IMPROVING LOCALITY WITH DYNAMIC MEMORY ALLOCATION

A Dissertation

by

ALIN JULA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

August 2008

Major Subject: Computer Science

IMPROVING LOCALITY WITH DYNAMIC MEMORY ALLOCATION

A Dissertation

by

ALIN JULA

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	Lawrence Rauchwerger
Committee Members,	Marvin Adams
	Nancy Amato
	Jennifer Welch
Head of Department,	Valerie Taylor

August 2008

Major Subject: Computer Science

ABSTRACT

Improving Locality with Dynamic Memory Allocation. (August 2008)

Alin Jula, B.S., Babes-Bolyai University, Cluj, Romania;

M.S., Babes-Bolyai University, Cluj, Romania

Chair of Advisory Committee: Dr. Lawrence Rauchwerger

Dynamic memory allocators are a determining factor of an application's performance and have the opportunity to improve a major performance bottleneck on today's computer hardware: data locality. To approach this problem, a memory allocator must first offer strategies that allow the locality problem to be addressed. However, while focusing on locality, an allocator must also not ignore the existing constraints of allocation speed and fragmentation, which further complicate its design. In order for a locality improving technique to be successfully employed in today's large code applications, its integration needs to be automatic, without user intervention. The alternative, manual integration, is not a tractable solution.

In this dissertation we develop three novel memory allocators that explore different allocation strategies that enhance an application's locality. We conduct the first study that shows that allocation speed, fragmentation and locality improving goals are antagonistic. We develop an automatic method that supplies allocation hints from C++ STL containers to their allocators. This method allows applications to benefit from locality improving techniques at the cost of a simple re-compilation. We conduct the first study that quantifies the effect of allocation hints on performance, and show that an allocator with high locality of reference can be as competitive as one using an application's spatial feedback.

To further allow dynamic memory allocation to improve an application's performance, new and non-traditional strategies need be explored. We develop a generic

software tool that allows users to examine unconventional strategies. The tool allows users not only to focus on allocation strategies rather than their implementation, but also to compare and contrast various approaches.

To Deborah

ACKNOWLEDGMENTS

I would like to thank my adviser, Lawrence Rauchwerger, for the advice, help and opportunities he has provided during my graduate studies at Texas A&M. His advice to turn the problem upside-down and his empirical skepticism have permeated, and will stay with me from here on. Lawrence also entrusted me to lecture his undergraduate compiler course, an experience that I very much appreciate and value.

I would also like to thank Nancy Amato, for her constant help and support, the committee members, Marvin Adams and Jennifer Welch, for their time, effort and insightful comments, and Dan Quinlan from Lawrence Livermore National Laboratories for the great mentorship during my internship.

I would like to thank all my friends and colleagues in our research group, Julio Carvallo de Ochoa, Francis Dang, Tao Huang, Guobin He, Will McLendon, Ioannis Papadopoulos, Maikel Penning, Antoniu Pop, Steve Saunders, Timmie Smith, Lidia Smith, Silvius Rus, Nageswar Tagarathi, Gabriel Tanase, Nathan Thomas, Dongmin Zang, Hao Yu, and everybody in Parasol group for their invaluable scholastic and personal advice. They made my stay exciting.

And last but not least, I would like to thank my whole family, especially my wife Deborah, for their unconditional support throughout.

TABLE OF CONTENTS

CHAPTER Page

I INTRODUCTION 1

A. Motivation 1
1. Shortcomings of Current Memory Allocators 3
B. Overview of the Dissertation 4
C. Summary of Contributions 6

II BACKGROUND AND RELATED WORK 7

A. Background 7
B. Related Work 8
1. Locality Improving 8
a. Allocators 8
b. Compiler and Profile Driven 10
c. Region-Based 11
d. Context Friendly 12
e. Miscellaneous 13
2. Memory Allocation Libraries 15

III EXPERIMENTAL METHODOLOGY 17

A. STL Benchmarks' Description 17
1. 471.omnetpp 17
2. 483.xalancbmk 18
3. 447.dealII 19
4. Atlas 19
5. Kolah 20
6. Molecular Dynamics 20
7. Debruijn 20
B. Benchmarks' Memory Profile 21
1. Size Distribution 22
2. Memory Distribution 22
C. Experimental Setup 23

IV LOCALITY IMPROVING MEMORY ALLOCATION TECHNIQUES 26

CHAPTER Page

A. Locality Considerations for Memory Allocation 26
1. Locality Accuracy and K-regions 29
2. How to Automatically Select and Supply Allocation Hints from STL Containers 32
a. Lists 34
b. Trees 35
c. Vectors 35
d. Deques 36
e. Hint Limitations 36
B. Allocators 38
1. TP 38
a. Allocation & Deallocation 41
b. Performance Analysis 42
c. The Optimal Value for K 46
d. Pitfalls 46
2. Defero 47
a. Allocation & Dealocation 48
b. Varying Locality Precision 49
c. Performance Analysis 53
d. The Optimal Value for K 53
e. Pitfalls 54
3. Medius 54
a. Allocation & Deallocation 56
b. Performance Analysis 59
c. The Optimal Value for K 61
d. Pitfalls 62
4. Summary of TP, Defero and Medius 63
C. Design Challenges for Locality Improving Allocators 65
1. Speed Competes with Fragmentation 66
2. Locality Competes with Speed 67
3. Fragmentation Competes with Locality 70
4. Balancing Allocation Speed, Locality and Fragmentation 70

V HINT VS. HINTLESS: A COMPARATIVE STUDY OF LOCALITY IMPROVING ALLOCATION TECHNIQUES 73

A. Locality of Reference - Improving Locality without Hints . 74
B. Hints vs. Hintless 75
1. Performance Comparison 76

CHAPTER Page

2. Applications Comparison 78
3. Containers Comparison 79
4. Locality Accuracy Comparison 80
C. STL Usage 85
1. STL Containers' Memory Profile 85
2. Containers' Effectiveness at Providing Hints 88
3. Containers' Dynamism 90
D. Discussions 92
E. Summary 94

VI GENERIC MEMORY ALLOCATION 98

A. Generic Memory Management and Allocation 100
1. Attributes of a Block 100
a. Types of Attributes 101
2. Memory Management with Partitions 102
a. Equivalence Classes 102
b. Storing Blocks in Containers 103
c. Partitions 103
d. Recursive Partitions 103
e. Simultaneous Partitions 104
3. Memory Allocation 105
a. Prioritized Allocation 106
b. Non-Prioritized Allocation 106
c. Allocation Predicates 107
d. Allocation = Search + Predicate 108
B. A Formalism for the Generic Memory Management and Allocation Problem 110
1. Generic Management 110
2. Generic Allocation 110
a. Prioritized Allocation 111
b. Non-Prioritized Allocation 111
c. Deallocation 111
C. Designing TP, Defero, and Medius 112
1. TP 112
2. Defero 113
3. Medius 114
4. Other Strategies 114
D. Designing Existing Allocation Strategies 115

CHAPTER Page

1. Splitting & Coalescing . . . 115
   a. Boundary Tags . . . 116
2. Buddy Systems . . . 117
3. Regions . . . 119
4. Cartesian Trees . . . 119

E. Comparison . . . 121
F. Summary . . . 122

VII CONCLUSIONS . . . 124

A. Contributions . . . 124
B. Future Work . . . 126

REFERENCES . . . 128

APPENDIX A . . . 139

APPENDIX B . . . 145

APPENDIX C . . . 148

VITA . . . 154

LIST OF TABLES

TABLE Page

I STL benchmarks description 18

II Memory statistics for objects allocated from STL containers 21

III Benchmarks' size distribution 22

IV Benchmarks' memory distribution per size classes 24

V Characteristics of TP, Defero, Medius, dlmalloc, PHKmalloc, and Vam 30

VI TP's external fragmentation 45

VII Defero's internal and external fragmentation 50

VIII Medius' external fragmentation 62

IX Comparison of TP, Defero and Medius 64

X Fragmentation for TP, Defero, Medius, PHKmalloc and dlmalloc . . 67

XI Statistics of STL allocations 96

XII Statistics of STL allocations (cont'd) 97

XIII Search traversals types used in memory allocation 108

XIV Termination predicates 109

XV Allocation predicates 110

XVI Generic partitions of various allocation schemes 123

LIST OF FIGURES

FIGURE Page

1 Object size distribution 23

2 Memory distribution 23

3 Example where traditional memory allocation does not exploit the allocation context to improve locality 27

4 Hint selection for list 34

5 Hint selection for tree 34

6 Hint selection for vector 35

7 Hint selection for deque 35

8 TP - internal data structure 39

9 TP - Average number of K-regions checked per allocation 41

10 TP - execution time normalized to dlmalloc 43

11 TP - instructions normalized to dlmalloc 44

12 TP - L1 misses normalized to dlmalloc 44

13 TP - L2 misses normalized to dlmalloc 44

14 TP - TLB misses normalized to dlmalloc 44

15 Defero - internal data structure 48

16 Defero - as K increases, the average number of tree nodes traversed per allocation increases, reducing allocation speed. 51

17 Defero - execution time normalized to dlmalloc - lower is better . . . 51

18 Defero - issued instructions normalized to dlmalloc 52

FIGURE Page

19 Defero - L1 misses normalized to dlmalloc 52

20 Defero - L2 misses normalized to dlmalloc 52

21 Defero - TLB misses normalized to dlmalloc 52

22 Medius - internal data structure 55

23 Medius - average distribution of the K-regions from which allocations occurred 56

24 Medius - active memory range 57

25 Medius - average number of visited array entries 58

26 Medius - execution time normalized to dlmalloc (lower is better) . . . 59

27 Medius - instructions normalized to dlmalloc 60

28 Medius - L1 misses normalized to dlmalloc 60

29 Medius - L2 misses normalized to dlmalloc 60

30 Medius - TLB misses normalized to dlmalloc 60

31 Execution time for TP, Medius, Defero, PHKmalloc, and malloc normalized to dlmalloc's (lower is better) 63

32 Average relative hint location of an allocated block for Defero, Medius, TP, dlmalloc and PHKmalloc 68

33 Speed, locality and fragmentation for five allocators and several configurations (closer to the line is better) 71

34 Hintless execution time normalized to hint allocation (lower is faster) 76

35 Hintless issued instructions normalized to hint allocation (lower is faster) 77

36 Hintless L1 misses normalized to hint allocation (lower is better) . . 77

37 Hintless L2 misses normalized to hint allocation (lower is better) . . 77

FIGURE Page

38 Hintless TLB misses normalized to hint allocation (lower is better) . 77

39 Page accuracies for Hint allocation 78

40 Page accuracies improvement of Hint over Hintless allocation (Hint - Hintless) 78

41 Page accuracies with Hint allocation for different types of containers 80

42 Improvement of page accuracy of Hint over Hintless for different types of containers 80

43 Average page and cache accuracies for Hintless Defero, Medius and TP as locality precision varies 82

44 Average improvement of Hint over Hintless of page and cache accuracies for Defero, Medius and TP as locality precision varies . . 83

45 Size distribution for objects smaller than 128 byes for each container type 84

46 Percentage of instantiated STL containers 87

47 Percentage of allocations originating from each container type 87

48 Percentage of memory used by each container type 87

49 Size distribution per container type 87

50 STL containers' effectiveness per application 89

51 Effectiveness for STL deques, lists, vectors and trees 89

52 Dynamism of STL containers 91

53 Example of generic memory allocation and deallocation requests . . . 98

54 Defero's and Kingsley's memory management schemes 99

55 Example of recursive partition 104

56 Example of simultaneous partition 105

FIGURE Page

57 TP's, Defero's, and Medius' generic partitions 113

58 Coalescing 117

59 Binary buddies 118

60 Regions 119

61 Cartesian tree 120

62 Cache set attribute 139

63 Block's structure for recursive and simultaneous partitions 140

64 Container's interface 142

65 Generalized allocation algorithm and allocation predicate 144

66 Kingsley's allocator implementation in Allotheque 147

CHAPTER I

INTRODUCTION

A. Motivation

Dynamic memory allocation is one of the fundamental problems in Computer Science, and it has been an active research field since the 1960s. Throughout its development, memory allocation has been confronted with the issues imposed by contemporary programming languages and computer designs. In its early development, memory allocation has been solely concerned with allocation speed and memory consumption. Lately, as the disparity between processor speed and memory bandwidth has become a major performance bottleneck, memory allocation has been confronting a new concern: data locality.

Today's computers process data faster than they can be fetched. This gap between processing power and memory latencies is also known as the von Neumann bottleneck or memory wall[5, 80]. The gap has been widening since the early 1980s with the introduction of memory hierarchy[1]. The goal of the memory hierarchy is to exploit the principle of locality. This principle states that programs tends to reuse data and instructions they have recently used[29]. There are two types of locality: temporal spatial. Temporal locality states that recently accessed items are likely to be accessed in the near future, while spatial locality states that items whose addresses are near one another tend to be reference close together in time.

Memory hierarchy exploits the principle of locality in hardware. The faster, more

---

This dissertation follows the style of *ACM Special Interest Group on Programming Languages.*

[1] In spite of a recent slowdown of the processor speed due to shrinking distances, power consumption and heat, memory bandwidth still does not keep up with processing speed[9]

expensive and yet smaller memory levels are closer to the processor, while the slower, cheaper and larger levels are farther. On modern architectures the latencies while accessing data located in register, cache and TLB typically vary by an order of magnitude with each level[29]. Moreover, the average number of data access instructions in an application is 40% of the total number of instructions[29]. For these reasons, the performance of an application is heavily affected by its memory accesses, and the goal of locality improving techniques is to optimize them.

Various techniques address data locality, such as software or hardware prefetching, field reorganization, and compiler techniques. However, memory allocation has the opportunity to solve the locality problem at its roots: data placement in memory. In this dissertation we focus on explicit memory management techniques for C++. While automatic garbage-collection environments, such as Java's, address data locality by moving data and reclaiming unused memory, manual (or explicit) memory management environments, such as C/C++'s, address data locality by selecting memory blocks at allocation time, without the luxury of moving them at a later time. Hence, the latter is a more difficult task.

For this reason, designing a explicit locality improving allocator is a complex endeavor that must combine allocation speed, fragmentation and locality into one component. While statically allocated data benefit from a more compact data layout and a compile-time approach to optimize it, dynamically allocated data, in contrast, possess an irregular data layout with a more constrained run-time solution. An important aspect of a locality improving allocator is the flexibility to adjust locality parameters, for different architectures. Another crucial facet of a locality improving allocator is the ability to exploit locality on all fronts, at the cache, virtual page, and page clustering level. All these demanding constraints make the design and implementation of locality improving allocators a challenging assignment.

1. Shortcomings of Current Memory Allocators

Traditional allocators prioritize allocation speed and memory fragmentation and treat locality improving as a side effect, due to its high exploitation cost. For example, one of the best available allocators, Doug Lea's allocator[53, 52], also included in the Linux operating system, belongs to this category. Alas, some allocators further pick one trait at the detriment of the other. For example, simple storage segregation and buddy systems are among the fastest allocators available, but they trade-off a higher memory fragmentation[59, 48].

The existing allocators that employ locality improving techniques often have sub-optimal solutions that might fit particular cases, such as FreeBSD's allocator, with an address order sorted list of virtual pages and increased fragmentation[46]. Furthermore, a lack of adjustable locality parameters prevents them from investigating and selecting of best fit locality improving techniques.

As new parameters that enhance an application's performance enter the memory allocation stage, such as call site and thread identifier, their integration is ad-hoc or, sometimes worse, non-portable. The current approaches address dynamic memory allocation problem by its theoretical needs only, i.e. size of the requested block. In practice, however, this information is not enough to optimize performance. For example, it has been shown that unconventional parameters can successfully be integrated in memory allocation to enhance performance, such as call site and stack pointer[6, 64]. These approaches, however, have not been widely accepted yet, partly due to their integration efforts. As other parameters that enhance performance will be explored, their exploration and integration into software applications must be facilitated.

B. Overview of the Dissertation

The goal of this dissertation is to develop memory allocation techniques that improve data locality and reduce an application's execution time. Improving data locality is one of the major performance challenges for modern architectures and virtually every software application faces it.

In chapter II we provide some background on memory allocation, along with related work in locality improving and libraries aimed at developing memory allocators. We describe the experimental methodology, the hardware platform and the applications we use in our experiments in chapter III.

In chapter IV We develop three novel memory allocation techniques that are locality aware and incorporate address in their management. While these three allocators, namely TP, Medius and Defero, employ an adjustable address search approximation, they all differ in their prioritizes and strategies. Defero prioritizes block size over block location, and Medius and TP prioritize block location over size. We show that TP improves the execution time of seven large, real world applications by an average of 7% and up to 22%, while matching the memory fragmentation, when compared one of the best memory allocators in Doug Lea's allocator. TP also outperforms state-of-the-art FreeBSD's allocator by an average of 17%. Defero outperforms Doug Lea's by an average of 5%, while Medius matches its performance.

We also develop an automatic technique in which C++ Standard Template Library (STL) containers supply allocation hints to their allocators to improve data locality. Benefiting from this technique requires only a re-compilation of the application, without user intervention. We further show that the main characteristics an modern memory allocator, namely allocation speed, locality and memory fragmentation, compete with each other in a circular dependency, similar to the game of rock,

paper, scissors: efforts to increase one hurt the others.

The performance improvement obtained with our allocators is a cumulative effect of their strategies on the one hand and STL automatic hints on the other hand. To study their separate contribution, in chapter V we compare two scenarios: in one scenario allocators use the hints provided by STL containers and in the other one allocators ignore them. Our analysis shows that while the automatic hints provided automatically by STL containers increase virtual page collocation by 40%, their locality improving benefits are offset by their exploitation costs, resulting in a similar performance in both scenarios. This result shows that the overall performance improvement is mainly attributed to our allocators' strategies. The locality aware allocation strategies reward the deallocation operation, even when hints are ignored. This is because when returning a block into its partition, deallocation benefits from the fast address searching strategies.

To further allow for exploration of unconventional allocation strategies, we develop a software library framework in chapter VI. This library decomposes the generic memory allocation problem into three independent dimensions: 1) block attributes, 2) partitions used to store these attributes and 3) allocation predicates that guides the search for attributes in the partitioned space. The library is to memory allocation what STL is to programmers: it allows the users to focus on the allocations strategies rather than their implementation. Our generic tool allows memory management to consider attributes other than size, such as address (which we study in this dissertation), allocation call site, stack pointer (which have been shown to increase performance), and thread identifier for parallelism. Furthermore, it allows one to further research unconventional allocation strategies, as well as compare existing mechanisms. Finally, we summarize our contributions and conclusions and discuss future work in chapter VII.

