impact on performance, but they are not portable.

Defero is different. It is *container-centric*, which means that containers guide the memory allocation. Containers collect their specific semantic information and pass it to Defero. Based on this information, Defero guides the memory allocation and deallocation routines. Therefore, Defero communicates with applications through their containers.

The result of container-centric memory allocation is a custom memory allocation policy nested into each container. It has the performance advantage of custom memory allocation, without its portability disadvantage. Defero can be integrated in STL, and thus the portability is shifted from the application, to STL. Applications need not change a single line of code to take advantage of the customized allocation scheme provided by STL containers. Defero, thus, provides custom memory allocation for free. Section $\S$ 5 provides more details on Defero integration into applications.

2.1 Design

Memory Partition We now present how Defero partitions the memory and how it manages its free chunks. Defero regards all its free memory chunks as an algebraic space of memory addresses. Defero organizes these addresses into *equivalence classes*, based on an *equivalence relation*. The equivalence relation is transitive, reflexive, and symmetric and defines a partition over the space. An example of an equivalence relation is "congruent to modulo 5" between integers. An equivalence relation has the mathematical property of partitioning the space into equivalence classes. The equivalence classes are disjoint and cover all the space.

This generic space partitioning based on an equivalence relation provides flexibility in the way Defero manages its free memory chunks. Because of its flexibility, previous allocation policies can also be implemented with our design. Consider the segregated lists approach, first described by Weinstock in [Wein 76]. It manages small object classes, for example up to 128 byte-objects, rounded up to a multiple of 8. All objects of a certain size are kept together in a linked list. The segregated lists partition can be expressed as an equivalence relation in Defero: let x and y be memory addresses, then $x \equiv y$ if $(size(x) \geq 128 \wedge size(y) \geq 128)$ or $(\lfloor \frac{size(x)}{8} \rfloor = \lfloor \frac{size(y)}{8} \rfloor)$ otherwise.

New partitions can also be created using this generic space partition. In this paper we analyze a new equivalence relation, named *K-Bit*, which improves data locality. K-Bit is defined as: if x and y are memory addresses, they are equivalent iff the first K higher-order bits are the same. K-Bit partition keeps the memory organized in groups of addresses where the first K bits are identical. The K-Bit partition results in 2^K groups, each with a maximum size of 2^{32-K}. This is an invariant throughout the whole program execution. At any point in the program execution, regardless of the deallocation pattern or distribution, the memory is organized into classes of available contiguous chunks. This property is not true for traditional allocators, since out-of-order deallocations violate this invariant. K-Bit ensures that even the most complicated allocation patterns will not affect Defero's memory organization.

The K parameter is adjustable. Assuming a 32-bit system and a virtual page size of 2^K bytes, the equivalence classes coincide with the system's virtual pages. With the appropriate K, memory can also be partitioned in cache lines.

Defero orders these equivalence classes based on an *order predicate*, which allows equivalence classes to be compared against each other. An example is the trivial order predicate $'less' =' \leq'$, which orders the K-Bit equivalence classes based on the first K bits. Equivalence and order predicates are traits associated with memory management. The user can select the equivalence and order predicates or use the defaults provided by Defero. The equivalence classes are ordered, thus searchable, by virtue of having an order predicate.

Allocation Predicate We can now focus on searching the memory partition. The idea behind flexible space partitioning is to facilitate the allocation/deallocation process. An *allocation predicate* stores a target equivalence class, with the intention of allocating a memory chunk from that specific equivalence class. The allocation predicate has two critical pieces of information: (i) a target class, the equivalence class from which the memory chunk is allocated, and (ii) a search algorithm, which describes how to find that specific equivalence class. The allocation's searching algorithm is implemented as a ternary predicate:(i) 0 for found; 'I found it, stop searching', (ii)-1 for moving left ; 'I need a smaller one' and (iii) 1 for moving right ; 'I need a bigger one'. This binary search guides the searching process among the ordered equivalence classes.

Defero's generic interface allows users to select their own memory partition and allocation predicate. This flexibility allows different containers to have different allocation policies within the same application.

2.2 Implementation

While other equivalence relations can be easily integrated in Defero, the focus of this article is on K-Bit equivalence relation. We analyze K-bit partitioning and K's impact on memory allocation. We define *K-class* as the equivalence class which contains all the memory address that have the same K first bits. In the remainder of the paper we will use the term K-class as an equivalence class instance, without loss of generality.

Now that we have the generic framework, we can proceed to concrete instantiations. Defero organizes its memory in a 2-dimensional space. The first dimension is based on 'size' and is partitioned using the segregated-lists approach. The second dimension is the 'address' , and is partitioned using K-Bit. Consequently Defero sees the memory as a 2-dimensional space, with 'size' and 'address' as orthogonal dimensions.

The first dimension, 'size', is organized in segregated classes. The benefit of segregated size classes is that they generally work well on a large class of applications. We followed the SGI STL allocator guidelines for the size dimension [SGI Web]. All free chunks of a certain size are partitioned and ordered according to the K-Bit partition. This design benefits from the empirically proven segregated lists approach, while it also allows for a custom partition of the memory space. Fig. 1(a) shows Defero's implementation where each size class has its own K-Bit partition.