C. Summary of Contributions

Our work has contributed to the field of explicit memory management by introducing three novel and adjustable mechanisms that improve locality by efficiently managing memory based on address. TP outperforms state-of-the-art memory allocators in terms of performance and matches the best memory allocator in terms of memory fragmentation. We develop an automatic technique in which C++ STL containers supply allocation hints directly to our allocators to improve spatial locality. This is the first automatic technique of this nature. Our work also presents two empirical studies. The first study presents the relations between allocation speed, locality and fragmentation in a memory allocator. The second study compares two locality improving techniques, with and without allocation hints. Finally, our work sets forth a generic software framework to enhance performance gains through exploration of unconventional memory allocation strategies.

The source code is available at the following URL:
http://parasol.tamu.edu/resources/downloads.php

CHAPTER II

BACKGROUND AND RELATED WORK

A. Background

A memory allocator must keep track of the memory blocks that are in use by an application (allocated) and the ones that are available for use (freed), minimizing time and memory usage in doing so. The interface for allocating a block of a certain size is `int*x = (int*) malloc(8)` for C and `int*x = new int` for C++, with the latter being type-safe. The interface for deallocating a block is `free(x)` for C and `delete x` for C++. For STL containers, the ISO C++ Standard recognized the importance of locality, and added a hint to the allocation interface `allocator<T>::allocate(size_t n, void* hint)`[1]. However, no selection technique is provided.

Maintaining the available blocks in a linked list is a popular mechanism. The blocks themselves are used to store the list's structure. Upon allocation, the list is searched for a fitting block. Some of the fitting criteria are first-fit (first large enough), best fit (smallest larger than requested size), and next-fit (similar to first-fit but starting from where last search left off). The unused portion within a block is known as internal fragmentation. Another popular mechanism, due to its speed, is size segregation: blocks within a size interval are linked in a list and stored in a hash table. In addition to these approaches, splitting and coalescing mechanisms are used for minimizing external fragmentation (the extra memory used by an allocator in addition to the memory used by the application).

B. Related Work

There is a significant amount of work in dynamic memory allocation. Wilson et al. provide an thorough survey of the literature [77]. Since the publication of their article in 1995, several new directions have emerged, including but not limited to region-based, cache-concious, probabilistic and locality improving techniques, which we describe shortly. We divide related work into two categories. The first category presents and compares the existing work related to locality improvement, including notable work in garbage collection environments. The second category surveys libraries designed to build memory allocators and draws a comparison with our generic framework.

1. Locality Improving

a. Allocators

Doug Lea's allocator, dlmalloc, is a widely-used malloc implementation that forms the basis of memory allocator included in the Linux GNU C library. It is considered one of the best overall memory allocators[53, 52]. Berger et al. show that dlmalloc competes with custom memory allocators, and sometimes even outperforms them[8]. Detlefs et al. study five allocators and show that Doug Lea's allocator is the best overall allocator, with best fragmentation, best execution time, and second fastest [22]. dlmalloc is an approximate best-fit allocator that groups blocks into four categories: small, less than 64 bytes, medium, between 72 and 512 bytes, large, between 512 and 128 KB and very large, larger than 128 KB. Each size group has a different policy. Small objects (less than 64 bytes) are stored in exact-size linked lists. dlmalloc tries to coalesce the small objects upon a request for a medium-sized object. For medium-sized objects, dlmalloc performs immediate coalescing and splitting and approximates

best-fit. For large objects, dlmalloc spaces them logarithmically into separate groups that are sorted by size. Within one of these groups, a first fit is the best fit. For very large objects, dlmalloc uses `mmap` and `munmap` [53, 52].

PHKmalloc is an allocator designed by Poul-Henning Kamp for the FreeBSD operating system[46]. In its pursuit of locality improvement, PHKmalloc does not use hints but rather exploits locality of reference. PHKmalloc has a page oriented design in which the heap is divided into 4KB pages. Blocks smaller than 2KB are rounded to the neareast power of two, while larger blocks are rounded to the nearest page multiple. In PHKmalloc, blocks within pages have the same size and their status is recorded in bitmaps, which are kept in a double linked list sorted in address-order. PHKmalloc's regions are stored in an address order sorted list, which is linearly traversed for every deallocation in search of the appropriate region.

Vam is an allocator designed by Feng and Berger, which refines PHK-malloc's fragmentation and uses a finer size segregation, with multiples of 8 bytes [23]. Vam's regions uses lists to manage their blocks, instead of bitmaps like PHKmalloc does. Vam does not sort its regions and does not use hints to improve locality, but rather exploits a region address division to increase its locality of reference. PHKmalloc emphasizes locality and speed in favor of fragmentation, while Vam reduces fragmentation and increases speed, but loses a bit in locality by not sorting its regions when compared to PHKmalloc.

Chilimbi et al. investigate collocating adjacent blocks in the same cache line at allocation time[15, 17]. They present two tools for improving cache locality for dynamic data structures, namely ccmorph and ccmalloc. Ccmorph rearranges trees in memory to reduce locality. This method is applicable to static trees only, which are allocated once and do not change over time and leaves our dynamic trees. Our work does not pose this constraint and is applicable to dynamic trees as well, which

we believe represent a large part of how trees are used. Ccmalloc places adjacent blocks in the same cache line, if possible.

Grunwald et al. profile the performance of five memory allocation schemes, first-fit, gnu g++, BSD-Kingsley, gnu local and quick-fit[27]. Their study shows that sequential-fits have a poor reference locality, while BSD and Quick-fit provide the best locality. The authors conclude that efforts to reduce memory utilization, such as coalescing adjacent free blocks, will increase both the execution time and reduce program reference locality. This work supports our findings presented in chapter IV that allocation speed, fragmentation and locality circularly compete with each other.

b. Compiler and Profile Driven

Lattner and Adve present a compiler technique based on pointer and escape analysis to identify logical data structures in the program[51]. Once a data structure is identified, its elements are allocated in a designated memory pool that improves locality for that data structure. Our work operates at the memory allocation level within STL, where it already has access to STL data structures. This approach works across different compilers and platforms and is applicable to all data structures, not only to the ones identifiable by the compiler as such. Calder et al. present a profile driven compiler technique for data placement (stack, global, heap and constants) in order to reduce data cache misses[10]. Their technique creates a temporal relationship graph that is used in the next runs to arrange highly temporal data in order to reduce the cache conflicts. Grunwald and Zorn present a custom memory allocator based on profiling, in which users profile an application to extract the used object sizes. These frequently used sizes are used for segregated lists in a custom allocator[26].

Huang et al. [38] used the Jikes RVM adaptive compiler to reorder objects based on their most common or "hot" traversal pattern, in order to improve locality.

Their approach has the advantage of knowing which data get traversed as well as the traversal order. Container dynamism, described in chapter V, deals with this issue and could inform allocators about which allocations are important. Shuf et al. [66] 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. In an effort to bridge the gap between garbage collection and C++ explicit allocation, Nishanov and Schupp present a case study of integrating a garbage collection into C++ STL [57], by creating adaptors between garbage collectors and containers.

c. Region-Based

Region-based memory allocation was first introduced by Tofte the for ML language [73, 72]. in which the memory is divided into fixed-size regions that are organized into a stack-like fashion. Conceptually, region-based memory allocation is very similar to our approach. The advantages of region-based are fast allocation by pointer-bumping, zero deallocation cost by deallocating a whole region at once and spatial locality for scope based allocation. The disadvantages are high fragmentation due to potentially unused portions of memory and limited use because of the scope based approach. Our approach does not impose a scope based allocation and works for imperative languages such as C++, without any language support. Moreover, our K-regions reduce the high fragmentation imposed by region-based allocation by allowing individual deallocations.

Gay and Aiken present the region-based memory management in which variables declared in the a syntactic scope are allocated in a contiguous memory region[24]. Upon exiting a scope its entire region is reclaimed. Cherem and Rugina extend the region-based memory management to Java programs through a compiler transformation that translates the program into an equivalent output program with region-based

memory management [12]. In a similar effort, Chin et al. automate the process of annotating a program to use regions for memory allocation [18] . The scheme is formally described using rules and lifetime constraints, and implemented for Core-Java, a core subset of Java. The explicit scope forces regions to follow variables closely when they get assigned to a different scope.

Chung and Moon present a hybrid memory allocator, between sequential fits and region-based [19]. Large regions are kept in a list and sequential fit strategies are employed for this list. Once a region is selected from the list, using a best, worst, first one next fit, the memory allocator uses region-based allocation within the region. This allocator improves data locality as well as speed over sequential fits.

Deters and Cytron use profiling to analyze the scoped memory regions for Java, by constructing a graph whose nodes are allocation sites and edges are possible "has references" at a different site [21]. This graph is used to associate virtual scopes with the allocation sites. This is an example of using unconventional attributes to improve performance through memory allocation, which can be implemented with our generic memory allocation library. Qian and Hendren allocate, by default, all objects on the stack, hoping the scope of the variables matches the syntactic scope [60]. If a variable survives the syntactic scope, and their experiments show that one in three variables does, it is placed in heap and its allocation site is switched to allocate its remaining variables in heap. An adaptive mechanism reduces the number of regions compared to region-based, as well as the fragmentation within the region.

d. Context Friendly

Previous work has used contextual information to improve performance through memory allocation, and the following work serves as concrete instantiation of the generic memory allocation we propose in chapter VI. Barret and Zorn present an algorithm

for predicting objects' lifetime and show that 90% of all bytes allocated are short-lived [6]. Objects are logically connected to program call-sites, which are used to place the short-lived objects in separate arenas. This work explores temporal locality and is complementary to ours, which explores spatial locality. The call site is used as a concrete and relevant attribute to the memory allocation performance.

Seidl and Zorn present an algorithm for predicting heap object reference and lifetime behavior at the allocation time[64]. Their profile-based approach considers various 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. This work is an example that provides several relevant attributes to memory allocation that can be used to improve performance.

Hirzel studies the locality benefits of 12 data layouts on 32 benchmarks in a garbage collection environment and shows that almost all layouts yield the best performance for some applications and the worst for others[33]. We believe that the wide performance variance of different data layouts reported by the author encourages explicit locality improving techniques, due to their ability to customize an application's data layout.

e. Miscellaneous

There are other memory allocation schemes that attempt to improve locality by various techniques such as instance interleaving [74], caching and coalescing [49]. Software pre-fetching has been used as a method for improving locality of dynamic data structures. Kaplan et al. present an adaptive mechanism that controls the number of pages to be fetched on a page fault, as well as which pages [47]. Aggarwal explores software caching for dynamic structures on which searches or sorted insertions are performed [2]. Hallberg et. al shows that software, hardware pre-fetching and cache

conscious allocation are competitive in improving locality for dynamic data structures [28].

Cheng and Ding show that the majority of programs exhibit the same reuse pattern regardless of inputs, with some programs showing high variability [11]. Chilimbi and Larus present a generational garbage collection scheme that improves data locality for small objects, based program profiling and construction of a temporal affinity graphs with weighted edges for temporal relation [16].

Prioritized allocation, which we propose in chapter VI, has been discussed before, though at a different level. Robertz proposes priorities for memory allocation in a Java system, without language support [61]. The priority based system preventively starts dropping the non-critical requests before the memory is all used, instead of starting the garbage collector as a non-priority system would normally do. The priorities we propose for memory allocation are attribute based and take place *within* the allocation process and not among allocations processes.

Chilimbi et al. propose a memory allocation trace format so that users can exchange these traces without running the applications [13] . The authors consider size, address,heap, thread, time and attributes - user defined. One of the problems they encountered was compacting large traces.

Larson presents a memory allocator designed for long running server applications, with an emphasis on scalability [50]. The authors conclude that it is not sufficient to focus on reducing lock contention - higher speedups require a reduction in cache misses and bus traffic. Kakkad et al. present a scheme for extracting run-time type description for objects allocated dynamically, less the local and static objects [45]. The run-time type description is used for garbage collection and persistent object storage.

Robinson shows the theoretical lower bounds of fragmentation[62, 63]. John-

stone and Wilson explore four ways of calculating fragmentation and evaluate the fragmentation of different allocation policies, such as best, first and next fit, address ordered, buddy, double buddy, segregated lists [40]. They conclude that fragmentation is minimal in all allocation policies. Hirschberg modifies the binary buddy-system management, from power of two's series to Fibonacci's, reducing fragmentation [32].

2. Memory Allocation Libraries

HeapLayers is an infrastructure for building memory allocators, designed by Berger et al., which uses overhead-free template mix-in technique to create a stack of layers[7]. This is the closest project to our generic memory allocation library. While we share the same goal, we take different approaches. HeapLayers provides twenty composition and three system layers that are used to build new allocators. This a *functional* and *concrete* decomposition of the problem. Our approach has a *conceptual* and *generic* decomposition of the problem in which we identify only three fundamental dimensions, namely block attributes, containers to store the attributes and allocation predicates to search them. We surmise that this 3-dimensional space is sufficient to formulate any memory allocation problem. When these dimensions take concrete types, a new allocator mechanism and strategy get instantiated. For example, consider allocating memory based on a new attribute, such as 'allocation sites' described by Barret et at. in [6] as a predictor for data locality. A simple description of the 'allocation site' attribute would allow one to incorporate it into their allocation strategies. To do the same thing in HeapLayers, one would develop a new layer that handles the allocation sites inside and which does not have the interface to communicate with the application, since the HeapLayers' interface is homogenous and has size as its only requested attribute.

Kiem-Phong Vo introduces the idea of organizing the memory into separate re-

gions, each with a discipline to get raw memory and a method to manage allocation[75]. 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 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 [27]. We use this mechanism for clustering K-regions.

Attardi et at. present a garbage collection framework, Customisable Memory Management (CMM), which allows different heaps to be managed by different strategies[4]. Unlike our work, theirs focuses on garbage collection and implicit memory management, which is complementary to ours as new allocation strategies can be integrated with their framework.

CHAPTER III

EXPERIMENTAL METHODOLOGY

In this chapter we describe the seven STL C++ benchmarks we used in our experiments, along with their memory profile. They are large, real world, and memory intensive applications that stress the memory hierarchy of most computer systems. They allocate large amounts of memory, up to 60GB, and use as much as 700 MB of memory.

A. STL Benchmarks' Description

The benchmarks we used in our experiments were chosen for their STL usage. Because our work targets applications written in C++ STL for reasons we detail in the next chapter, their availability is still limited given the STL's later arrival in the programming world, when compared to C or C++ [1]. We used three SPEC 2006 CPU benchmarks, namely 471.omnetpp, 483.xalancbmk and 447.dealII, along with a mathematical, a mesh transport, a molecular dynamics, and a network simulation application [68, 30, 31].

First, we detail the seven benchmarks and then we follow with their STL memory profile. Table I describes the number of files, the number of bytes, the number of lines of code, the input parameters we used, and their corresponding execution times in seconds for these benchmarks. They are large, real world and memory consuming applications, whose execution time ranges from 16 to 1,152 seconds.

1. 471.omnetpp

The first benchmark, 471.omnetpp, performs discrete event simulation of a large Ethernet network, which is based on the OMNeT++ discrete event simulation system,

Benchmark	Files	Bytes	Lines	Input Used	Execution Time (sec)
471.omnetpp	155	1,352,414	47,910	ref	1,152
483.xalancbmk	1,773	18,616,140	553,643	ref	841
447.dealII	452	7,160,611	198,649	ref	1,040
Atlas	230	1,325,699	53,869	type B8 sc s wgraph 7	557
Kolah	227	1,745,754	52,921	ellipsoid.pdb	70
Md	30	34,696	1,259	20 0.45 1.6 5	64
Debruijn	3	13,463	499	60,000 24 5	15

Table I. STL benchmarks description

a generic and open simulation framework. OMNeT++'s primary application area is the simulation of communication networks, but its generic and flexible architecture allows for its use in other areas such as the simulation of IT systems, queueing networks, hardware architectures or business processes as well [68]. This benchmark "... encourage[s] the OS to speed up malloc and may encourage optimizers to perform malloc optimization." [78]

2. 483.xalancbmk

The second benchmark, 483.xalancbmk, is an XSLT processor for transforming XML documents into HTML, text, or other XML document types. The program is a modified version of Xalan-C++, an XSLT processor written in a portable subset of C++. Xalan-C++ version 1.8 is an implementation of the W3C Recommendations for XSL Transformations (XSLT) and the XML Path Language (XPath). This benchmark is a "... large e-business app that uses STL with very large data input set. It will push

most stack memory to the limit while encouraging malloc improvement.” [78]

3. 447.dealII

The third benchmark, 447.dealII, uses deal.II, a C++ program library targeted at adaptive finite elements and error estimation. The library uses state-of-the-art programming techniques of the C++ programming language, including the Boost library. The main aim of deal.II is to enable development of modern finite element algorithms, using among other aspects sophisticated error estimators and adaptive meshes [68]. This benchmark ”... uses Boost libraries and complex template techniques. Best representative of future C++ directions.” [78]

4. Atlas

The fourth benchmark, Atlas, is a mathematical application that was developed by the Atlas of Lie Groups and Representations collaborators and makes extensive use of STL containers and algorithms. The application is memory intensive, being limited only by the amount of available memory in the system, up to 128 GB[1]. Atlas computes ”... one of the most complicated structures ever studied, the object known as the ’exceptional Lie group E_8’ ...”, whose complexity ”... invites comparison with the Human Genome Project”, according to American Institute of Mathematics [3]. The application computes ”... subtle ways in which different equations or geometric shapes can be seen as facets of the same underlying thing - an insight that has led to some of the century’s biggest discoveries in particle physics and may play a role in future theories”, according to *Scientific American*, March, 2007 [41].

[1] Personal communication with the authors

5. Kolah

The fifth benchmark, Kolah, is a mesh transport framework that analyzes numerical methods on arbitrary polygonal and polyhedral meshes. The code was developed at Lawrence Livermore National Laboratories. Within this framework, we have analyzed a benchmark that solves the Euler equations using the Lagrangian method and an ideal gas law equation of state.

6. Molecular Dynamics

The sixth benchmark, Md, is a molecular dynamics application that computes the molecular interactions between physical particles in a time step algorithm. The code was developed by Danny Rintoul of Sandial National Laboratories. The code fully utilizes STL containers, such as lists, trees and STL algorithms such as `for_each`, `sort` and `inner_product`.

7. Debruijn

Finally, the seventh benchmark, Debruijn, is a simple yet STL intensive network simulation micro-kernel, which was developed by Derek Leonard and Dmitri Loguinov of the Department of Computer Science at Texas A&M University. The application uses a Debruijn graph[2] to simulate network behaviors under random circumstances, such as node failures.