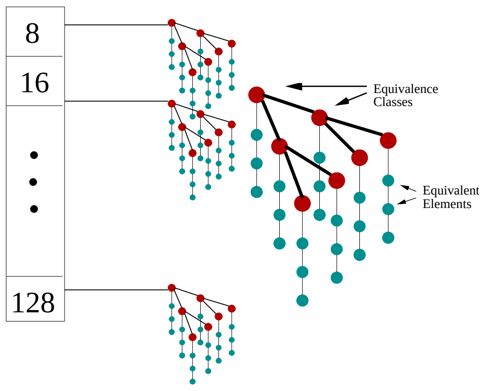


Fig. 1. (a) K-Bit organization (b) Defero's internal structure

The second dimension, 'address', is organized by the K-Bit partition. Defero organizes the K-classes of K-Bit partition into an ordered red-black balanced tree, based on the order predicate. A node in the tree represents a K-class, and all of the elements in the same K-class are stored in a linked list. Fig. 1(b) depicts the K-Bit's organization.

The allocation process selects a free chunk of memory from a 2-dimensional memory space. The allocation takes into consideration both dimensions. The first dimension selects size, and the second dimension selects locality. Allocation based on 'size' is performed using the traditional segregated-lists allocation, where the size of the memory request determines the appropriate entry into the hash table. It then selects a K-class within this tree, based on the allocation predicate. A memory chunk from the selected K-class is returned as the allocated memory.

To improve locality, the allocation searches for the 'closest' available address to a *hint* address. This hint address is automatically provided by the container semantics. For example, an element to be inserted at the back of a list, can be allocated near the tail, which is the hint address. This information is not known at compile time, and only the container has that information at run-time. If the K-class of the hint address is found, then an equivalent address from that class is returned. There is no guarantee which one, only that the returned address is equivalent with the hint address. Otherwise, the 'closest' K-class is considered, according to the order predicate, and an element from this class is returned.

Upon deallocation, the memory address is returned to its designated K-class. If its K-class exists in the tree, then the address is inserted in the K-class linked list, otherwise, a new tree node is inserted in the tree.

The allocation complexity depends on two variables: (i) the allocation predicate and (ii) the number of K-classes in the tree. The deallocation complexity depends only on the number of K-classes in the tree. Both allocation and deallocation complexities are $O(K)$.

Examples *Defero*'s interface is similar to the STL allocator interface. The STL allocator accepts the size of the memory request as input parameter, along with an optional hint address [Inte 98]. Defero generalizes the hint address to an allocation predicate. The next section § 3 shows how the allocation predicate can hold a hint address or address distribution information. The deallocation interface remains unchanged. Fig. 2 shows an example of how Defero is used. Lines 1-2 show allocation with different predicates, while lines 3-4 show different container instances using different allocation predicates.

```
      // Allocate 4 bytes with First predicate
1. int* y= defero::allocate(4,First(0));
      // Allocate z near x, or the closest
2. int* z= defero::allocate(4,Best(x));
      // List with Defero(K-Bit, First)
3. list<int,defero<int,Kbit<12>,less<int> >,First> my_smart_list_1;
      // List with Defero(K-Bit, Best)
4. list<int,defero<int,Kbit<12>,less<int> >,Best> my_smart_list_2;
```

Fig. 2. Example of Defero using Equivalence, Order and Allocation Predicates.

It is worth noting the ease of selecting a different allocation policy, which is a few lines of code. Users can easily experiment with different allocation policies. Different architectures and containers can benefit from specific memory partitions and allocation predicates. Defero's generic interface allows for an easy and elegant selection of these salient attributes.

3 Allocation Predicates

In this section we describe two existing allocation predicates and a novel one, named *Path*, with which we experimented. We describe each individual one and then discuss the advantages and disadvantages when combining them with K-Bit partition.

First allocation predicate selects the first K-class situated at the root of the tree. If the root class has elements, it returns the first element in the list, otherwise, it returns the root. First guarantees consecutive allocations from the same K-class, regardless of the allocation/ deallocation pattern. Thus, First favors programs that exhibit *temporal locality*. The allocation complexity is constant time amortized, and deallocation complexity is O(K) .

Match allocation predicate takes a hint address and returns the closest available address to this hint address. It is a binary search performed on a tree, with the key as hint address. If the hint address's K-class is found in the tree, an equivalent address is returned, else an element from the closest K-class is returned. Match can be described as a Best fit with regard to address location. We will use these two terms, Match and Best, interchangeably in the remainder of the paper.

Match exploits *spatial locality*, by attempting to allocate objects close to each other. Match also favors irregular allocation patterns by explicitly ensuring spatial locality. The closeness metric, or how close, is dictated by K. The allocation/deallocation complexity is O(K).

We now describe a novel allocation predicate, **Path**, designed to improve locality for sorted trees. Balanced trees perform rebalancing operations that modify the structure of the tree and hurt the spatial locality. An example is the red-black trees with its *rotate left* and *rotate right* rebalancing operations. The Path allocation predicate attempts to solve the spatial locality problem introduced by the rebalancing operations. When an element is inserted in a tree, a top-down search is performed. The search path is recorded as a series of bits: 0 for visiting the left sub-tree and 1 for visiting the right sub-tree. Path then allocates elements with similar paths together, since it is very likely they will end up in the same part of the tree. The complexity of Path is O(K).