---

[2]A Debruijn graph has $O(N)$ vertices and an average of $O(\log N)$ edges for each vertex

Application	Total Objs.(10^3)	Max Objs. in Use(10^3)	Average Size(B)	Total Mem. Req.(MB)	Max Mem. in Use(MB)	Total/ Max
471.omnetpp	269.8	27.6	39.4	10.1	0.9	11.2
483.xalancbmk	85,179.8	365.7	702.4	57,063.5	196.3	305.9
447.dealII	144,637.8	13,196.9	38.4	5,300.1	327.7	17.7
Atlas	156,628.7	2,482.1	96.8	14,465.8	695.8	22.1
Kolah	11,732.3	4,313.4	94.7	1,100.8	672.4	1.64
Md	12,540.3	49.9	20.0	239.5	16.1	18.3
Debruijn	428.7	60.2	84.3	34.4	15.9	2.2

Table II. Memory statistics for objects allocated from STL containers

B. Benchmarks' Memory Profile

Table II shows the total number of allocated objects, maximum number of live objects, average object size, total memory requested by STL, maximum live memory by STL containers, and the ratio between the total and maximum live memory. Atlas and Kolah use the largest amount of memory, with almost 700 MB, while 483.xalancbmk and 447.dealII also use several hundreds MB of memory for their STL containers. These benchmarks are memory intensive, allocating from 10 MB to 57 GB of memory. Their memory recycling factor - the ratio of total memory over maximum memory in use - also shows a high interaction between STL containers and their allocators, varying from from 2 to 305. This means that every byte used by 483.xalancbmk was allocated and deallocated an average of 305 times. STL containers allocate mostly small objects with an average size of less than 100 bytes, except for 483.xalancbmk, which had an average of 702 bytes.

Application	0-128 (10^3)	128-512 (10^3)	512-1KB (10^3)	1-4KB (10^3)	4-16KB (10^3)	>16KB (10^3))
471.omnetpp	269.9	-	-	-	-	-
483.xalancbmk	44,515.1	10,242.3	9,699.4	18,668.3	2,054.7	0.05
447.dealII	137,114.0	6,821.5	690.6	63.1	0.6	2.7
Atlas	155,303.2	432.8	65.2	199.9	466.2	161.4
Kolah	11,548.3	183.8	0.01	0.02	0.02	0.3
Md	81,474.6	-	-	-	0.0	0.07
Debruijn	360.1	68.5	0.05	0.05	-	0.0

Table III. Benchmarks' size distribution

1. Size Distribution

Table III shows the applications' object size distribution, arranged into six groups: 0-128 bytes, 128-512 bytes, 512-1024 bytes, 1-4 KB, 4-16 KB and larger than 16 KB. Fig. 1 shows the same size distribution as a percentage from the total number of objects for each application. Except for 483.xalancbmk, which has 50% of objects smaller than 128 bytes, at least 94% of objects are small. On average, 90% of the total allocated objects were small, less than 128 bytes, and 99.9% less than 16 KB.

2. Memory Distribution

Table IV shows the memory amount allocated within each of the six size groups for each application. Fig. 2 shows the same memory distribution as a percentage of the total amount of memory allocated by each application. Even though more than 99% of the objects allocated in Atlas are small, they only account for 22% of the total

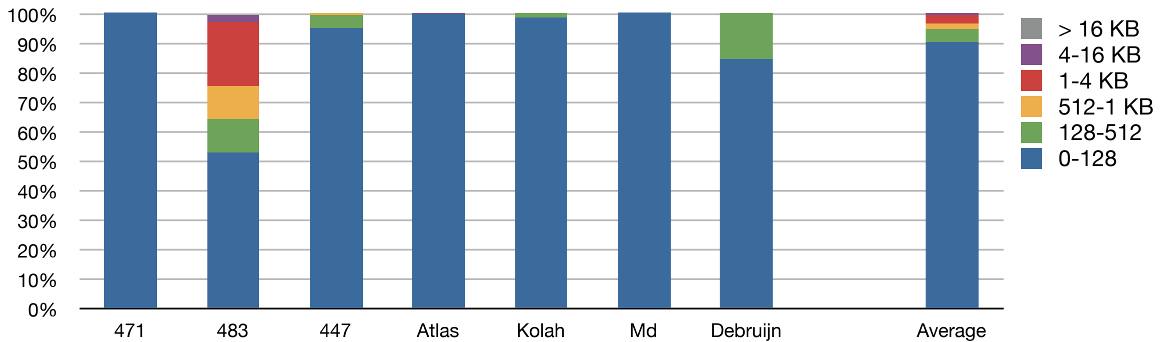


Fig. 1. Object size distribution

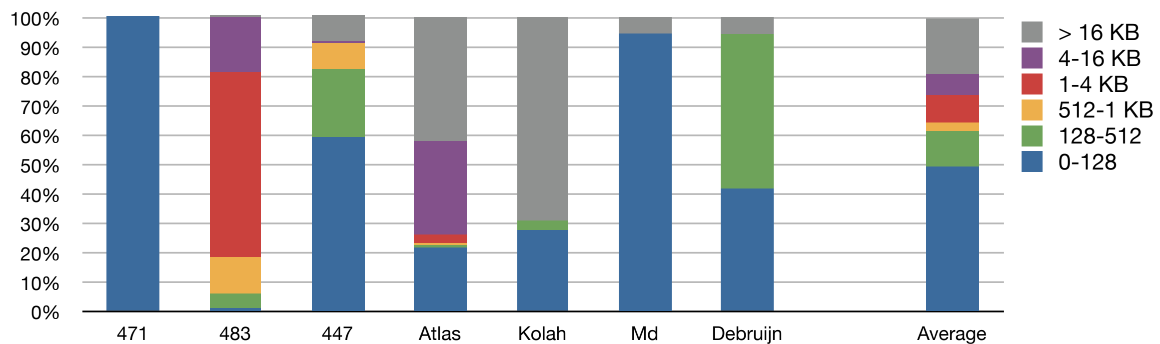


Fig. 2. Memory distribution

memory. A more extreme example is 483.xalancbmk, where 52% of the objects are small, but they only account for 0.83% of the total memory. Medium objects between 1 to 4 KB account for the majority of the memory allocated in 483.xalancbmk. On average 49% of the total memory was allocated for objects less than 128 bytes and 81% for objects less than 16 KB.

C. Experimental Setup

We compare our allocators to three state-of-the-art allocators that exploit the locality of reference without hints. The first one is the Lea allocator, dlmalloc, version 2.8.1. The second allocator is the FreeBSD operating system's allocator, PHKmalloc, version 1.89, which was designed by Poul-Henning Kamp [46]. The third one is Vam, packaged with HeapLayers version 3.4.0, was designed by Feng and Berger as

Application	0-128 (MB)	128-512 (MB)	512-1K (MB)	1-4K (MB)	4-16K (MB)	>16K (MB)
471.omnetpp	10.14	-	-	-	-	0.00
483.xalancbmk	513.342	3,072.64	7,250.16	37,669.52	11,228.29	323.28
447.dealII	3,404.11	1356.44	495.42	13.89	21.43	511.35
Atlas	3,294.92	189.01	49.32	468.10	4,877.02	6,517.93
Kolah	296.06	40.84	0.01	0.05	0.21	763.70
Md	327.74	-	-	-	0.02	19.43
Debruijn	14.65	18.80	0.03	0.08	-	2.06

Table IV. Benchmarks' memory distribution per size classes

a locality of reference improving allocator[7, 23].

The experiments were conducted on a Linux Intel Xeon CPU 3.00GHz processor with 1 GB of memory, 1 MB L2 cache, with 64 byte L1 and L2 cache line sizes and a virtual page size of 4 KB, using the GNU `g++` compiler, version 4.1.2. SPEC 2006 CPU benchmarks were compiled with `base` flags, which included an O2 optimization level. As a result, all our allocators used in STL were also compiled with the same O2 optimization. To ensure fairness, we compiled all allocators and applications in this dissertation with the same O2 optimization level.

All the experiments were executed three[3] times, and the average was reported. The standard deviation was less than 1%. We used PAPI, a performance monitoring tool to measure five hardware parameters that we considered relevant for locality purposes: CPU cycles and issued instructions to measure speed, and L1 misses, L2 misses and TLB misses to measure locality[39]. We used the native `gettimeofday` to monitor the applications' overall execution time.

[3]Md was executed six times

CHAPTER IV

LOCALITY IMPROVING MEMORY ALLOCATION TECHNIQUES

In this chapter we present three novel allocators, named TP, Defero and Medius, that use these hints to search for a free block within their vicinity. While the three allocators share the same goal, they use different strategies. They are the only documented memory allocators able to manage and satisfy memory requests based on address. We first discuss locality considerations for memory allocation and propose an automatic solution for providing allocation hints directly from STL containers. We then describe the allocators' design and implementation and show that they outperform state-of-the-art allocators, such as Doug Lea's dlmalloc and FreeBSD's PHKmalloc on the seven benchmarks we study.

A. Locality Considerations for Memory Allocation

Traditional memory allocation schemes, such as C++ memory allocator `malloc`, are designed to satisfy a memory allocation requests based on only one parameter, block size. While, indeed, size is the only parameter needed to answer the *theoretical* memory allocation problem, the *practical* problem, such as locality or parallelism, benefits from other parameters, such as allocation hints or thread identifier. These practical problems arise in memory allocation due to the computer hardware design and programming language paradigm. In a thought experiment, if one were to ignore the address virtualization and memory hierarchy, and assume a flat, constant access memory system, then locality issues would not exist.However, in practice both mechanisms exist.

For example, consider the code example in fig. 3, where 10 elements are inserted in two lists in an interleaved sequence. The STL allocator does not differentiate be-

```
std::list<int> a,b;
...
for(int i=0;i<10;++i) {
        a.push_back(i);
        b.push_back(i); }
```

Fig. 3. Example where traditional memory allocation does not exploit the allocation context to improve locality

tween the two allocation streams as coming from two distinct containers and sees only a series of 20 allocations with the same size. This happens in spite of the fact that STL allocators have the container element type as template parameter, and could conceivably differentiate between two different element types, such as `std::list<int>` and `std::list<double>`. Most allocators would pick up on the same size parameter, and, at best, would allocate the two lists' elements interleaved in a contiguous block, or, at worst, would allocate them in the order in which the blocks are deallocated by the application. Either solution does not provide the optimal data layout for the two lists, which is to have each list in a contiguous block[1]. Without additional support from compilers or such, a traditional allocator cannot distinguish which allocations originate from which container. However, if it did, then a memory allocator could allocate each container's elements in a cluster to improve containers' locality and an application's performance. This chapter presents a practical solution to this problem.

We refer to locality improving memory allocators that, regardless of their own locality, improve the *application's spatial locality* and reduce the overall execution

[1]One could argue that interleaving might actually increase locality, given a fitting traversal pattern, but more often then not, clustered containers, such as vectors, serve locality better.

time. The application's temporal locality cannot be affected by allocators[2]. Allocators are *run-time* algorithms whose performance directly participates in an application's performance. Inherently, both allocator's temporal and spatial locality contribute to the over application's performance[29].

There are two approaches for a memory allocator to improve data locality: 1)*implicitly* and 2)*explicitly*. Most of the previous work has been in implicit locality improvement. In this approach an allocator improves its own temporal - re-allocating recently deallocated blocks - and spatial locality - allocating blocks from the same memory region. Such allocator expects to overlap its temporal locality with the application's, leading to both temporal and spatial locality improvement. This is the most common approach, employed by various allocators, such as Doug Lea's dlmalloc, Feng's Vam and FreeBSD's allocator[53, 23, 46]. Its most appealing trait is the plug-and-play integration by replacing the traditional `malloc`. Its drawback is that this approach does not consult the application for specific locality needs, but rather uses a "one size fits all" allocation strategy.

In the explicit approach an allocator searches for a block within a neighborhood provided as a parameter in the form of allocation hint. This is the approach employed by Chilimbi's ccmalloc and Truong's ialloc[15, 17, 74]. The most appealing trait is that this approach uses an application's hints to increase its spatial locality. Its drawback is that it requires an additional parameter, the address hint. The shortcomings of the current techniques are threefold: 1) the hints require to be delivered manually, 2) the collocation is limited to cache line[15, 17], and 3) code restrictions are imposed[74]. These shortcomings make the current techniques impractical and error-prone for large

[2]An allocator can pollute the cache and TLB during its operation and thus interfere with the application's temporal locality. However, only code restructuring tools, such as compilers, can change an application's temporal locality.

applications.

Both approaches, implicit and explicit, have merits and drawbacks. In this dissertation, we first develop techniques that benefit both, and then compare them. For the implicit approach, our techniques improve deallocation and allocation speed, while offering fast temporal locality. For the explicit approach, our techniques allow fast address based allocation, offering efficient spatial locality. Table V summarizes the allocation speed, locality, internal fragmentation, memory reuse, region clustering and strategy of our three explicit locality improving allocators, along with three state-of-the-art implicit locality improving (a.k.a. locality of reference) allocators, namely dlmalloc, PHKmalloc and Vam.

1. Locality Accuracy and K-regions

We denote by *locality accuracy* the distance in the virtual address space between the hint and the returned block, upon successfully satisfying a memory request for locality. This locality accuracy quantitatively describes the "closeness" in spatial locality [29]. For example, an allocator that guarantees a locality accuracy of 64 bytes collocates a new block at an address that is at most 64 bytes apart from the hint's address. For our experimental platform, a 64 byte locality accuracy corresponds to allocating a block within the hint's cache line, or *cache-conscious collocation*, while a 4,096 byte locality accuracy corresponds to allocation within the hint's virtual page, or *page-conscious collocation*.

We generalize the locality accuracy and divide the memory address space into contiguous regions. Each such region contains all addresses that share the most significant (32-K) bits, i.e. differ only in their least significant K bits, with $K \in$

Allocator	Allocation Speed	Locality	Fragmentation		Region Clustering	Strategy and Complexity (R = number of regions)
			Internal	Reuse		
with hints						
TP	high	medium-high	medium	yes	no	Location O(1) or Size O(1)
Defero	medium	high	low	no	medium	Size O(1) then Location O(log R)
Medius	medium	high	low	no	high	Location O(1) then Size O(R) worst case
w/o hints						
Lea	medium	low-medium	low	yes	low	Size O(1) + splitting and coalescing.
PHK	medium	medium	high	yes	high	Size O(1)+ locality of reference. Deallocation O(R)
Vam	medium	medium	low	yes	no	Size O(1)+ locality of reference

Table V. Characteristics of TP, Defero, Medius, dlmalloc, PHKmalloc, and Vam

$[0, 32^3]$. We refer to such a region as K-region in short. For example, 6-regions are $2^6 = 64$ bytes and 12-regions are $2^{12} = 4,096$ bytes. Collocating a new block within the hint's K-region guarantees a 2^K locality accuracy.

However, not every 64 byte memory block is a 6-region. In addition to being 64 byte long, a memory block need be aligned by 64 bytes to be classified as a 6-region. For example, a memory block starting at address 0x0000000F and ending at 0x0000004F has a size of 64 bytes, but it is not a 6-region. This is because not all of its addresses differ in their least significant 6 bits: address 0x0000000F and address 0x00000040 differ in their least significant 7 bits. Similarly, not every 2^K byte memory block is a K-region. To be classified as K-region, such blocks also need be aligned by 2^K bytes[4].

Finding memory allocation techniques that quickly locate a block close to an address hint is the main challenge for improving locality. Our techniques use address approximation, i.e. locality accuracy, when searching for a block of a certain address, in the same manner as traditional size segregation schemes use size approximation, i.e. size segregated classes, when searching for a block of a certain size. A precise search is more accurate, but also more expensive. This in turn would reduce or maybe even negate the locality benefits. Finding a block with a certain locality accuracy can be done faster, and still yield good locality.

In our approach, we use K-regions to approximate spatial vicinity. The idea is to find an available block in the hint's K-region. This collocation guarantees a distance of less than 2^K between the hint and the newly allocated block. When K is 6, this leads to cache line collocation. Similarly, when K is 12 it leads to page collocation.

[3]Without loss of generality we assume a 32-bit system.

[4]Cache lines and virtual pages are also aligned by their size in memory

We also vary the locality accuracy, i.e. K, between 0 and 32, which allows us to study its effect on performance. At a first glance, it may seem irrelevant to study K's variance beyond the values corresponding to the cache line and page size, since these are the basic values that affect locality in a hierarchical memory. However, a further analysis reveals that other values of K also improve locality. For example, collocating two blocks in a 13-region, which includes two virtual pages in our system, produces a 50% probability that the blocks are in the same page. Thus, other values affect the *probability* for cache or page collocation. Large values of K have also an effect on locality improving, similar to the "wilderness preservation" concept[75]. Allocating from the same large K-region reduces the interval of virtual memory over which the application spreads. This reduces the memory footprint and increases an application's locality.

2. How to Automatically Select and Supply Allocation Hints from STL Containers

There are two ways in which allocation hints are supplied to a memory allocator: 1) manually by users and 2) automatically by automated tools, such as compilers or libraries. The manual approach is customizable and allows users to make the most of their knowledge in supplying the hint addresses, even in the most complex data structures. This is the approach used in [74, 14]. We used the manual approach to supply hints into the Olden Benchmarks, which is a collection of 10 C++ applications with approximately 10,000 lines of code[43, 58]. The process of inserting allocation hints into the code took us approximately one week. The challenge in this process was to detect the hint addresses that favored the most dominant traversal pattern in the program, which required an understanding of the code and its data structures before choosing the hint. The lesson we learned is that despite its flexibility in hint selection, the manual process, however, is impractical, tedious and error prone. As the

size of the code grows, its manageability reduces, and the manual approach becomes intractable in both its design and implementation. Moreover, the manual approach might not be feasible for existing applications, due to lack of code knowledge.

For this reason, we believe that the only viable solution to supply allocation hints to a memory allocator must be automatic. Next, we propose such an automatic technique to supply allocation hints directly to the memory allocator. We propose that C++ Standard Template Library containers provide allocation hints directly to their allocator, without user intervention. The 1998 C++ ISO Standard added an allocation hint parameter to the STL allocator's interface[1], but to the best of our knowledge no STL implementation uses this feature yet. STL is a C++ standard library that offers containers, iterators, algorithms, and functors as basic blocks of generic programming [1, 56, 71]. STL contains the most common data structures, such as vector, list, deque, set or map, which are designed as generic containers with template parameters and high level of reuse. STL increases productivity by reusing code and eliminating programming errors. Its biggest issue - error messages involving templates that tend to be very long and difficult to decipher- will be addressed in the future with the introduction of concept checking [67, 25].