The idea behind Path is that low values should be allocated together and high values should be allocated together, regardless of the order in which they were inserted in the tree. Values that create similar paths are allocated together, since they are likely to be to be close in the tree. The tree passes the path to the allocator, and the allocator follows this path in its own top down tree search. Path maps values to addresses. Elements that have approximately the same value are allocated together, thus likely to be accessed together. Intuitively, it makes sense to allocate together the elements which are logically grouped by the application. The allocation process follows a given path into the K-Bit searching tree. This allows for directing the allocation requests toward a certain part of the tree. It thus creates an isomorphism between the element's value and address.

K-Bit with ⇒	First	Match	Path
Advantage	+ Temporal Locality + Fast	+ Spatial Locality + Container aware + Fixes worst case allocation distributions	+ Logical Locality + Container aware
Disadvantage	- Not container aware	- Slower	- Slower

Table 1. Strengths and weaknesses of allocation predicates with K-Bit

The table 1 shows the strengths and weaknesses for each allocation predicate when used with K-Bit.

4 Selecting K

In this section we characterize the dynamic property of containers and discuss how this characterization helps in selecting an optimal K. We start by asking how dynamic is a container?The answer depends on how the application operates the container. Assuming we know the operations applied on the container by the application, we now quantify the dynamic property of containers. First, we classify the types of operations on a container into two categories: (i) *modifying* (M) operations, such as insertion or deletion and (ii) *non-modifying*(NM) operations, such as traversals or element access. [2] M operations stress the memory allocator, while NM operations stress the locality of the container.

Next, we define the **dynamism** of a container as a measurable metric. The dynamism of a container represents the percentage of modifying operations in the total operations:

$$D = \frac{M}{NM + M} * 100, withD \in (0, 100), M \geq 1, NM \geq 1^3 \tag{1}$$

Intuitively, the dynamism describes the speed of change in a data structure. The more often it changes, the higher the dynamism.

The dynamism of the container is highly correlated with memory behavior. The higher the dynamism, the more important the speed of the memory allocation becomes, while data locality diminishes in its importance. The lower the dynamism, the more important locality is, while memory allocation's speed becomes less dominant. Defero benefits applications in between these extremes, since the extremes already have optimal solutions.

K is an adjustable parameter. It varies between 0 and the size of the pointer, 32 or 64 depending of the system. On one end, when K is the size of the pointer, each memory chunk is its own K-class. In this case the equivalence relation is the classic equal predicate '='. On the other end, when K is 0, all memory chunks are equivalent, and there is only one K-class in the tree, the whole memory. Therefore, K dictates the ratio between trees' and lists' sizes and the number of K- classes it creates.

To reduce the allocation overhead, one might consider either reducing K to obtain fewer K-classes, choosing an allocation predicate that selects faster, like First for example. To increase data locality, one might consider either increasing K for a better refinement of the space or choosing an allocation predicate that selects better, like Match.

Therefore, high values of K speed up NM operations, but slow down M operations. Low values of K reverse the trend. The process of selecting K is dictated by the application needs. A highly dynamic container requires a low K, and vice-versa. Low K though does not help locality, and a high K does not allocate fast. Each application has a point of equilibrium which maximizes the interaction of these two antagonistic characteristics.

5 Integration of Defero with STL

The advantage of integrating Defero with STL is knowing the information rich context from where memory is allocated. The container allocates its elements and is the perfect candidate to provide allocation guidelines to its elements' allocation. Defero allows the communication of this semantic context to the allocation policy.

[2] Modifying can be regarded as Write operations and Non-Modifying as Read operations

[3] Each container has at least 1 M operation, the creation of that container. Every container has at least 1 NM operation. If it didn't have one, it meant that that container was never accessed by the program and thus would be dead code.

We integrated Defero into the C++ STL. The library provides various generic data containers, such as lists, sets, and trees. STL container's modular design made Defero's integration easy.

For list-like containers, traversals imply a linear order of their elements. Elements in these containers find themselves in a partial order. Because of this intrinsic invariant property, the best place to allocate a new element to improve locality, is in the proximity of its neighbors. This neighbor(s) context is passed to the allocation predicate, which is then passed to the allocation routine.

For tree-like containers, insertion is based on the new element's value, a run-time variable. This insertion context is more complex than the list's. One can think of several allocation predicates that increase locality: allocation close to the parent, allocation close to the sibling, allocation based on the value distribution, etc.

Each container can benefit from selecting its own allocation predicate by providing the best allocator for the application. Applications use theses STL containers, without the knowledge of their allocators. Consequently, Defero keeps the application portable and thus provides custom memory allocation for free.

6 Experimental results

We tested Defero using several dynamic applications from various areas, such as compiler infrastructure, molecular dynamic, network simulation, as well as in-house micro-kernels. These applications use STL containers. Defero integration effort was minimal. The integration required the inclusion of the modified STL in the include path of the compilation command. We did not modify the applications.

We conducted the experimental results on a Intel(R) Xeon(TM) 3.00 GHz, 1GB memory, using GNU C++ compiler *g++* version 3.3.6, with -O3 level of optimization. All the experiments were run three times and the average was considered.

We compared Defero against Doug Lea's allocator (DL), the GNU STL allocator [SGI Web], and the native g++ 3.3.6 "new" and "malloc" allocators. The DL allocator uses a size-segregated approach, and boundary tags. This allocator is considered the best overall memory allocator [Lea 90,Lea 96]. Berger et al. [Berg 02] show that it competes with custom memory allocators, and sometimes even outperforms them. The GNU STL allocator is a segregated list allocator, with first fit policy and no coalescing. This allocator was based on a SGI implementation. We set the GNU STL allocator as our base allocator because of its portability and ubiquity. It is worth noting that the difference between the GNU STL allocator and Defero is exactly the K-Bit partition. Consequently, any improvement over the GNU STL allocator is solely attributed to K-Bit partition with allocation predicates.

Defero, DL and GNU STL allocators used exactly the same amount of memory for all applications analyzed in this paper. The minimum memory request was 12 bytes, which was rounded up by the GNU STL allocator to 16 bytes, while Defero requires a memory chunk threshold of at least 16 bytes, the same threshold required by the DL allocator.

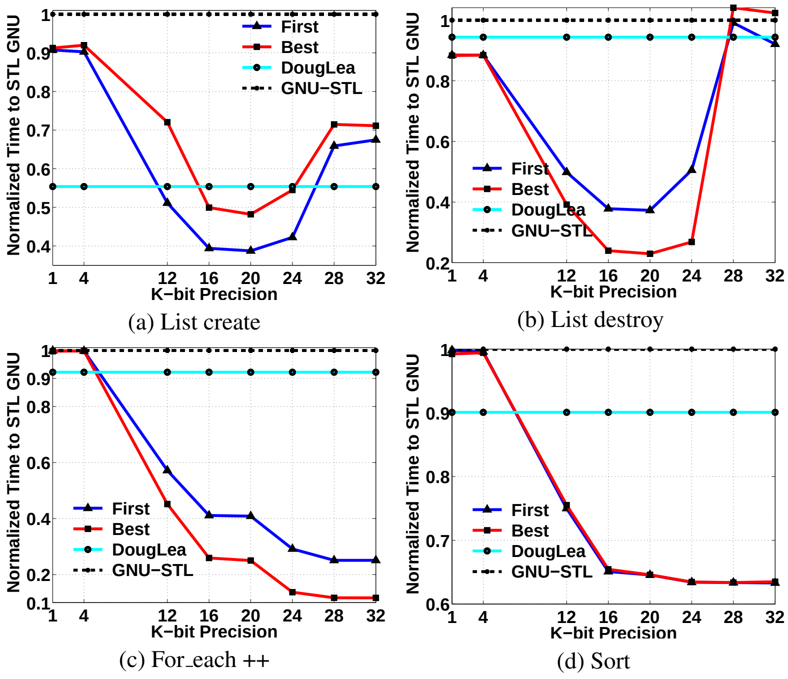
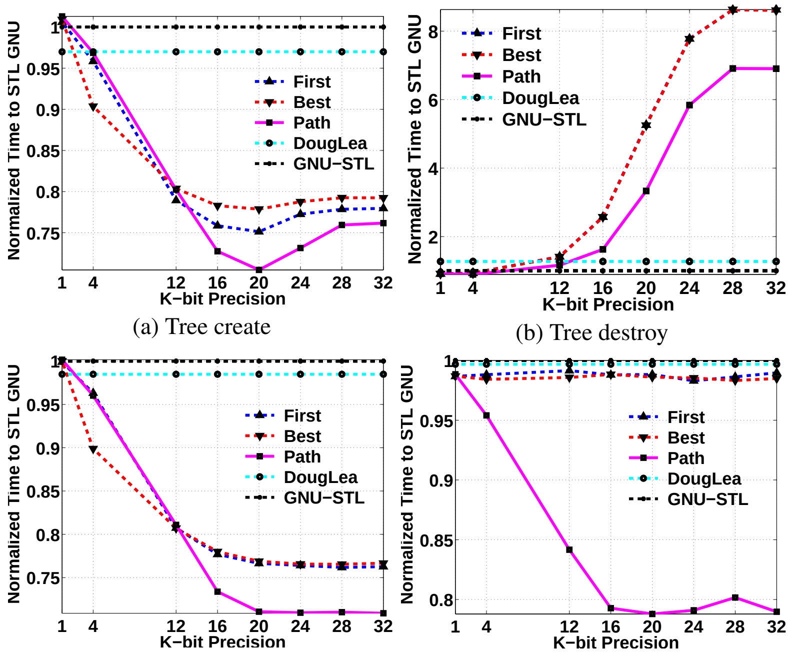


Fig. 3. List operations with Defero and K-bit partition

List Micro-kernels In the List micro-kernel we created a list of one million random integers. The memory was warmed up by allocating two millions integers and then we randomly erased one million, in order to simulate a real scenario where the application randomly allocates and deallocates objects. We then created a list of elements alternating *push_back* and *push_front* methods. We then invoked two STL algorithms: list's own *sort* and *for_each* with incrementing the elements as the called function. All measurements are relative to the native GNU STL allocator.