In our technique, each STL container provides its allocator with its own hint address. Applications use STL containers without knowledge of their allocators, which consequently keeps them portable. When allocation hints are used from STL containers, the result is *custom memory allocation for free*[42]. It is custom because specific contexts provide their hints directly to their allocator. It is free because there is no user intervention. STL containers are perfect candidates to provide allocation hints directly to their allocators. First, the containers allocate their data dynamically, using allocators that are selected upon their instantiation. Second, the containers know their own elements' logical layout and can supply the appropriate hints directly to

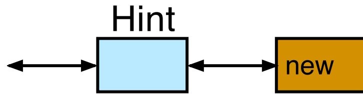


Fig. 4. Hint selection for list

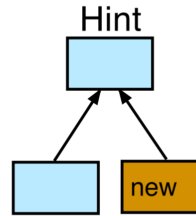


Fig. 5. Hint selection for tree

their memory allocator without user intervention. In essence, a container provides the "togetherness" concept used to describe spatial locality. Each STL container has a different semantic and requires a different hint selection.

The strategy for hint selection is to collocate containers' elements that are likely to be accessed together. The "togetherness" concept is dictated by the traversal patterns impose by STL algorithms and containers' methods. Most STL algorithms have a linear traversal pattern, predicated by the iterator trait. For example, algorithms that take forward iterators as input, such as `count(forward_iterator begin, forward_iterator end, value_type val)`, use only the `operator ++` defined by each container type as to reach the "next" element in a container. This operator implies a linear traversal. Very few algorithms, such as `sort`, use random access iterators that no longer imply a linear traversal, but rather a random one. The second type of traversals on STL containers is performed by their own methods, such as `set`'s `find`, and these methods are particular to each container. We next discuss how we select the allocation hints for the four types of containers that cover the implementation of all standard STL containers: lists, trees, deques and vectors.

a. Lists

Traversals of list containers imply a linear order of their elements. Elements in these containers are stored in a partial order. Because of this intrinsic invariant property

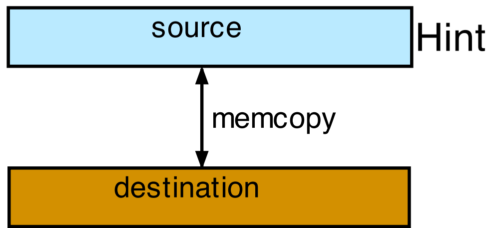


Fig. 6. Hint selection for vector

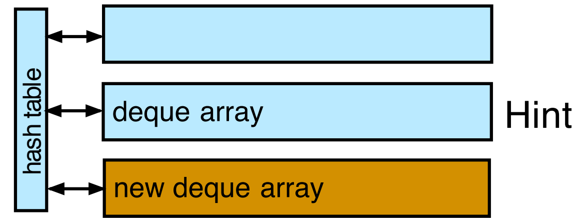


Fig. 7. Hint selection for deque

of the list, the best place to allocate a new element is in the immediate proximity of its next or previous elements. Thus, for lists, we selected the allocation hint to be the address of the previous element, as depicted in fig. 4.

b. Trees

Trees[5] are more complex than lists. One can think of several potent hint addresses, such as parent's and sibling's address, and path isomorphism[42]. Each one affects the spatial data layout in a different way. Depending on how the container is traversed, one might offer a better spatial locality than others.

We selected the parent's address as the default hint address, as depicted in fig. 5. The reason is because the STL trees are traversed in two ways: i) depth-first search, such as forward iterators on the tree nodes, as in `for_each( container.begin(), container.end(), foo())`, and ii) top-down searches, such as `set::find(elem)`. The hint to parent favors both traversal types.

c. Vectors

Vectors have a contiguous block on memory where its elements are stored, and thus there is nothing its allocator can do to improve the locality of its elements. However,

[5]Currently, the GNU STL 4.1.2 uses trees to implement these containers. However, the standard does not specify their implementation.

there are a couple of vector operations that could benefit from increased spatial locality, namely the copy constructor and the `operator =`. The rationale is that the memory copy of the source to the destination container, which is invoked in both vector's methods, benefits from their spatial collocation, as depicted in fig. 6. Other operations, such a copy out when passing vectors as parameters by value, also benefit from having the copy close to the original.

Another method to improve vectors' locality is the inter-container locality, which deals with how the vectors are laid in memory in relation to other vectors. A similar method was employed by Wu et. al to determine the relationships between nested Java containers for parallelism purposes, but this method lays outside the scope of our dissertation [79].

d. Deques

Deques are data structures for which elements can be added to or removed from the front or back[35]. STL deques are implemented as a hash table of fixed sized vectors, known as deque arrays. When a deque is traversed from its beginning to its end, its deque arrays are traversed in the order in which they are stored in the hash table. It is the location of these fixed size deque arrays that can improve deque's spatial locality. We use the neighboring deque array's address as hint when a new array gets allocated, as depicted in fig. 7. This strategy puts the neighboring arrays close to each other and increases their spatial locality.

e. Hint Limitations

The hint selection methodology we propose above leaves out a specific category of hints, namely hints pointing to elements of different sizes. These hints were omitted because of two reasons: 1) their low participation in data locality improvement

within STL containers and 2) their inclusion sacrifices allocation speed in our locality improving allocators.

The argument for the first reason is that data structures are homogenous collections of elements of the same size. The goal of our work is to improve the data locality within a dynamic containers, thus consequently, between the same sized elements. STL lists and trees allocate exclusively same sized blocks, and thus hints within these two containers point exclusively to same sized blocks. STL deques and vectors though allocate blocks of different sizes. For deques, the hash table that holds the deque arrays is allocated in the heap and grows as the the number of elements increases. This growth triggers a new allocation for a hash table larger than the previous one - twice as large in most STL implementations - followed by a copy of the old hash table into the new one and the deallocation of the old one. The allocation of the new hash table could use the address of the old hash table as hint. However, this operation is rarely used and is not a dominant operation. For vectors, when grown one element at a time, most of STL implementations, including the one we used, employ a similar mechanism to the one used to grow the deque's hash table: when a vector outgrows its afferent memory, a new allocation for twice as much memory is triggered and the contents of the old vector is copied into the new vector, after which the old vector gets deallocated. As with deque's hash table growth, this operation is not dominant and occurs $\log N$ times, where N is the number of elements in the vector. On the other hand, vector and deque traversals are the dominant operations that benefit from improved data locality.

The argument for the second reason is that in explicit memory management schemes, unlike garbage collected environments, efforts to manage adjacent but different sized blocks, using mechanisms such as coalescing and splitting, a leads to an allocation speed and locality deterioration[27]. We detail this argument in the next

sections.

B. Allocators

1. TP

The first allocator we present, named TP, is a locality improving memory allocator that emphasizes speed and locality. TP tries to collocate a block within the hint's K-region. If the hint's K-region is not available, TP reverts to the traditional allocation based on size.

TP starts with a simple storage segregation[59]. Unlike general size segregation schemes, where a bucket could have blocks of different sizes that are searched for the requested size, the simple segregation storage maintains blocks of equal size. A memory allocation request is rounded up to the nearest size class that is larger than the requested size, and the first block in that bucket is returned. For example, assume a size class of $(8, 16]$, which holds only blocks of 16 bytes. Then a request for 13 bytes is rounded up to 16 and the first block from this size class is returned. This scheme favors allocation speed over internal fragmentation.

TP uses a progressive size segregation[6] to round up the blocks' sizes:

- 0-512 : 64 size classes 8 bytes apart, (0,8], (8,16], . . ., (504, 512].
- 512-1,024: 8 size classes 64 bytes apart, (512,576], . . ., (960, 1024] .
- 1 - 2 KB: 8 size classes 128 bytes apart, (1024, 1152], . . ., (1920, 2048].
- 2 - 4 KB : 8 size classes 256 bytes apart, (2048, 2296], . . ., (3840, 4096].

[6] A size segregation mechanism clusters objects of the same size together, possibly increasing locality between size classes, but not within a size class.

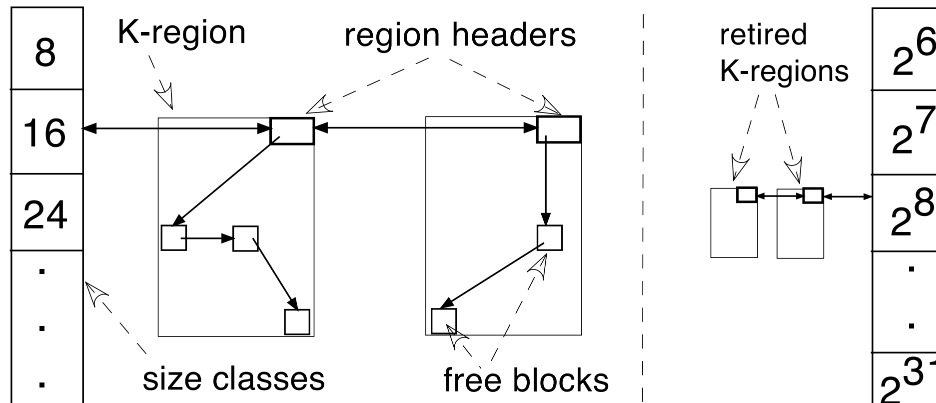


Fig. 8. TP - internal data structure

- 4 - 8 KB : 8 size classes 512 bytes apart, (4096, 4608], ..., (7680, 8192].
- 8 - 16 KB: 8 size classes 1,024 bytes apart, (8192, 9216], ..., (15360, 16384].
- larger than 16384 - handled directly by the operating system's `mmap` and `munmap`.

TP uses K-regions for address location purposes. TP acquires a K-region from the operating system and splits it into same sized blocks. All K-regions that hold blocks of the same size are linked into an unsorted list, which is stored into the hash table, as depicted in fig 8(left). The right side of the picture is detailed on page 42. A K-region links its same sized blocks into an unsorted linked list as well. This list's header is stored in the reserved topmost 16 bytes of a K-region, along with three other items: block size in that region, previous and next pointers for linking K-regions in a double linked list. We refer to the topmost 16 bytes in a K-region as the *header* of that K-region.

To provide constant time access to a K-region given a hint address within that region, TP masks the hint's most significant (32-K) bits to calculate the K-region's header. For example, if 0x12345678 is an address hint into a 12-region, then its 12-

region's header is 0x123456FF0. Once the header is located, TP returns the first block from the K-region's list of free blocks. This technique provides constant time access to hint's K-region and requires no searching. It checks the K-region's header in a handful of instructions, without touching additional memory.

However, this technique requires that *whole* K-regions are acquired by TP. Since acquiring 2^K bytes - when 2^K is greater than the virtual page size - from the operating system is not guaranteed to be within one K-region and could, in fact, span over two K-regions, TP ensures that the acquisition of whole K-regions is performed in the software. To obtain a K-region, TP acquires twice its size from the system, within which a K-region is selected and the leftover memory is reused for smaller K-regions, as we describe later. When 2^K is less than the virtual page size, TP acquires a whole virtual page and splits it into 2^K byte long K-regions.

TP has an *adjustable* locality accuracy. To accommodate the increasing size classes, TP uses a progressive K-region: objects less than 1KB are stored in 13-regions, less than 2KB in 14-regions, less than 4KB in 15-regions, less than 8KB in 16-regions, and finally objects less than 16KB are stored in 17-regions. To study how this locality accuracy affects performance, we vary the K-region size for size classes smaller than 512 bytes, since 95% of allocated objects are in this class. We varied K between 14, 12, 10, 8 and 7, which creates K-regions of 16KB, 4KB, 1KB, 256 and 128 bytes respectively. We refer to these configurations as TP-14, TP-12, TP-10, TP-8 and TP-7. For small objects larger than half the K-region's size, we use a 12-region. This is because otherwise a K-region would not be able to store two objects and thus object collocation would not be possible. For example 80 bytes objects are not stored in 128 byte regions, since no other 80-byte object can be stored in that region. A pointer bumping technique within the region to allow different sizes to be collocated within it leads to the same issue region based allocation faces: high fragmentation.

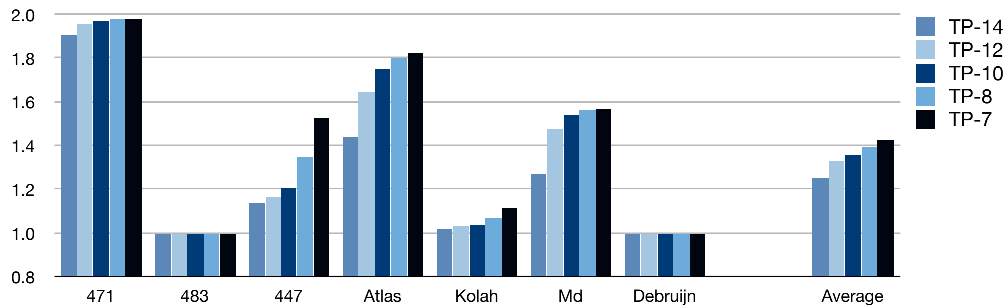


Fig. 9. TP - Average number of K-regions checked per allocation

a. Allocation & Deallocation

TP's strategy is to allocate based on address first and revert to allocation based on size when the former fails. In the first attempt, TP tries to allocate a block from the hint's K-region. It locates the hint's K-region header and returns the first block from that K-region's linked list. If the list is empty, then TP abandons the locality search and reverts to fast allocation based on size. It locates the appropriate hash table entry, selects the first K-region from the size class list and returns its first available block. TP's strength is that K-regions are accessible either by address or size, and both in constant time. This dual access to K-regions allows TP to quickly locate a certain K-region based on an address and revert to size when address fails. Fig. 9 shows the average number of touched K-regions per allocation, which varies between 1 and 2. As K decreases, the likelihood of finding a free block in the hint's K-region decreases as well, forcing TP to allocate more often based on size.

Deallocation

A deallocation request finds the block's K-region and inserts it in the front of its region's linked list. The fast address based search improves not only allocation, but deallocation as well. When a block is returned to its K-region, the address based search proceeds exactly as for allocation with hints. This fast deallocation

gives explicit locality improving allocators a performance edge over locality reference schemes that resort to more intricate solutions, such as PHKmalloc, which linearly traverses an address sorted list of regions.

Complexity

TP has $O(1)$ allocation complexity, with constant time access to any of its K-regions and constant time access to any of its size classes. The deallocation complexity is $O(1)$ as well.

Memory Reuse

When the last of a K-region's objects gets deallocated, the K-region's header is removed from the size class list and retired into a separate array of free K-regions. This array has 32 entries and keeps a linked list for all power of two sized K-regions, see right of fig 8. TP always checks the array of retired K-regions before allocating more memory from the OS. This technique is a coarse form of coalescing at the region level; a K-region is reused when all its blocks are free. This mechanism not only reuses free K-regions among different size classes and reduces memory fragmentation, but also speeds up allocation by assuring that the first K-region in the size class' linked list is non-empty, thus eliminating the search for one.

The source code for TP is available at the following URL:

http://parasol.tamu.edu/resources/downloads.php

b. Performance Analysis

We set TP as STL's default allocator and recompiled the applications. Fig. 10 shows the execution time normalized to dlmalloc's. On average, TP matches or outperforms dlmalloc on every application, reducing execution time by an average of 5-7%, as K varies. Atlas has the largest performance improvement with 22%. TP's performance is sensitive to K's variance: 471.omnetpp and 483.xalancbmk slightly improve with

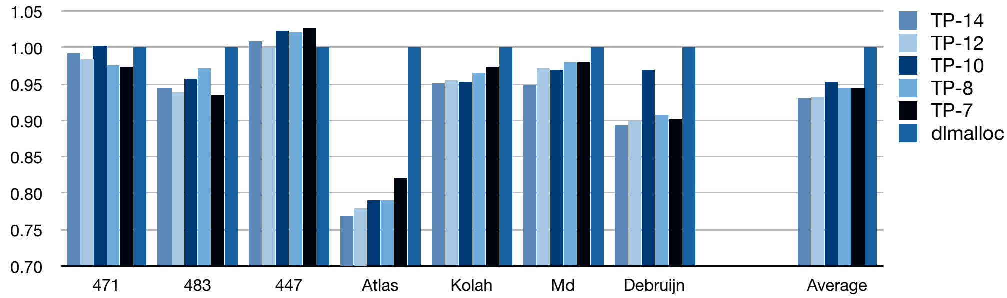


Fig. 10. TP - execution time normalized to dlmalloc

larger K-regions, while the other benchmarks show a reverse trend. This is because speed and locality play different roles in different applications. There is a performance spike for TP-10 (1 KB regions), which, when holding 512 byte deque arrays, has larger fragmentation and slower performance. Page-conscious allocation is slightly more efficient than cache-conscious allocation.

Fig 11 shows the issued instructions normalized to dlmalloc's. TP is faster than dlmalloc on each application, reducing the instructions by as much as 4.2% for Atlas, with an average of 1%. This reduction contributes in part to the execution time improvement. TP's allocation speed slightly decreases with K, since there are more second allocation attempts for smaller Ks. This shows that within TP locality accuracy and speed are inversely proportional.

Fig. 12 shows L1 misses normalized to dlmalloc's. While TP reduces them by an average of 1-5%, page and cache-conscious allocations have almost the same ratios. L2 and TLB misses, shown in fig. 13 and 14 respectively, show a more pronounced average improvement, with 11% for L2 and 18% for TLB misses. Atlas shows the largest TLB misses improvement with 59%. While L2 misses are hardly sensitive to K's variance, TLB misses decrease as K gets larger, showing that page-conscious allocation reduces TLB misses over cache-conscious allocation.