Fig. 3 (a) shows the list creation. First reduces the list creation by 60%, while Best reduces it by 50%. Even though the DL allocator is faster, Defero improves locality which eventually wins over allocation speed. Fig. 3 (b) shows the list clearing which is reduced 77% with Best and 62% with First. It is worth noting that Best performs almost 2 times faster than First and 3.5 times faster that DL allocator. Fig. 3 (c) shows the STL foreach algorithm. The higher the locality precision of the partition, the better the locality. Best reduces the execution time by almost 90% and it is 9 times faster than the DL allocator. Fig. 3 (d) shows the execution time of the list's sorting algorithm. Both Best and First perform about the same with a 35% improvement over the GNU STL allocator and 25% over the DL allocator.

The experimental results corroborate our hypothesis: high locality precisions favor traversal operations, such as for_each, sort, at the expense of modifying operations, such as insert and clear. The sweet spot is determined by the ratio of these two types of operations. For traversals, the Best allocation predicate performs two times better than First, but First is faster on modifying operations.



(a) Tree create (b) Tree destroy (c) Find (d) Inorder traversal

Fig. 4. Tree operations with Defero and Kbit partition

Tree Microkernels In the Tree micro-kernel, the memory was warmed up in a similar fashion as in the list experiment. An STL tree with nine million integers was allocated and then three million elements were randomly erased. We then created an STL red-black tree of three million random integers. Once the tree was created, each key was randomly selected and searched in the tree. We then applied a for_each algorithm, after which the tree was destroyed.

Fig. 4 (a) shows the tree creation operation. Defero with Path improves the execution time by 30% over GNU STL allocator and 25% over the DL allocator. Fig. 4 (b) shows the tree destruction. This operation observed a slow-down. The inorder traversal is not substantially sped up to compensate for the extra memory management involved in deallocation. As the precision decreases, the execution time approaches the STL allocator execution time. Luckily, the tree destruction is not a common operation in many applications. Fig. 4 (c) shows three million top-down traversals performed by *find* operation. Defero with Path improves the execution time by 28% over the GNU STL and 25% over the DL allocator. Fig. 4 (d) shows the for_each algorithm. While Defero with

Best and First did almost the same as DL and the GNU STL allocators, Defero with Path improved the execution time by 20% over all of them.

These experiments show that specialized allocation predicates benefit different containers. For example Path works best for trees, while First and Best work best for lists.

While we do not expect real applications to exhibit such improvements, these algorithms and operations are the basic blocks of each application. Understanding the impact of memory allocation on these basic yet fundamental, algorithms, is important for a better understanding of real applications.

Next we present applications results at the application level.

Application	Input	Code size	Exec. Time (sec)	Total (MB)	Max Mem (MB)	Alloc/Dealloc (Mil)	Avg. Size (bytes)
MD	7000 particles	1800	88.32	167.34	14.68	13.9 / 12.7	12
DeBruijn	20,736 nodes	600	12.96	268.98	6.22	22.4 / 22.3	12
Polaris	Spec89, Perfect	600,000	300	128.921	18.977	3.9 / 3.7	32.73

Table 2. Benchmarks General Characteristics

Table 2 describes the benchmarks and their memory behavior. The table contains the inputs used, the size in lines of code of the benchmarks, the execution time, total memory allocated, maximum memory used, number of allocation and deallocation performed, and the average size of the memory allocation requests.

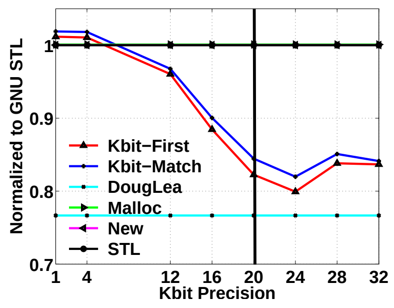
Applications	Containers	#Containers	Total Op. (Mil)	# NM (Mil)	# M (Mil)	Dynamism
MD	list,tree	6,912	551.46	524.75	13.93 / 12.78	2.58
DeBruijn	list	20,736	94.26	59.50	22.39 / 22.37	27.34
Polaris	list	-	27.47	18.75	3.93 / 3.78	17.15

Table 3. Benchmarks Containers Characterization

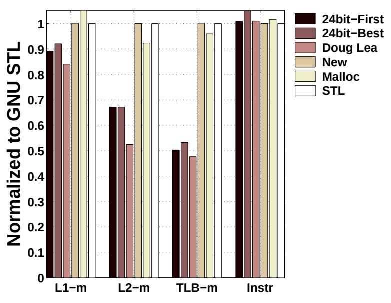
Table 3 characterizes the data containers used by these benchmarks. The table contains the containers used in the application, the number of containers used, the total operations performed on these containers, the number of non-modifying operations and modifying operations, along with the average dynamism of the containers.

Molecular dynamics was written by Danny Rintoul from Sandia National Laboratories. The application fully utilizes STL containers, such as lists and trees and STL algorithms such as for_each, sort. It is an N-body problem with time steps.

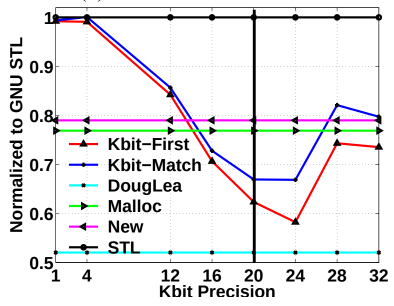
Fig. 5(a) shows the normalized execution time of the whole application. While the DL allocator reduces the execution time by 22%, Defero comes in very close. Defero with First improves by 20%, and Defero with Best by 18%. Fig. 5(b) shows a comparison of five hardware counters : L1 misses, L2 misses, TLB misses and Issued Instructions. These hardware counters were collected for the following memory partitions



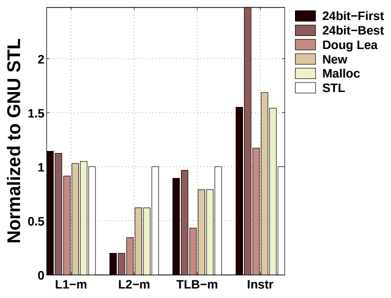
(a) MD- Execution Time



(b) MD- Hardware Counters



(c) Debruijn Execution Time



(d)Debruijn - Hardware Counters

Fig. 5. Molecular Dynamics and Debruijn Network

and allocation policies: 24-bit with First, 24-bit with Best, DL, native 'new' and native 'malloc'. The vertical line in figures 5(a) and (c) represents K=20, the native virtual page partition of the system.

Molecular dynamics is a discrete event simulation. In between time steps, the application releases almost all of its memory to the system, only to allocate it again in the next step. This behavior benefits the DL allocator, which uses coalescing to reduce external fragmentation.

Network Simulation -Debruijn was written by Derek Leonard and Dmitri Loguinov of Department of Computer Science at Texas A&M University. The network simulation application uses a Debruijn graph [4] to simulate network behavior under random circumstances, such as node failures.

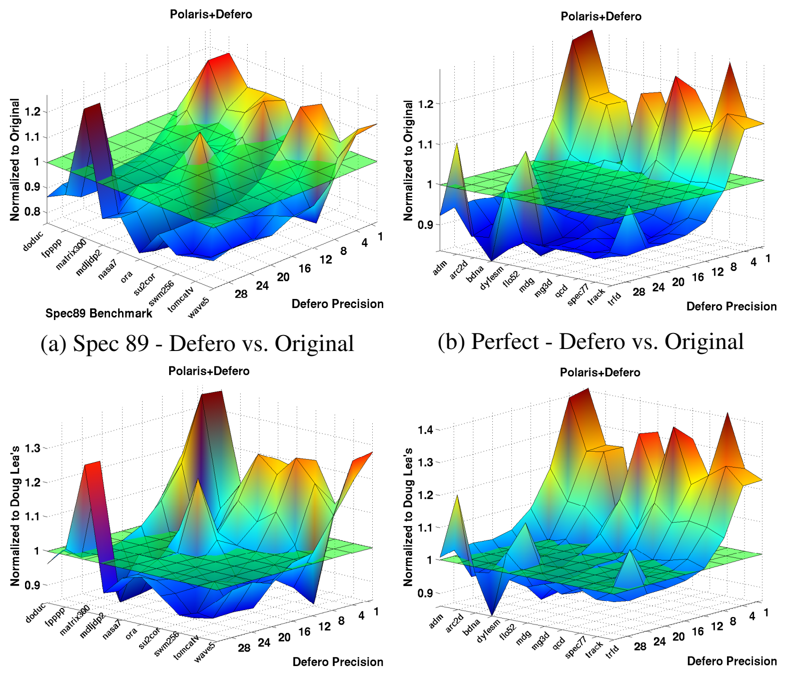
Fig. 5(c) shows the normalized execution time of the Debruijn application. Defero improves 42% over the GNU STL allocator and 20% over native 'new' or 'malloc', but the DL allocator is 7% faster than Defero. The dynamism of the containers used in this application is the highest of our benchmarks (see table 3), with a dynamism of 27.34. The high dynamism favors fast allocators, and thus the DL allocator wins over Defero. Fig. 5(d) shows hardware counters for the Debruijn application. It is interesting

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

to note that even though Defero with Best has twice as many instructions as Doug Lea's allocator, the execution times are very close. This shows that in this application the effort put into improving locality is compensated by the locality improvement it produces.

These experimental results corroborate our hypothesis that highly dynamic containers benefit from fast allocators, while less dynamic containers benefit from improved locality.

Polaris is a Fortran compiler infrastructure [Pola 01]. 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 Defero as the underlying memory allocator for these lists. 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.



(a) Spec 89 - Defero vs. Original

(b) Perfect - Defero vs. Original

(c) Spec 89 - Defero vs. Doug Lea

(d) Perfect - Defero vs Doug Lea

Fig. 6. Execution Time - Polaris with Defero on Spec 89 and Perfect Benchmarks

We ran Polaris with inputs from Perfect and Spec 89 benchmarks. For each input we varied the K from 0 to 32. Fig. 6(a) and 6(b) show the performance of Polaris using Defero with First relative to the original memory allocation. First outperformed Best with Polaris. In the figures, anything below the plane is faster and anything above is slower. There is a 22% improvement relative to the original implementation (malloc) for input 'ora'. Fig. 6(c) and 6(d) show the relative improvement over the DL allocator. Compared to the DL allocator, the largest improvement of 12% was obtained for input 'tomcatv'.