Table VI shows TP's external fragmentation for each configuration. We com-

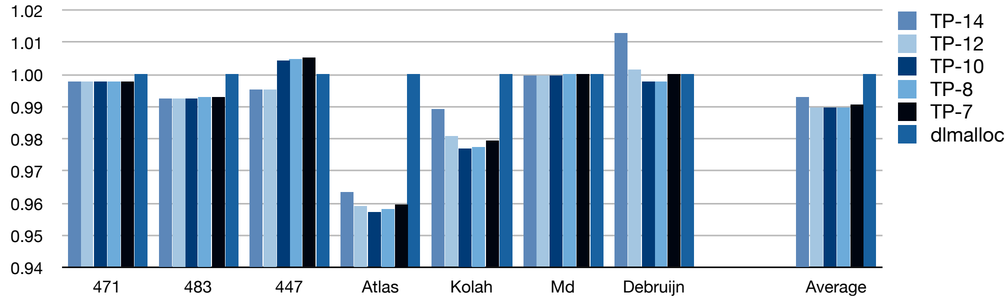


Fig. 11. TP - instructions normalized to dlmalloc

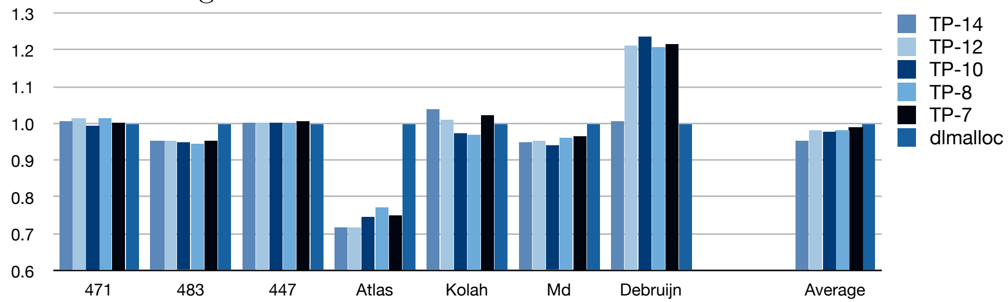


Fig. 12. TP - L1 misses normalized to dlmalloc

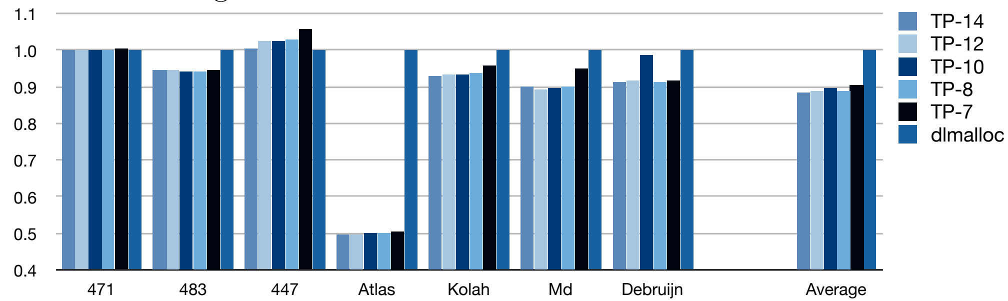


Fig. 13. TP - L2 misses normalized to dlmalloc

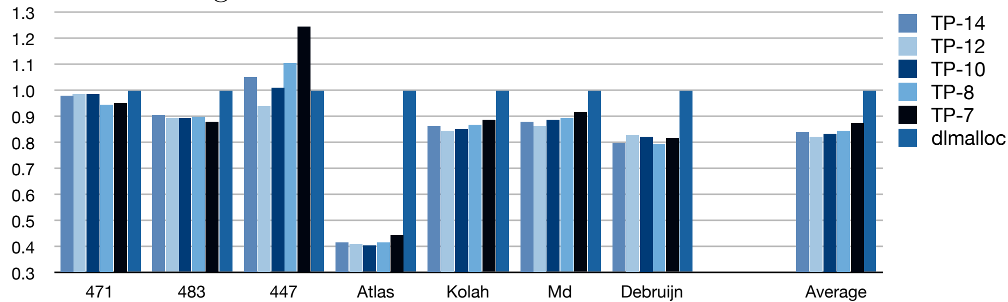


Fig. 14. TP - TLB misses normalized to dlmalloc

Fragmentation	471	483	447	Atlas	Kolah	Md	Debruijn	Geom. Mean
TP-14	100.37	90.01	105.3	15.99	23.29	20.95	20.38	39.5
TP-12	0.64	75.52	21.22	10.77	1.05	21.35	6.64	7.7
TP-10	3.66	76.69	26.33	12.63	3.42	29.87	79.22	18.5
TP-8	7.98	77.57	31.18	14.79	2.87	41.27	42.21	20.3
TP-7	54.6	76.44	60.44	14.28	7.51	26.5	25.12	29.2
dlmalloc	23.48	1.26	18.24	2.66	12.32	8.48	6.77	7.2

Table VI. TP's external fragmentation

puted the external fragmentation as the percentage of the extra memory used by the allocator over the maximum memory requested by the application [40]. TP's fragmentation is very sensitive to the value of K, with TP-12 having the lowest average fragmentation of 7.7%. On the one hand, for K-regions smaller than page size, such as TP-10, TP-8 and TP-7, the fragmentation increases due to end of K-region fragmentation. For example, a 128 byte region that holds objects of 40 bytes has a chunk of 32 bytes at the end of the region that cannot be used and accounts for fragmentation. This explains the high fragmentation for Debruijn, which uses 512 byte arrays that create 80% fragmentation when allocated in 1,024 byte regions. On the other hand, for K-regions larger than page size, such as TP-14, fragmentation also increases because of the alignment reinforcement whose leftover regions are not used. As such, the 40% fragmentation for TP-14 is the largest, while TP-12's page sized regions lead to the lowest fragmentation, comparable to dlmalloc's.

c. The Optimal Value for K

A fundamental question quickly arises: how does one select the optimal value of K? Unfortunately, while we have the qualitative relationship between speed and locality, the empirical one is harder to predict since it depends on the machine hardware parameters as well as the applications. Selecting K faces a tug-of-war: a smaller K increases locality but reduces allocation speed. This is similar to choosing a size segregation scheme, where more size classes reduces internal fragmentation but also increases complexity. We set TP's default value of K to 12 for small objects, or page-conscious allocation. TP-12 optimizes the three major traits of allocation speed, locality and fragmentation.

d. Pitfalls

We identified two pitfalls in TP's design. The first one is that K-regions need to be aligned by 2^K bytes. When K is less than or equal to the virtual page size, this alignment requirement does not pose an issue, since `mmap` allocates memory from the system in multiples of page size. However, when K is larger than the virtual page size, the K-region alignment needs to be done in software by allocating twice the size of the K-region and reusing the alignment leftovers. This technique might lead to higher fragmentation if the leftover K-regions are not used by other size classes. Fortunately, the percentage of large objects that require aligned K-regions is less than 10%, and even in that case, the alignment leftovers are used for smaller objects. The second pitfall is that allocation hints need to be addresses that were previously allocated with TP. If, however, the hints point to memory that was not allocated by TP, then correctness is compromised. Fortunately, in our integration method, all hints are provided from within STL containers, whose memory is allocated by TP. This makes

our proposed STL hint allocation safe. A detail proof for the guarantee that STL containers provide valid hints TP is provided in appendix C. The next allocator we present addresses both of TP's pitfalls.

2. Defero

The second allocator we present, Defero, is a locality improving allocator that addresses both of TP's pitfalls: 1) does not require whole K-region acquisition and 2) it accepts any hints. We will describe shortly how the latter is a corollary of the former.

First, we describe Defero's structure, which lifts the whole K-regions constraint. Defero uses the same size class segregation as TP, see page 39. All blocks of the same size are stored within a size bucket. Within this bucket, the blocks are grouped in separate lists, based on their most significant (32-K) bits. This grouping creates a collection of lists, one for each distinct group of significant bits. To facilitate searching among these lists, they are sorted and stored into a red-black tree, using their most significant (32-K) bits as keys. The red-black tree is balanced and assures that each list is accessed in at most K steps. In essence, Defero's structure is a hash table of trees of lists, as depicted in fig. 15.

Because the access to a list corresponding to a certain K-region is done by traversing a red-back tree, the K-region's header is no longer required. This in turn allows *fragments* of K-regions to be managed, lifting the whole K-region acquisition imposed by TP. Moreover, different fragments of a K-region can be used by different size classes. This is not the case in TP, which splits a K-region into same sized blocks.

When Defero acquires more memory from the operating system, it asks for a size of either page size or nearest larger multiple of page sizes, whichever is smaller. The returned memory chunk is further split into rounded-up same sized blocks. These blocks are then linked into lists based on their most significant (32-K) bits. The

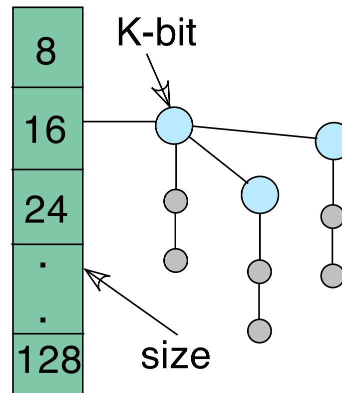


Fig. 15. Defero - internal data structure

resulting lists are then inserted into the red-balck tree corresponding to the block size. For example, suppose a request for 16 bytes triggers Defero to acquire more memory. In this case Defero acquires a virtual page, 4KB on our experimental machine, and splits this page into 256 blocks of 16 bytes each. It then links these blocks into a linked list and inserts this list into the red-black tree corresponding to the 16 size class entry. If a request of 5KB triggers Defero to acquire more memory, then Defero gets 8KB and splits into into two chunks of 5 KB and 3 KB, and then inserts then into their corresponding size classes.

a. Allocation & Dealocation

Defero organizes each size class into a sorted tree of lists of blocks for the purpose of searching for a specific K-region. Upon allocation, Defero receives a 'size' and a 'hint' address. First, it locates the appropriate size class and selects its tree of K-regions. Within this tree, it searches for the list that shares the most significant K bits with the hint address, and its first block is unlinked and returned. However, if such list is found, then the leaf list at which the top-down search ends is selected instead.

If there exists a block in the hint's K-region, Defero guarantees to return it. If there are more than one, Defero returns the front of the K-region's list. The order of blocks within a list depends on the application's deallocation pattern. If there is no block within the hint's K-region, then Defero guarantees to return a block from the closest K-region.This mechanism no longer imposes any restrictions on hints, and hints can have any value. If a hint points to memory that is not in its management, Defero's structure still searches the red-black tree and returns a block from the leaf list found at the end of the top-down search. This list corresponds to the closest K-region to the hint's.

In essence, Defero's strategy prioritizes 'size' over 'address'. In the first step, it selects the appropriate size class, while in the second step, it searches for a block within the hint's K-region. The search is performed only among the blocks of the same size, thus 'size' takes priority over 'address'. Blocks of different size that are closer to the 'hint' address are not considered.

Upon deallocation, a block's size class and its corresponding tree are selected. Within this tree, the block is returned into its K-region. Deallocation's search for a K-region is similar to allocation's. If no K-list is found, the block is inserted into the tree as the header of the new K-region. The allocation and deallocation complexities are $O(K)$.

b. Varying Locality Precision

K is an adjustable parameter between 0 and 32. When K is 0, no two blocks belong to the same 0-region, thus there is only one block per list, and each block is a node in the tree. When K is 32, all blocks are belong to the same 32-region and they are stored in the same list, and the red-black tree has only one node. K's variance between 0 and 32 sets the ratio between the tree size and K-regions' size: larger K means larger lists

Fragmentation	471	483	447	Atlas	Kolah	Md	Debruijn	Geom. Mean.
External	1.5	28.37	24.68	7.21	0.9	22.14	44.74	9.5
Internal	0.01	5.25	9.48	6.43	3.87	9.01	3.32	2.3

Table VII. Defero's internal and external fragmentation

and fewer tree nodes, while smaller K means smaller lists but more tree nodes. This creates an inverse correlation between the location guarantee and allocation speed. The faster the allocation speed the weaker the locality, and vice-versa.

To accommodate larger size classes, the corresponding K-regions increase with size classes, similarly to TP's, see page 39. Objects less than 512 bytes are stored in 12-regions, objects less than 1,024 bytes are stored in 13-regions, less than 2KB in 14-regions, less than 4KB in 15-regions, less than 8KB in 16-regions and finally, objects less than 16KB in 17-regions. Objects larger than 16KB are handled directly by the operating system's `mmap` and `munmap`.

This size segregation has a finer division for smaller sizes, up to 512 bytes, which favors a small internal fragmentation. Following the simple storage segregation philosophy, Defero does not have a K-region reuse mechanism in order to maximize allocation speed. Keeping track of which fragments of a K-region are in use reduces allocation speed, as discussed in section C. Table VII shows the external and internal fragmentation for the seven benchmarks. Please note that TP and Medius (presented in section 3) also share the same internal fragmentation, since this depends only on the size segregation scheme.

To study the effect of locality precision on Defero's performance we varied K between values of 20, 16, 12 and 8. However, we vary K only for sizes less than

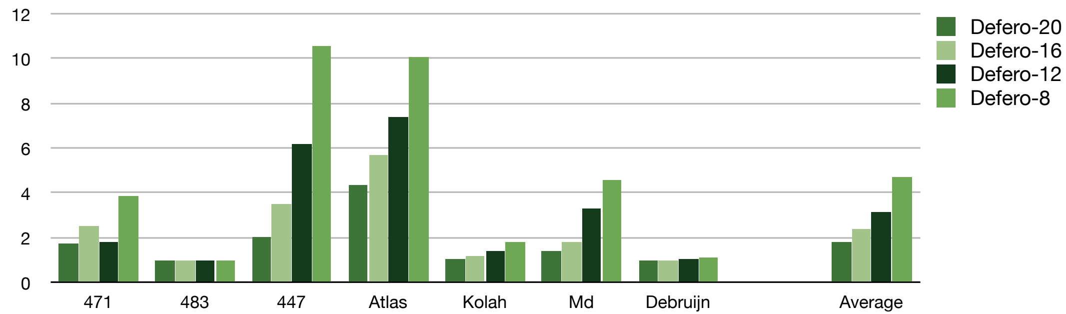


Fig. 16. Defero - as K increases, the average number of tree nodes traversed per allocation increases, reducing allocation speed.

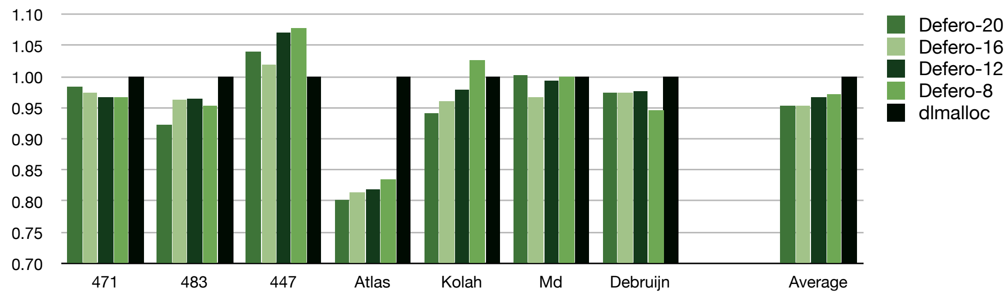


Fig. 17. Defero - execution time normalized to dlmalloc - lower is better

512 bytes, since this category represents 95% of total allocations. We refer to these configurations as Defero-20, Defero-16, Defero-12 and Defero-8.

An allocation traverses the tree top-down, and the number of tree nodes traversed is inversely proportional with K. Fig. 16 shows the average number of tree nodes per allocation for the seven C++ applications. On average, the number of traversed tree nodes increased from 1.8 to 2.4 to 3.2 to 4.7 when K decreases from 20 to 16 to 12 to 8. The allocation speed is proportional to this number, and inversely proportional to K. The number of nodes traversed also affects Defero's spatial locality due to the K-regions being touched during an allocation.

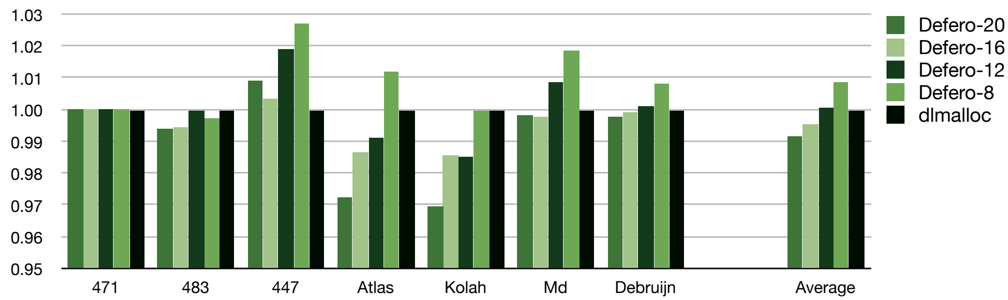


Fig. 18. Defero - issued instructions normalized to dlmalloc

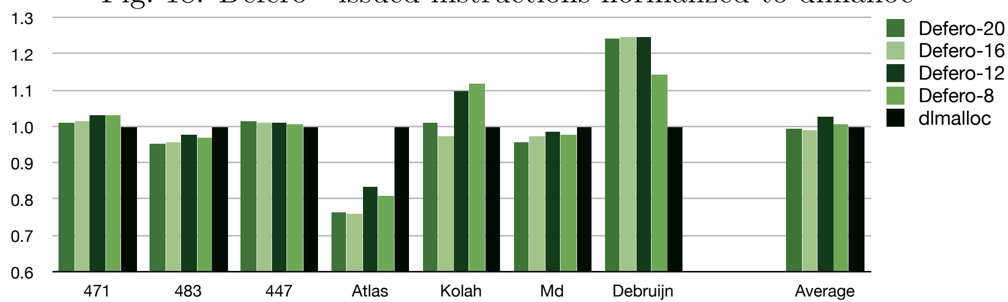


Fig. 19. Defero - L1 misses normalized to dlmalloc

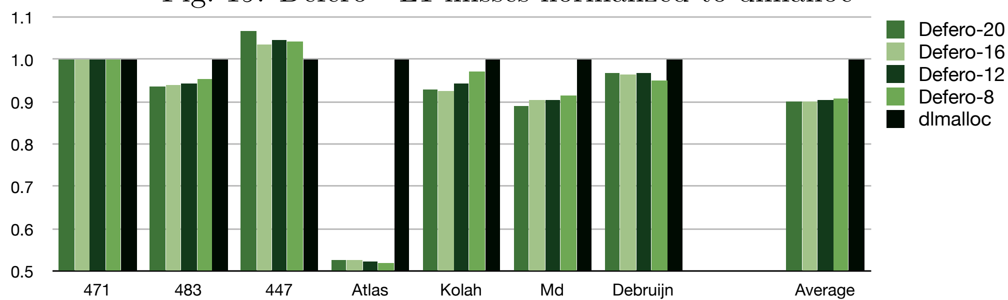


Fig. 20. Defero - L2 misses normalized to dlmalloc

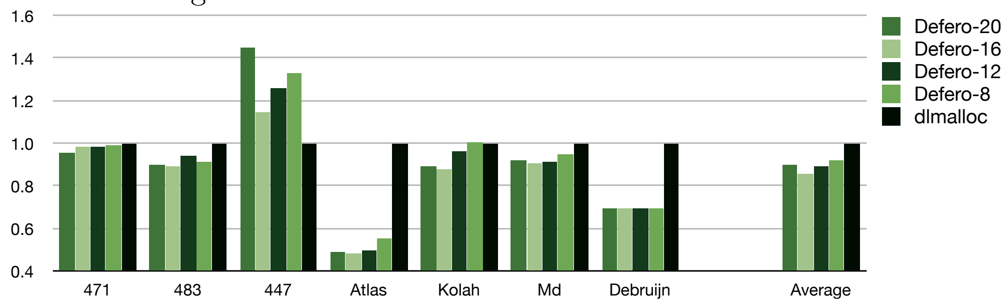


Fig. 21. Defero - TLB misses normalized to dlmalloc

c. Performance Analysis

We set Defero as the STL default allocator and recompiled the applications. Fig. 17 shows the application's execution time normalized to dlmalloc. Defero reduced the execution time by 3-5% on average, and up to 20% for Atlas. Six of seven benchmarks performed better with Defero than with dlmalloc. Larger values of K do better on 471.omnetpp, 483.xalancbmk, and Debruijn, while smaller values of K reverse the trend for the other applications. Except for Debruijn, the page-conscious allocation did slightly better than the cache-conscious one. Fig. 18 shows the number of instructions issued by each application normalized to dlmalloc's. As K decreases, the allocation speed decreases as well. On average, a lower K is slower than a higher one.