It is worth noting the consistency in improving execution time across all inputs from the two benchmarks. This consistency shows that Defero improves data locality regardless of the input. Almost all inputs exhibit the same pattern with respect to K's variance. For Spec 89 benchmark, the average improvements are 15% over the original and 10% over the DL allocator. For the Perfect benchmark the average improvements are 10%over the original and 5% over the DL allocator.

Defero can help automatic program parallelization. For run-time analysis, the compiler needs to compile parts of a program at run-time, due to lack of information at compile-time [Rus 03]. The compilation time, thus, becomes an pivotal cost factor for the success of the program parallelization. Defero can reduce the compilation time involved in the run-time parallelization methods.

In summary, Defero improves data locality, especially large applications with complex allocation patterns. Defero increases data locality explicitly, with the help of the allocation predicates. The Best allocation predicate explicitly improves locality, while First improves locality implicitly, by exploiting temporal locality.

Different containers can benefit from different allocation predicates. For example, the Path allocation predicate encapsulates the semantic structure of the tree and helps improving tree locality. Path is specific to balanced trees. Lists benefit better from other allocation predicates, like First and Best.

7 Related Work

In this section we describe related work in memory management and we compare it with Defero.

Lattner and Adve [Latt 05] developed a compiler technique to identify logical data structures in the program. Once a data structure has been identified, its elements are allocated in a designated memory pool. Defero works at the library level, thus it already has the rich information environment, without compiler support. Chilimbi et al. in [Chil 99a] investigated the idea of cache-conscious data structures. They present two tools for improving cache locality for dynamic data structures, namely *ccmorph* and *ccmalloc*. Chilimbi's tools are not available[5], so we could not empirically compare them against Defero. Nonetheless, Defero works automatically with containers, while Chilimbi's tools are semi-automated and require programmer intervention and hardware knowledge about the system. Ccmorph works only with static trees that do not change once created, while Defero does not impose this restriction. On the contrary, Defero offers several policies that benefit different dynamic behaviors.

HeapLayers [Berg 01] is an infrastructure for building memory allocators. Unfortunately we could not use HeapLayers since Defero's design requires a generic template

[5] Personal communication with the author

allocation scheme, a generic template memory partition, and a new allocation interface. Huang et al. [Huan 04] used the Jikes RVM adaptive compiler to sample the methods used by the program at run-time, and when they become ”hot”, they get compiled. This approach increases data locality by profiling at garbage time, while Defero improves locality through semantics at allocation time.

Grunwald et al. [Grun 93] profile the performance of five memory allocation schemes (first-fit, gnu g++, BSD, gnu local, quick-fit) from the reference locality point of view, and conclude that efforts to reduce memory utilization, such as coalescing adjacent free blocks, will increase both the execution time and reduce program reference locality, in most cases. Barret and Zorn [Barr 93] show that 90% of all bytes allocated are short-lived. Short-lived objects are placed in separate arenas, while long-lived objects are placed together. This implicitly increases the locality of short-lived objects. Shuf et al. [Shuf 02] show that prolific type objects could benefit from allocating them together, since they tend to be related and short-lived.

Seidl and Zorn [Seid 98] present an algorithm for predicting heap object reference and lifetime behavior at the time objects are allocated. Calder et al. [Cald 98] present a profile driven compiler technique for data placement (stack, global, heap and constants) in order to reduce data cache misses. Grunwald and Zorn [Grun 92] present a custom memory allocator based on profiling. Users profile an application, and based of the object sizes the application uses, customalloc creates segregated lists with the frequently used class sizes.

Doug Lea’s allocator, DL, uses boundary tags in its implementation and a size-segregation organization [Lea 90,Lea 96]. Boundary tags allow for quick coalescing, which improves data locality. Defero does coalescing at allocation, explicitly through the allocation predicate, while DL does coalescing at deallocation using boundary tags.

Gay and Aiken [Gay 88] study region based memory management. A pseudo region-based allocation is supported by Defero implicitly as a K-class region with the locality advantage but without the space wasting disadvantage. There are other memory allocation schemes worth mentioning which attempt to improve performance by speed [Vo 96] or by padding [Truo 98].

8 Conclusions

We presented Defero, a container-centric memory allocator, which allows higher level semantic information present in the allocation context to be communicated to the memory allocator. Defero is automatically integrated in STL, thus applications need not change a single line of code. This approach provides custom memory allocation for free.

Defero’s generic interface allows users to select their own partition and allocation policy, thus making Defero highly tunable. We analyzed a novel and adjustable K-Bit memory scheme. K-Bit performs best when it is virtual page aware and partitions its space into pages. We explored several allocation policies on several applications. We show that containers benefit from specialized allocation predicates. We propose a novel allocation predicate named Path, which produces better locality for balanced trees. It outperforms First and Best allocation predicates, and even Doug Lea's allocator. For list containers, First and Best produce the best locality improvement.

As future work, we plan on experimenting with different equivalence relations, such as cache set partitioning for reducing cache misses.

References

[Barr 93] Barret, D., & Zorn, B. (1993). Using Lifetime Predictors to Improve Memory Allocation Performance. *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Albuquerque, New Mexico, USA.

[Berg 02] Berger, E., Zorn, B., & McKinley, K. (2002). Reconsidering Custom Memory Allocation. *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*, Seattle, Washingthon, USA.

[Berg 01] Berger, E., Zorn, B., & McKinley, K. (2001). Composing High-Performance Memory Allocators. *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Snowbird,Utah,USA.

[Cald 98] Calder, B., Krintz, C., John, S., & Austin, T. (1998). Cache-Conscious Data Placement. *Proceedings of the ACM-SIGPLAN International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, San Jose, California, USA.

[Chil 99a] Chilimbi, T., Davidson, B., & Larus, J. (1999). Cache-conscious structure definition. *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA.

[Chil 99b] Chilimbi, T., Hill, M., & Larus, J. (1999). Cache-conscious structure layout. *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Atlanta, Georgia, USA.

[Ding 99] Ding, C., & Kennedy, K. (1999). Improving cache performance in dynamic applications through data and computation reorganization at run time. *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, Atlanta, Georgia, USA.

[Gay 88] Gay, D., & iken, A. (1998). Memory Management with Explicit Regions. *Proceedings of the ACM-SIGPLAN Conference on Programming language Design and Implementation, (PLDI)*, Montreal, Quebec, Canada, pp. 313-323.

[Grun 92] Grunwald, D., & Zorn, B. (1992). CustoMalloc: Efficient Synthesized Memory Allocators. *Technical Report CU-CS-602-92, Department of Computer Science, University of Colorado, Boulder*.

[Grun 93] Grunwald, D., Zorn, B., & Henderson, R. (1993). Improving the Cache Locality of Memory Allocation. *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, Albuquerque, New Mexico, USA.

[Huan 04] Huang, X., Blackburn, S., McKinley, K., Moss, J., Wang, Z., &Cheng, P. (2004). The garbage collection advantage: improving program locality. *Proceedings of ACM-SIGPLAN Conference on Object-Oriented Programming, Systems, Languages and Applications, (OOPSLA)*, Vancouver, British Columbia, Canada.

[Inte 98] International Standard ISO/IEC 14882. (1998). Programming Languages – C++. First Edition.

[Latt 05] Lattner, C., & Adve, V. (2005). Automatic Pool Allocation: improving performance by controlling Data Structure Layout in the Heap. *Proceedings of the ACM-SIGPLAN 2005 Conference on Programming Language Design and Implementation, (PLDI)*, Chicago, Illinois, USA.

[Lea 90] Lea, D. (1990, Jan/Feb). Some storage management techniques for container classes. *The C++ Report http://g.oswego.edu/pub/ papers/C++Report89.txt*

[Lea 96] Lea, D. (1996). A memory allocator. *Unix/Mail, Hanser Verlag http://gee.cs.oswego.edu/dl/html/malloc.html*

[Pola 01] Polaris - Compiler Optimization *http://parasol.tamu.edu/groups/rwergergroup/research/compilers/*

[Rus 03] Rus, S., Rauchwerger, L., & Hoeflinger, J. (2003) *Hybrid Analysis: Static & Dynamic Memory Reference Analysis In the International Journal of Parallel Programming*, 31(4), pp. 251283.

[Seid 98] Seidl, M., & Zorn, B. (1998). Segregating Heap Objects by Reference Behavior and Lifetime. *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems, (ASPLOS)*, San Jose, California, USA.

[SGI Web] SGI SGI STL Allocator Design. *http://www.sgi.com/tech/stl/alloc.html*

[Shuf 02] Shuf, Y., Gupta, M., Franke, H., Appel, A., & Singh, J. (2002). Creating and Preserving Locality of Java Applications at Allocation and Garbage Collection Times. *Proceedings of the ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Language and Applications (OOPSLA)*, Seattle, Washingthon, USA.

[Stan 80] Standish, T. (1980). *Data Structures Techniques.* Addison-Wesley Press.

[Toft 94] Tofte, M., & Talpin, J. (1994). Region-Based Memory Management. *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, (POPL)*, Portland, Oregon, USA.

[Truo 98] Truong, D., Bodin, F., & Seznec, A. (1998). Improving cache behavior of dynamically allocated data structures. *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques, (PACT)*, Paris, France, pp. 322.

[Vo 96] Vo, K. (1996). Vmalloc: A General and Efficient Memory Allocator. *Software Practice & Experience*, 26(3), pp. 357-374.

[Wein 76] Weinstock, C. (1976). Dynamic Storage Allocation Techniques. (Doctoral dissertation, Carnegie-Mellon University, Pittsburgh, April 1976).

[Wils 95] Wilson, P., Johnstone, M., Neely, M., & Boles, D. (1995). Dynamic storage allocation: A survey and critical review. *Proceedings of the International Workshop on Memory Management, (ISMM)*, Kinross, Scotland, UK, pp.1-116

[Wolf 91] Wolf, M., & Lam, M. (1991). A Data Locality Optimizing Algorithm. *Proceedings of the ACM-SIGPLAN Conference on Programming Language Design and Implementation, (PLDI)*, Toronto, Ontario, Canada.

[Wolf 87] Wolfe, M. (1987). Iteration Space Tiling for Memory Hierarchies. *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, Los Angeles, California, USA, pp. 357-361.