Fig. 19, 20 and 21 show L1, L2 and TLB misses normalized to dlmalloc's. While Defero reduces L1 and L2 misses by 1-3% and 10% respectively over dlmalloc, they do not show a strong correlation to locality accuracy. TLB misses do show a slight inverse correlation though. TLB misses increase with smaller K-regions, due to the increasing number of K-regions touched in the top-down tree traversal for each allocation/deallocation.

d. The Optimal Value for K

To improve allocation speed one might consider increasing K. However, such an increase leads to a deterioration of the locality precision. A comprise between these two antagonistic trends leads to a middle value of K. We empirically found that a value of 12 for K provides the best performance blend.

e. Pitfalls

We identified two pitfalls for Defero. The first one is that Defero does not reuse memory, which leads to higher fragmentation. The second pitfall is that Defero touches several pages in its search for address, which increases the costs of locality improving.

Overall, TP is better than Defero in terms of execution time and fragmentation. However, as we discuss in section C, Defero outperforms TP in terms of fragmentation. The next allocator we present attempts to address TP's and Defero's pitfalls, but introduces new ones.

3. Medius

The third allocator we present, Medius, is a locality improving allocator that addresses Defero's pitfalls, and directly accesses a K-region without touching additional memory.

Medius divides the memory into the same K-regions. Each K-region in memory has an entry in a 2^{32-K} size array, regardless of their acquired status. Medius does not require whole K-region acquisition from the operating system, and it can manage *fragments* of K-regions, as well as whole K-regions. Medius uses TP's and Defero's simple size segregation, and rounds up requests to the nearest largest size class, see page 39. In contrast to TP and Defero, Medius does not split the whole K-region into same size blocks, but allows it to hold blocks of different sizes. However, same sized blocks within a K-region are all stored into an unsorted linked list. If a K-region holds more than one size class, it links all its size classes into a linked list as well. Thus, in essence, a K-region has a list of lists. Fig. 22 shows Medius' structure.

The fine size segregation reduces internal fragmentation, see table VII, but it

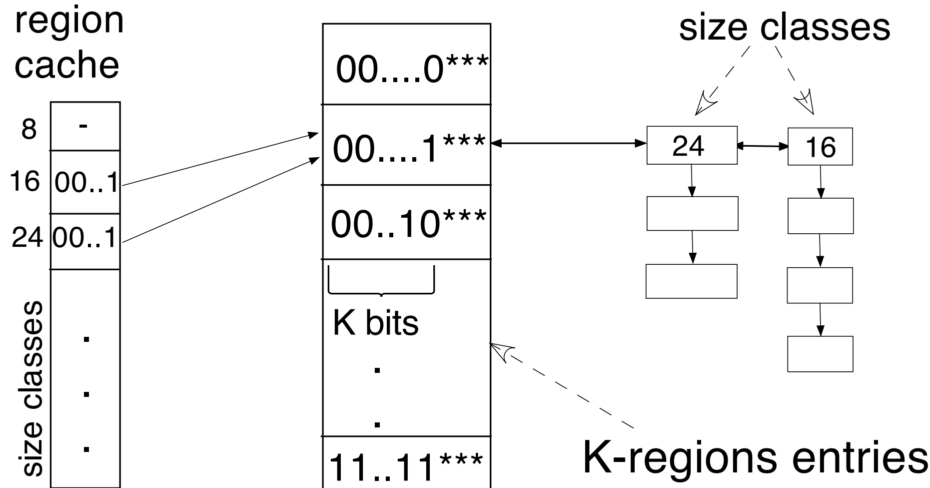


Fig. 22. Medius - internal data structure

creates a larger number of size classes. Medius, just like Defero, acquires memory from the operating system in quanta of either page size or nearest larger multiple of page sizes, whichever is smaller. This memory chunk is further split into rounded-up same sized blocks, which are linked in a list. This list gets inserted in its corresponding K-region's entry. For example, suppose a request for 16 bytes triggers Medius to acquire more memory. In this case Medius acquires a virtual page, 4KB on our experimental machine, and splits this page into 256 blocks of 16 bytes each. Medius then links these blocks into a linked list and insert it into its corresponding page entry. If a request of 5KB triggers Medius to acquire more memory, Medius gets 8KB and splits into into two chunks of 5 KB and 3 KB, which are further inserted into their corresponding K-region entries.

To study how the locality accuracy affects Medius' performance, we vary K between 22, 20, 16, and 12 and use Medius-22, Medius-20, Medius-16, and Medius-12 to represent these configurations. These values of K create K-regions of 4 MB, 1MB,

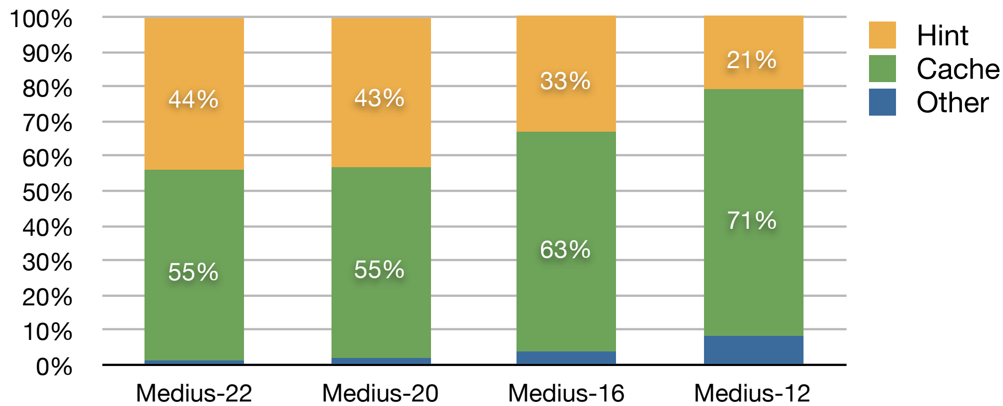


Fig. 23. Medius - average distribution of the K-regions from which allocations occurred

64 KB and 4 KB, respectively.

a. Allocation & Deallocation

Medius' strategy increases region clustering and favors fast address searching. Medius prioritizes address over size: first it finds the hint's K-region and then searches for the right sized block within it. Allocation selects the hint's K-region and linearly searches for the requested size class within this K-region. Once the size class is located, the first block in this size-list is returned.

Cache for Size Classes

If no blocks are found in the hint's K-region, either because the K-region is empty or the requested size class is not found, then Medius abandons the locality request and begins searching for the first accessible block of the requested size. To speed up this search and to increase its locality of reference, Medius stores the index of the K-region from which the requested size was last allocated in a cache array, with one entry for each size class, as depicted in fig. 22. When the hint's K-region fails, Medius checks the cached K-region corresponding to the requested size class. Fig. 23 shows the percentage of allocations satisfied from the hint's K-region, the

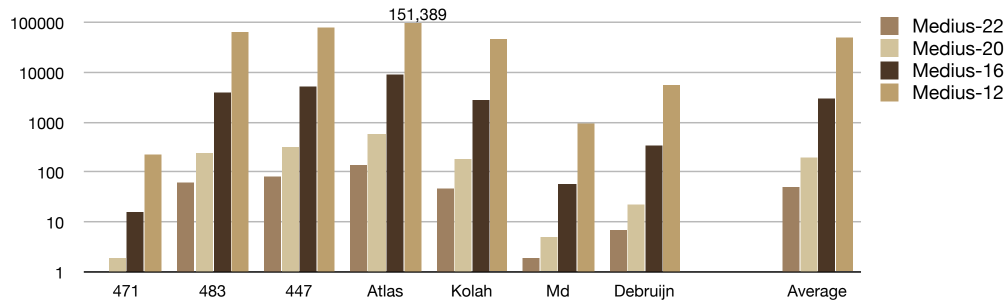


Fig. 24. Medius - active memory range

cached K-region and other K-regions for four configurations and averaged over the seven applications. More than half of all allocations occur from the cached K-region. The cache mechanism also increases Medius' temporal locality.

Exhaustive Search for Other K-regions

Medius' strategy expects the majority of allocations to get satisfied from the hint's or cached K-regions. When these two attempts fail, Medius starts an exhaustive search for a K-region that has the requested size class. The exhaustive search sweeps the K-region array, starting with the lowest available entry and working its way up. This strategy of allocating memory from the lowest address improves an application's spatial locality and increases region clustering. However, the exhaustive search slows down the allocator for small Ks. This comes as a compromise when storing K-regions in a array for constant time access. The "Other" in fig. 23 indicates the percentage of allocations satisfied from the K-regions found using the exhaustive search. This percentage is directly proportional to the K-region's size, increasing from less than 1% for 22-regions to 7.5% for 12-regions. If none of the K-regions has the requested size, then Medius acquires more memory from the operating system.

The exhaustive search can be sped up by reducing the number of visited array entries. Due to heap and stack increasing from opposite ends, the occupied entries are concentrated at one end of the array. Medius records the minimum and the

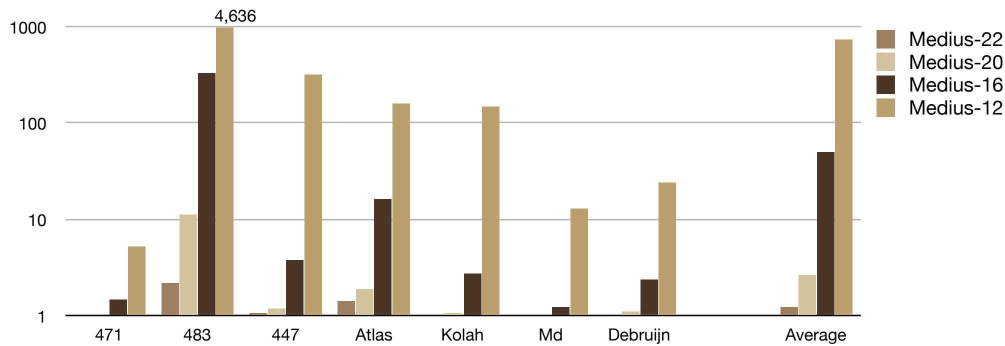


Fig. 25. Medius - average number of visited array entries

maximum array indices for its managed address into an active memory range. This range gets updated with every memory acquisition from the operating system. The exhaustive search only traverses this range instead of the whole array. Fig. 24 shows the active memory range in a logarithmic scale for the seven applications we use. This technique reduces the range of valid array entries from two million to one hundred and fifty thousands entries for Medius-12, and from four thousands to less than two hundred for Medius-20. In addition, this technique also reduces the average number of visited K-regions per allocation. Fig. 25 shows this number in a logarithmic scale and for each application, number which rapidly increases from 1.3 for Medius-22 to 757 for Medius-12. Another advantage is that despite the whole array being mapped in the virtual address space, only the active array entries get mapped to the physical memory. Therefore, the actual physical memory used for the array is proportional to the amount of actual used memory by the application.

An alternative strategy within the exhausting search is to stop at the first block that is larger than the requested size and split it into two. However, splitting larger blocks into smaller ones leads to an accumulation of smaller size classes in the front of the size lists as ”froth”, which slows down the search [77]. We chose not to split and instead to search for an existing size class, expecting that deallocated blocks will

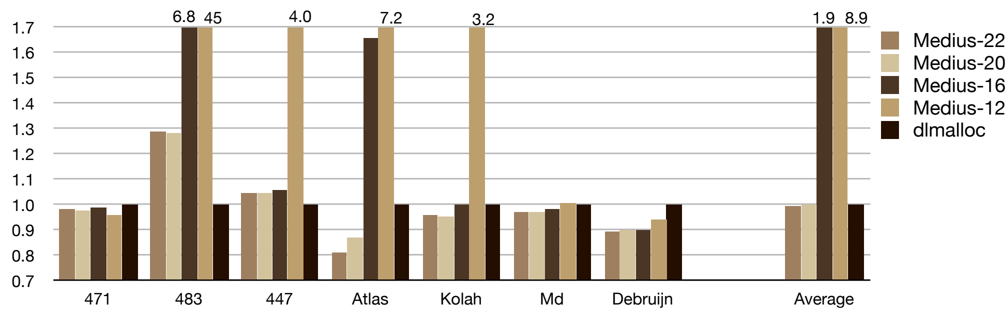


Fig. 26. Medius - execution time normalized to dlmalloc (lower is better)

be requested again.

Deallocation

A deallocation request finds the block's K-region and searches that region for the right size class. If found, the block is inserted in the front of that size-list, otherwise the block becomes the header of a new size-list which gets inserted in the front of the K-region's list of size-lists.

Complexity

The complexity in the best case scenarios, allocating from a hint's K-region or cached K-region, is $O(S)$, where S is the largest number of size classes within a block's K-region. The complexity in the worst case scenario, exhaustive search, is $O(S * I)$, where I is the size of the active memoy range. Fig. 23 shows the proportion of the best case scenarios, "Hint" and "Cache", and the worst case scenario in "Other". The complexity for deallocation is $O(S)$.

b. Performance Analysis

We set Medius as STL's default allocator and recompiled the applications. Fig. 26 shows the execution time for the seven applications normalized to dlmalloc's. Medius-22 and Medius-20 perform similarly to dlmalloc on average. Medius-16 and Medius-

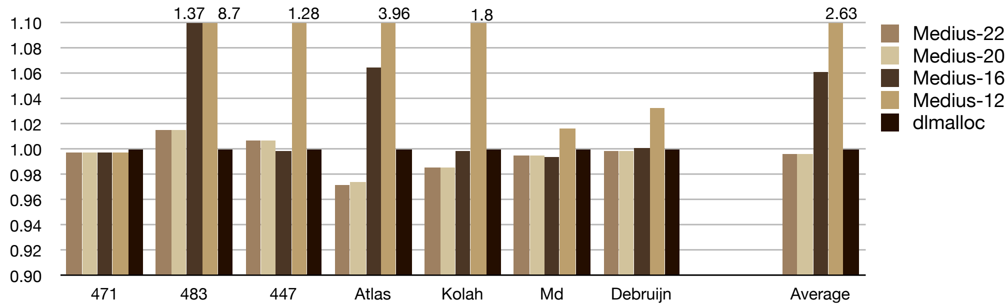


Fig. 27. Medius - instructions normalized to dlmalloc

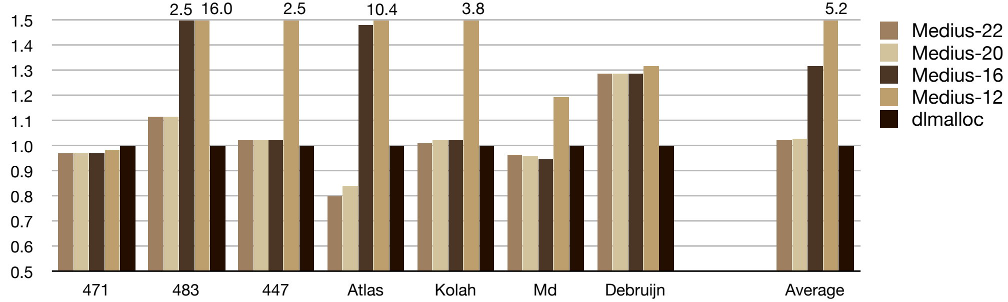


Fig. 28. Medius - L1 misses normalized to dlmalloc

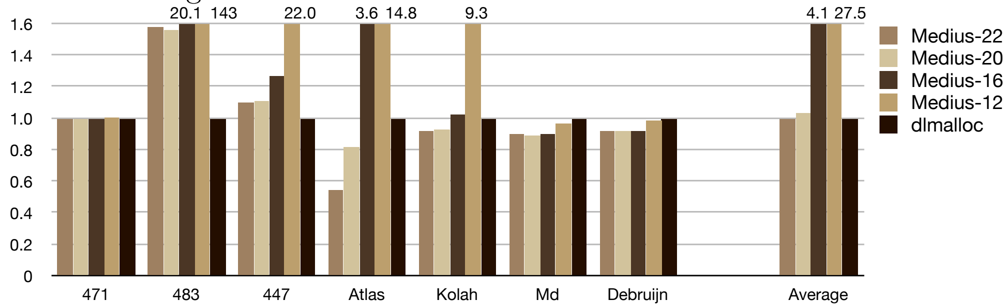


Fig. 29. Medius - L2 misses normalized to dlmalloc

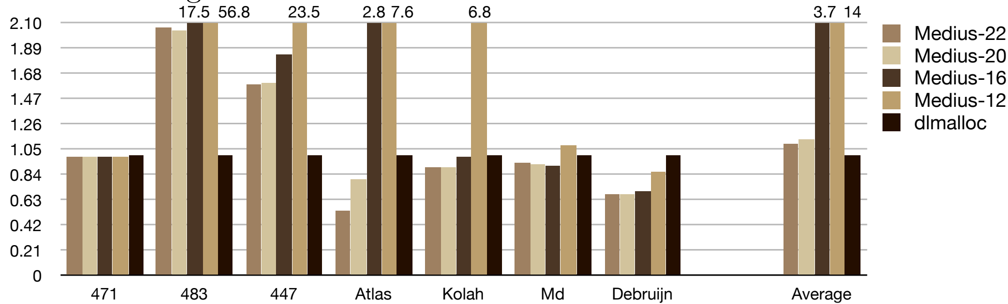


Fig. 30. Medius - TLB misses normalized to dlmalloc

12 though are not competitive for applications that use hundreds of megabytes of memory, due to its exhaustive search. Medius-16 is still on par with dlmalloc on five out of six application, except for 483.xalancbmk, for which is three times slower. On the other hand, for lower values of K, Medius improves over dlmalloc on five of the seven applications. This result is perhaps surprising due their lower locality accuracy. Fig. 27 shows the issued instructions normalized to dlmalloc's. Perhaps expected this time, lower values of K reduce the number of instructions by an average of 0.7% over dlamlloc's. Fig. 28 shows the L1 misses normalized to dlmalloc's. Medius-20 performs similarly to dlmalloc in this category. Fig. 29 and fig. 30 show the L2 and TLB misses normalized to dlmalloc's, which have the same trends as L1 misses. These locality measurements complete the picture for Medius-20's performance improvement, which is due to a reduction in TLB misses and in the number of instructions.

Table VIII shows Medius' fragmentation as K varies. Medius' array accounts solely for the fragmentation difference between the four values of K. Medius-22 and Medius-20 have a fragmentation of 11% , the lowest of the four configurations, but still higher than dlmalloc's 7%. For applications that use large amounts of memory, the 4 MB array corresponding to Medius-12 increases fragmentation minutely. However, for applications that use less memory, such as Deburijn, Md and 471.omnetpp, the 4 MB array increases fragmentation more substantially. These results show that Medius' efforts to increase locality accuracy also increase fragmentation.

c. The Optimal Value for K

In spite of its improved page collocation accuracy, presented later in section 2, Medius-12's speed is too slow. A smaller K improves speed, but decreases locality. As K increases, the complexity of the exhaustive search diminishes, but now Medius searches larger K-regions to find a size class. These two antagonistic trends balance

Fragmentation	471	483	447	Atlas	Kolah	Md	Debruijn	Geom. Mean
Medius-22	1.89	37.22	28.33	12.86	0.85	19.65	44.59	11.0
Medius-20	3.18	37.19	25.33	12.88	0.88	20.15	44.67	11.7
Medius-16	29.09	37.3	24.8	12.91	0.88	25.9	46.14	16.7
Medius-12	70	39.29	26.15	13.46	1.4	120.86	69.72	27.3
dlmalloc	23.48	1.26	18.24	2.66	12.32	8.48	6.77	7.2

Table VIII. Medius' external fragmentation

each other for the smallest number of instructions for Medius-20. These results are directly correlated with execution time, showing that locality accuracy and speed compete against each other; as one increases, the other one decreases. Thus, we set the default value for Medius to 20, for which we have experimentally observed that Medius performs the best. This value of K has less locality accuracy than its higher counterparts, but it has the fastest allocation speed and the lowest fragmentation.

d. Pitfalls

We identified two pitfalls in Medius' design. The first one is the exhaustive search phase, which slows down the allocation speed. As K increases, this pitfall starts to vanish, but a different one arises: the locality accuracy deteriorates. The second pitfall is that Medius does not recycle free memory between size classes. Medius' attempts to address TP's and Defero's pitfalls lead to an overall performance deterioration in terms of execution time and allocation speed, but Medius reaches the highest page collocation accuracy, as described in section 2.

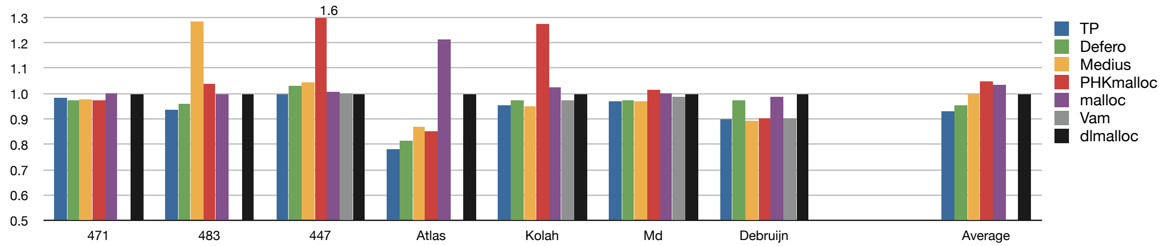


Fig. 31. Execution time for TP, Medius, Defero, PHKmalloc, and malloc normalized to dlmalloc's (lower is better)

4. Summary of TP, Defero and Medius

Before we summarize the three locality improving strategies, we first compare their performance to three other memory allocators: two locality improving allocators, FreeBSD's allocator and Feng's Vam[23], and the native GNU 4.1 `malloc`. We used Vam, packaged with HeapLayers version 3.4.0, on the seven benchmarks, but unfortunately it run only on four programs, 447.dealII, Kolah, Md and Debruijn, crushing on the remaining three[7]. On the four running programs, TP was 0%, 2%, 1% and 0% faster than Vam. Our three allocators perform or match on average theses state-of-the-art allocators.

Fig. 31 shows the execution time of TP, Defero, Medius, PHKmalloc, and the native GNU 4.1 `malloc` normalized to dlmalloc. Each allocator was selected as the underlying default allocator for STL and codes were recompiled without any modifications to the code. TP outperforms all other allocators on all applications. On average, TP is 7% faster than dlmalloc, 17% faster than PHKmalloc and 2% faster than the second closest, Defero. Medius performs as well as dlmalloc.

In this chapter we present three novel locality improving allocators: Defero, which prioritizes size over location, Medius, which prioritizes location over size and TP, which can do both. All three allocators, TP, Defero, and Medius maintain their blocks partitioned into K-regions at all times during the application's execution. This

	TP	Defero	Medius
Priority	Search location then size	Search size then location	Search location then size
Strengths	O(1) operations Memory reuse	Any K Any hints	High locality Any hints
Weaknesses	Whole K-Regions Hint restriction	Touches memory No memory reuse	Expensive when missing in hints region or cache No memory reuse
Trade-offs	Fragmentation for speed and locality	Locality for speed	Speed for locality

Table IX. Comparison of TP, Defero and Medius

property remains invariant regardless of the application's allocation/deallocation pattern. It is this invariant that guarantees an allocation to find a block within a specific K-region, if it exits. The invariant is also enforced at deallocation, which ensures that a block is returned in its own K-region. This allocator's property improves locality even without hints for applications that exhibit temporal locality, as we detail in chapter V.

TP and Defero increase the K-region as the size class increases, to accommodate for fragmentation and allocation speed, respectively. Medius, in contrast, uses the same K-region for all size classes. Despite its attempts to improve TP's whole K-region and hint restrictions, Medius looses speed in the process. Defero lifts both TP's restrictions but at the price of touching more memory when searching a K-region. TP is a fast allocator that retains the speed of simple segregated storage. Its constant time address search makes it suitable for locality improving techniques,

while its memory reuse reduces fragmentation. The combination of these three traits make TP an allocator for all three purposes: locality *and* speed *and* memory reuse, optimizing the first two. Table IX summarizes the strategies, strengths, weaknesses and trade-offs for TP, Defero and Medius. Next, we analyze the relations between speed, fragmentation and locality.

C. Design Challenges for Locality Improving Allocators

In this section we examine the relations between three main characteristics of a locality improving allocator: allocation speed, locality[7] and fragmentation. An ideal locality improving allocator has high locality, high allocation speed (fast) and low fragmentation (little wasted memory). We show that, perhaps expectedly, an allocator's three major characteristics, allocation speed, locality and fragmentation, circularly compete with each other in a game of rock, paper, scissors. Efforts to increase locality slow down speed and fragmentation; efforts to increase speed impairs fragmentation and locality; and efforts to improve fragmentation hurt both locality and speed. These three antagonistic traits force an allocator's design to sacrifice a trait over the others.

Each of our three allocators has its own blend of characteristics, with its strengths and weaknesses: TP focuses on speed and locality, Defero keeps fragmentation constant but trades speed for locality, and Medius increases locality and fragmentation to speed's detriment. All three allocators have an adjustable locality precision that can be simply tuned by a number $K \in [0, 32]$. This feature allows us to study the effect of locality on an allocator's design. While our synthesize is confined to TP, Medius, and Defero only, they have different allocation strategies, with different strengths and

[7] We use the term "locality" to denote the locality accuracy, expressed by parameter K in our allocators: high locality corresponds to low K.

weaknesses. Furthermore, to the best of our knowledge, these three are the only documented techniques that explicitly allocate based on hints and that require no code modification.

1. Speed Competes with Fragmentation

The relation between allocation speed and fragmentation is inversely proportional. This relation is better understood since it has been studied for several decades. Previous work shows that allocation speed competes with fragmentation; the fastest allocators, such as simple segregated storage and binary buddy systems, exhibit a higher fragmentation [77, 40, 48, 32, 59]. Our experiments also corroborate this relation. Table X shows the fragmentation of the five allocators we studied, namely TP, Medius, Defero, PHKmalloc and dlmamloc. dlmalloc and TP have the lowest fragmentation, while PHKmalloc shows the largest fragmentation due to its power of two size segregation. Despite whole K-regions alignment, TP has a smaller fragmentation than Medius and Defero, which is due to its reuse mechanism. None of our allocators store object headers in the allocated blocks, since they are used and deallocated by STL containers, which supply the size of a free block upon deallocation. This technique reduces internal fragmentation over allocation strategies that store object headers, such as dlmalloc.

TP and dlmalloc have similar average fragmentation when computed as a geometric mean. This mean more accurately describes the average, since it is more tolerant of large numbers. The arithmetic mean on the other hand is more susceptible of these outliers. However, both means maintain the order trend. Accounting fragmentation as a geometric mean, **our results show that TP outperforms dlmalloc in fragmentation, allocation speed and locality**: matches it in fragmentation, and outperforms it in allocation speed and locality.

Application	TP	Defero	Medius	PHKmalloc	dlmalloc
471.omnetpp	0.6	1.5	3.2	58.0	23.5
483.xalancbmk	75.6	28.4	37.2	30.3	1.3
447.dealII	21.2	24.7	25.3	57.0	18.2
Atlas	10.8	7.2	12.9	23.4	2.7
Kolah	1.1	0.9	0.9	5.4	12.3
Md	21.4	22.14	20.15	20.7	10.7
Debruijn	6.6	44.7	44.7	1.3	6.8
Arithmetic Mean	19.6	18.5	20.6	28.0	10.8
Geometric Mean	**7.7**	**9.5**	**11.7**	**18.1**	**7.2**

Table X. Fragmentation for TP, Defero, Medius, PHKmalloc and dlmalloc

While allocation speed vs. fragmentation has been studied before, locality's relation to speed and fragmentation *within the same allocator* has not been studied before. In a locality comparison study, Grunwald et al. compare five *different* allocators[27]. In contrast, in the next two sections we show that for the same allocator, locality stands antagonistically in its relations to both speed and fragmentation, just as speed does against fragmentation.

2. Locality Competes with Speed

We found that locality directly competes with speed. To increase locality, an allocator invests more effort searching for a smaller hint's vicinity. All three allocators we experimented with exhibited the same trend, executing more instructions as locality increases, see fig. 11, 18, and 27. For TP, this is due to the increasing number of second attempts based on size, for Defero to the increasing number of tree nodes

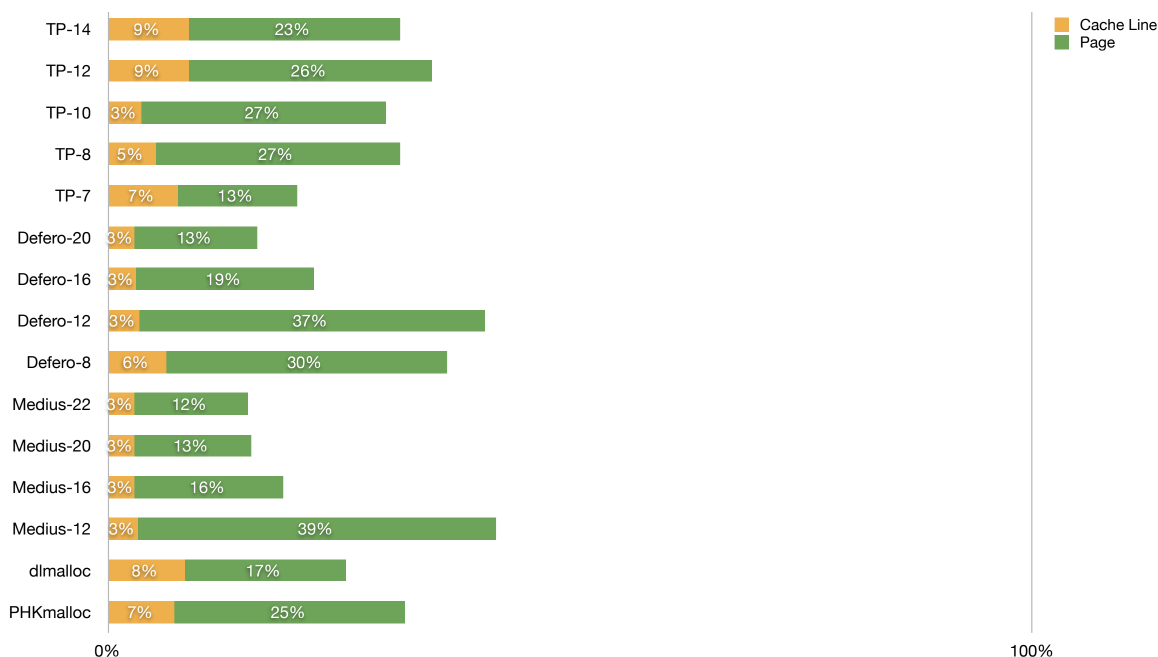


Fig. 32. Average relative hint location of an allocated block for Defero, Medius, TP, dlmalloc and PHKmalloc

traversed, and for Medius to the exhaustive search.

To isolate the benefits of locality from the costs, we measure the location of the allocated blocks relative to their hints and we break it down into four categories: 1) within hint's cache line, 2) within hint's virtual page, but not within its cache line, 3) in a different virtual page than hint's and 4) allocation has no hint. For the latter category, STL containers provided hints to 48% of allocations averaged over seven applications. The remaining 52% did not have hints either because there were no neighbors, such as allocation of list and tree headers, or its neighbors had a different size, such as vector expansion. Fig. 32 shows the first three categories. The figure shows the allocated address relative to the hint's as an average for the seven benchmarks for TP, Defero, Medius, dlmalloc and PHKmalloc, normalized to the allocations that had hints. For TP, as locality accuracy gets closer to the cache line, so does the cache line accuracy, but to the detriment of page accuracy. However,

471.omnetpp has a high cache line collocation for TP-18 and TP-20, which increases their average. For Defero, page-conscious allocation maximizes the page accuracy, but decreases its cache accuracy. For Medius, page accuracy increases from 16% to 52% as locality accuracy increases, while cache line accuracy remains around 3%. Fig. 32 also shows a strong correlation between region size and locality.

We attribute the *page conscious* accuracy difference between allocators to the order selection of K-regions when allocating with no hints. In this case, for the same sized K-regions and the same size segregation, the allocators differ only in the order in which K-regions get selected. Suppose there are three regions, with addresses of 1, 2 and 3. TP selects them in no order, but rather based on the application's allocation/deallocation pattern[8] Defero selects them in 2, 1 and 3 order, a semi-order, since the root of the K-regions tree is chosen last. Medius selects them in 1, 2 and 3 order, a full order, since it always selects the K-region with the lowest address. There is a direct correlation between the hint allocation's page accuracy and the order in which K-regions are selected: TP with 35% (drops to 31% without the memory reuse mechanism), Defero with 39%, and Medius with 42% . This shows that not only page-conscious allocation is effective at improving locality, but also page *clustering*. Unfortunately, it also requires more effort: when one studies the allocators' speed, the trend is reversed, with TP the fastest, Defero the second and Medius last. The page clustering technique is also observable when comparing PHKmalloc to dlmalloc: the former, which uses page clustering by sorting its pages based on address order, yields a 32% page accuracy, while the latter, which relies only on coalescing for clustering, has a 23% page accuracy.

[8]TP's region reuse mechanism reduces the number of used K-regions and thus increases locality regardless.

3. Fragmentation Competes with Locality

We also found that fragmentation competes with locality. From a space availability perspective, reducing fragmentation also reduces the number of free blocks available for searching. Since a request for locality is satisfied from the available memory at the time of the request, efforts to reduce this space for fragmentation purposes also reduces spatial locality. The more memory available to the allocator, the better the chances are for a block to be placed close to its hint. From a region based perspective, the smaller the region the higher the fragmentation. This is noticeable in TP's fragmentation at the end of the region and Medius' array size, both increasing with locality accuracy, see tables VIII and VI. Defero on the other hand does not trade fragmentation for locality, but it trades speed for locality.

4. Balancing Allocation Speed, Locality and Fragmentation

Balancing these three antagonistic characteristics - allocation speed, locality and fragmentation - remains a challenging task and one has to tradeoff one trait for the other two. We use the experimental results to *qualitatively* depict these three antagonistic traits for the five allocators we studied, Medius, TP, Defero, PHKmalloc and dlmalloc, 15 total allocation configurations. Fig. 33 shows these configurations using speed, fragmentation and locality as spatial antagonistic coordinates. The closer to the line in absolute distance an allocator is, the better that trait. The three axes stand in opposition to one another and form a triangle that represents the ideal allocator. The picture shows that all configurations we have studied emphasize two of the three traits, while sacrificing the other.

We found TP's blend of these three antagonistic properties to provide the best overall execution time improvement. While its fragmentation can probably be further

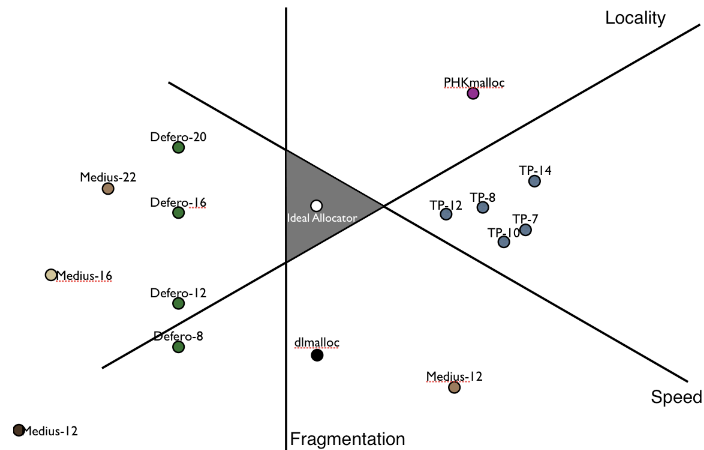


Fig. 33. Speed, locality and fragmentation for five allocators and several configurations (closer to the line is better)

reduced with careful consideration, we believe it will come at the price of either speed or locality. Defero and Medius can also benefit from reducing fragmentation, but any effort will probably slow them down too. Medius risks more, but has the highest page accuracy. However, this strategy yields high complexities in the worst cases. PHKmalloc reduces speed but increases fragmentation. PHKmalloc also increases locality by sorting its pages in address order.

An important issue to locality improving techniques is the selection of allocation hints. An ideal allocation hint points to a spatial neighborhood that is traversed by the application's algorithm. For example, in the tree container, if all traversals are depth-first search, the sibling address is not a good hint, since it creates a memory layout that favors breadth-first search. Another aspect for allocation hints is the constraints imposed on them. TP' imposes that a hint needs to be an address within its memory management, while Defero and Medius do not impose such constraints.

We believe that, just as the size segregated quick lists stand at the core of high performance allocators[26], such as dlmalloc, PHKmalloc, QuickFit, Vam, Defero and TP[26, 53, 46, 76, 23], the aligned K-region technique used by TP to instantly access

a certain region stands at the core of explicit locality improving allocators. This is because of two reasons: 1) it finds a vicinity in constant time and 2) it does not touch addition memory. We believe this technique has a good balance of speed, locality and fragmentation, with emphasis on speed and locality. While fragmentation is high for smaller Ks, it drops to a minimum when a K-region matches the virtual page size.

CHAPTER V

HINT VS. HINTLESS: A COMPARATIVE STUDY OF LOCALITY IMPROVING ALLOCATION TECHNIQUES

Locality conscious allocation techniques improve an application's performance by optimizing its data layout and reducing cache and TLB misses. However, the credit for the performance improvements is not clearly defined. How much of this improvement should be credited to the allocators and how much to the hints? This chapter investigates this issue and shows that allocating with hints performs on average as well as allocating without them, provided an allocator with high locality of reference. Our findings also reveal that while hint allocation does improve locality by up to 40%, the costs offset the benefits. We further show that the STL audience is knowledgeable about locality issues and uses mostly vectors. This STL usage pattern partly explains why allocating with hints and without them yields similar performance and further shows the limitations for locality improving techniques in the STL environment.

We analyze three different allocation strategies, TP, Defero and Medius, on seven STL benchmarks and present a quantitative and qualitative comparison between two locality improving techniques: 1) allocating with hints, or *hint* allocation for short, and 2) allocating without hints, or *hintless* allocation for short. To infer with certainty that the observed effects are indeed credited to the hint switch, we only changed the hint parameters, and kept the other parameters the same. Our three allocators can allocate with and without hints, which allowed us to fairly compare the two scenarios[44]. Unless otherwise stated, we use the allocators' default settings in this comparison.

The ability to *use* allocation hints efficiently belongs to allocators, while the ability to *supply* the hints to the allocators belongs to the STL containers. For this

reason, we divide our comparison into two parts. In the first part we analyze the efficiency of allocators to use hints and their impact on performance and locality. In the second part we analyze how allocation hints are generated and how effective STL containers are at providing hints to their allocator.

A. Locality of Reference - Improving Locality without Hints

A qualitative comparison between explicit locality improving allocators, such as TP, Defero, and Medius, on the one hand, and locality of reference allocators, such as PHKmalloc, on the other hand, reveals two main differences. The first difference is that the former category accommodates explicit address allocation, being capable of exploiting applications' spatial hints, while the latter has no such capability. The second difference is that deallocation in the former benefits from the address based search as well. This comes as a corollary to the first difference. These two differences, explicit address based allocation and fast deallocation, distinguish explicit locality improving allocators from locality of reference improving allocators.

Our three allocators turn into locality of reference improving allocators even when they allocate without hints. This is because consecutive allocation requests are serviced from the same K-region: hintless TP allocates from the first available K-region, hintless Defero allocates from the K-region located at the root of the tree, and hintless Medius allocates from the last recently used K-region. If an application has temporal locality, i.e. consecutive allocations are accessed together, then this technique guarantees spatial locality, i.e. blocks accessed together are allocated together. Upon deallocation, a block is returned to its corresponding K-region, thus maintaining the K-region organization regardless of the allocation/deallocation pattern.

Another qualitative difference between hint and hintless allocation is that hintless

allocation is predicated by application's temporal locality while hint allocation is not. The two conditions that hintless allocation presupposes to improve spatial locality are: (1) locality of reference and (2) applications' temporal locality. When these two conditions happen simultaneously, the hintless allocation becomes a spatial locality improving technique as potent as hint allocation.

B. Hints vs. Hintless

To isolate the benefits of locality from the costs we measure the location of the allocated blocks relative to their hints and we break it down into four categories: 1) within hint's cache line, 2) within hint's virtual page, but not within its cache line, 3) in a different virtual page than hint's and 4) allocation has no hint. The first three measurements are reported relative to the allocations that received hints. The allocators use the hints for relative measurement in the hintless scenario. We refer the ability of an allocator to collocate a new block within the hint's cache line as *cache accuracy*. Similarly, we use the term *page accuracy* to denote the same concept for virtual page collocation.

We compare hint and hintless allocations side by side and observes their impact on the applications' performance. We inspect four perspectives in this comparison:

performance - the overall execution time, instructions, L1, L2 and TLB misses

application - the allocators' accuracy for each application

container - the allocators' accuracy for each STL container

locality accuracy - the variance of locality accuracy impact on page and cache accuracies

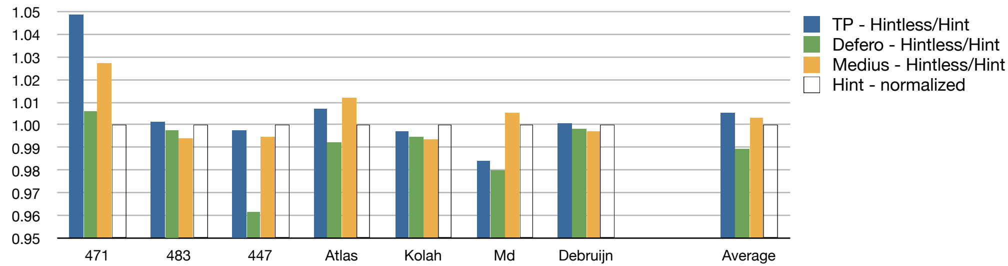


Fig. 34. Hintless execution time normalized to hint allocation (lower is faster)

Each of these four perspectives contributes to the overall comparison: the performance has the bottom line, the applications reflect the allocators' ability to exploit their locality, the containers show how and what type is affected and finally the locality precision variance shows how it impacts with allocators' accuracies.

1. Performance Comparison

Fig. 34 shows the hintless execution time for the three allocators over the seven benchmarks normalized to the hint allocation scenario. Overall, hint allocation is slightly faster for TP and Medius, and slower for Defero, but none statistically significant. For 471.omnetpp all allocators performed better with hints, up to 5% for TP. Since the average variance is within the standard deviation of 1%,we conclude that hintless allocation performs as well as hint allocation. The remainder of this chapter explains why this is so.

We start by isolating the *costs* from the *benefits* in the two scenarios. Qualitatively, the cost difference between hint and hintless allocation is the extra search for the hint's K-region. This cost is reflected in the issued instructions. Fig. 35 shows the average number of issued instructions for the hintless allocation normalized to hint allocation. Hint allocation is slightly more costly than hintless, for all allocators. This explains in part the hint allocation's performance slowdown. The remaining

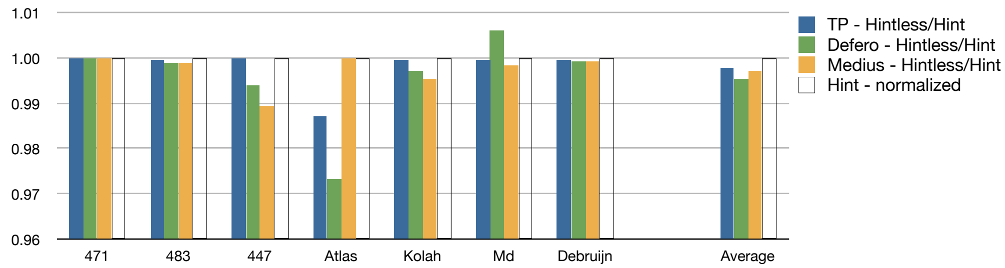


Fig. 35. Hintless issued instructions normalized to hint allocation (lower is faster)

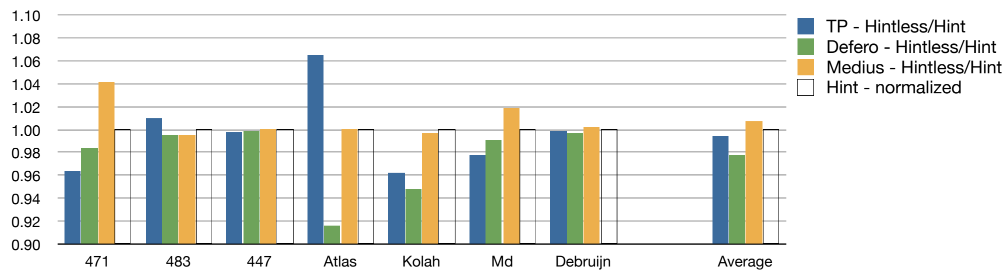


Fig. 36. Hintless L1 misses normalized to hint allocation (lower is better)

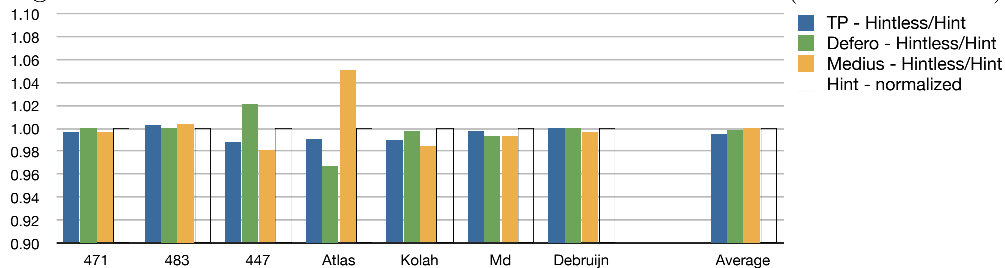


Fig. 37. Hintless L2 misses normalized to hint allocation (lower is better)

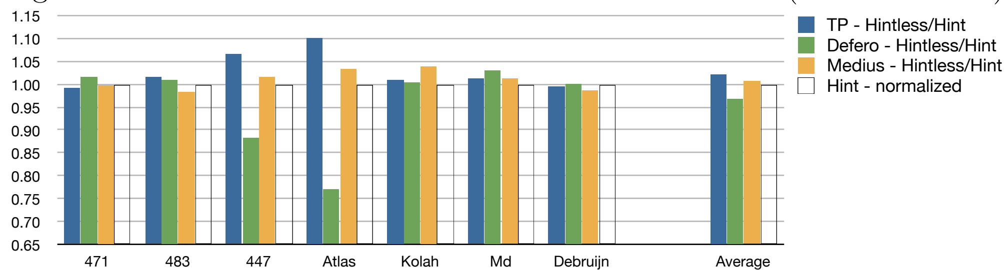


Fig. 38. Hintless TLB misses normalized to hint allocation (lower is better)

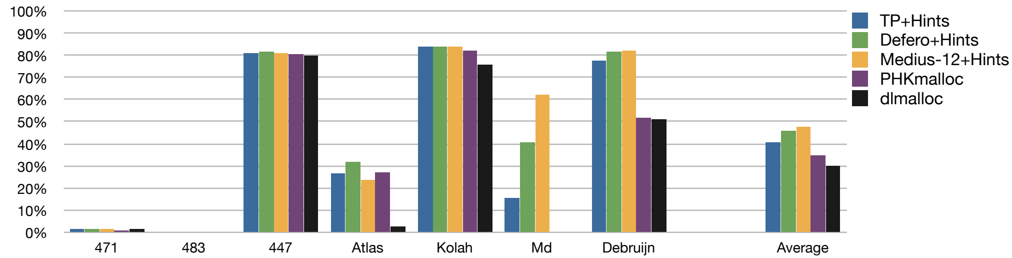


Fig. 39. Page accuracies for Hint allocation

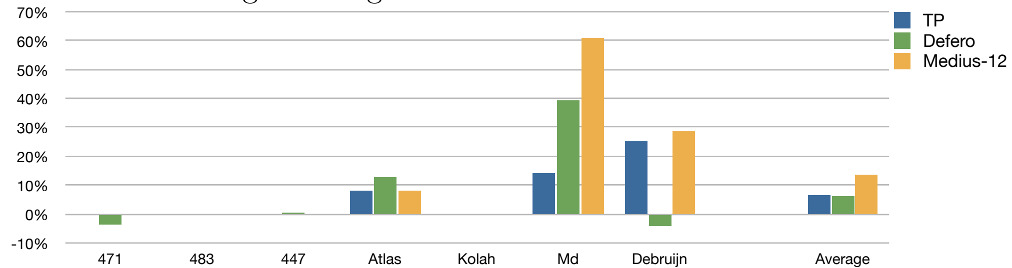


Fig. 40. Page accuracies improvement of Hint over Hintless allocation (Hint - Hintless)

part comes from the memory latencies imposed by these extra instructions, due to touching different cache lines or virtual pages. For example, Defero touches several different K-regions in the top-down tree search for a hint's region.

Hintless Defero is 0.45% faster than its hint counterpart, Medius 0.26% and TP 0.21%. This shows that TP spends the least amount of effort in finding the hint's vicinity, followed closely by Medius, while Defero spends the most effort. This order is preserved for the average number of pages touched per allocation: TP with 1.3, Medius with 2.7 and Defero with 3.2, see see fig 9, 25 and 16. Fig. 36, fig. 37 and fig. 38 show the same ratio of hintless over hint allocation for L1, L2 and TLB misses. All three show the same trends as the execution time and instructions.

2. Applications Comparison

The page accuracy measures the ability of an allocator to collocate a new block in the hint's virtual page. Fig. 39 show the page accuracies for TP, Defero, Medius,

PHKmalloc and dlmalloc respectively, for the seven benchmarks when allocating with hints. Fig. 40 shows the page accuracy improvement of hint over hintless allocation. Allocating with hints increases page accuracy for all allocators by an average of 7%, 7% and 14% respectively for TP, Defero, and Medius-12. This relative improvement corresponds to 21%, 39%, and 40% respectively. The hint switch impact on page accuracy occurs for Atlas, Md and Debruijn. The remaining four applications do not observe a significant page accuracy change. Interestingly, Defero on 471.omnetpp and Debruijn shows a decrease in page accuracy when allocating with hints.

These results show that all allocators increase on average their page accuracy when using hints: TP from 34% to 41%, Medius-12 from 34% to 48% and Defero from 39% to 46%. In this comparison, all three allocators use the same size segregation, the same K-region size, 4 KB, and maintain all other parameters the same. Thus, the performance difference is directly attributed to hint switching.

3. Containers Comparison

We now evaluate the impact of switching between hint and hintless allocation on four container types: deque, list, tree and vector. Our results indicate that tree and list are the most sensitive to allocation hints from a page accuracy viewpoint, while deque and vector are only slightly sensitive. Fig. 41 shows the page accuracy for these four types of containers with hint allocation with TP, Medius, Defero, dlmalloc and PHKmalloc. List has the highest page accuracy, with 70% of its allocations landing in the hint's virtual page. Deque and tree follow with a 50% page accuracy, while vector's page accuracy is below 10% for all allocators. Fig 42 shows the improvement of page accuracy of hint allocation over hintless for TP, Medius-12 and Defero, categorized for each container type.

All containers increase their page accuracies with hint allocation, except for deque

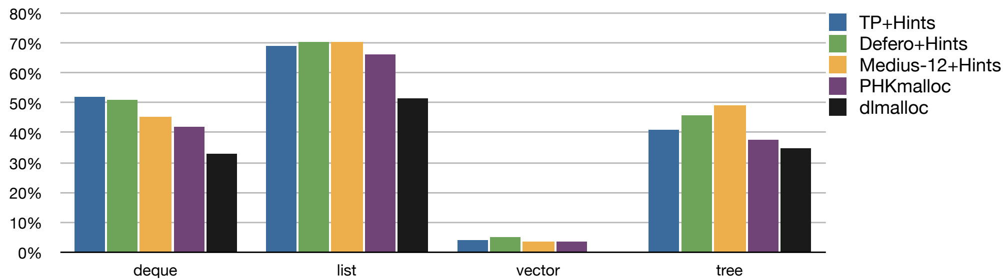


Fig. 41. Page accuracies with Hint allocation for different types of containers

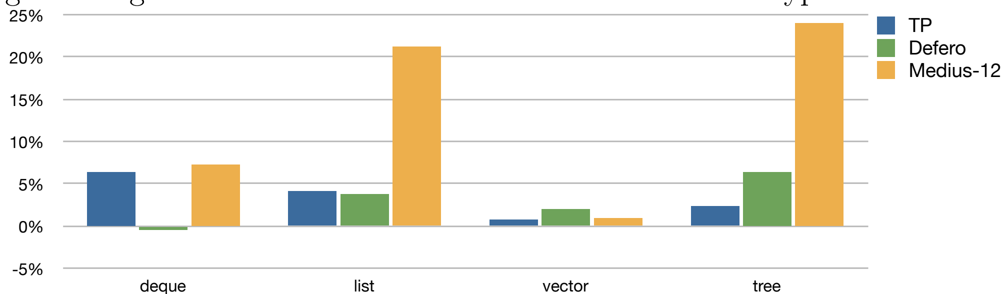


Fig. 42. Improvement of page accuracy of Hint over Hintless for different types of containers

with Defero, which observes a 0.4% deterioration. Medius-12 is the most sensitive to hints, improving the page accuracy when using allocation hints by more than 20% for lists and trees. Except for vector, the other containers have a high page accuracy. This is partly because their small size, and partly because of the allocation hints. List has the highest page accuracy, followed by tree, deque and vector. Vector benefits from all three allocators in both scenarios and given that they account for two thirds of allocations, the importance of this result gets magnified. Hint allocation increases vector's page accuracy by 5.1%, 3.8% and 4.1% for Defero, Medius and TP respectively - for dlmalloc and PHKmalloc this value is 0% and 3.9% respectively.

4. Locality Accuracy Comparison

We now evaluate how the allocators' locality accuracy affects performance. We varied it for our three allocators and measured the impact on page accuracy. For TP we

varied K between 14, 12, 10, 8 and 7, for Defero between 20, 16, 12 and 8, and for Medius between 22, 20, 16 and 12. These series have a common value of 12, corresponding to the size of the virtual page. For this value, the differences between allocators are solely attributed to their allocation strategies, since they share the same region size and size segregation.

Fig. 43 show the average page and cache accuracies for TP, Defero, and Medius with *hintless* allocation (the same accuracies with hint allocation are shown in fig. 32). This picture shows that when allocation without hints, TP, Defero, and Medius have approximately the same accuracies as PHKmalloc, and higher than dlmalloc's. Fig. 44 shows the variation of page and cache accuracies when switching from hintless to hint allocation. Defero and TP benefit from hint allocations in all configurations, up to 9% and 7% for Defero-12 and TP-8 respectively. However, Medius' three configurations (Medius-22, Medius-20 and Medius-16) actually decrease their page and cache accuracy when using allocation hints. We attribute this difference to the allocation/deallocation pattern that could create a different order of size classes in Medius in the two allocation scenarios. Since all three configurations have K-regions larger than 64 KB, their size classes spread over multiple pages, so the block's order within a size class is not guaranteed. Medius-12 on the other hand is the most sensitive of all allocators' configurations with almost 10% improvement in page accuracy when using hints. Medius-12 guarantees the hint's page upon a hit, thus increasing the page accuracy substantially over the other configurations.

Hint allocation produces a higher page accuracy than hintless for all allocators. The best page accuracies for hint and hintless allocations, as well as the largest difference between them, occurs in page conscious configurations, i.e. 12-regions, in all three allocators. Locality precision impacts the page accuracy for hint and hintless allocations in different ways for each allocator. The page accuracy difference

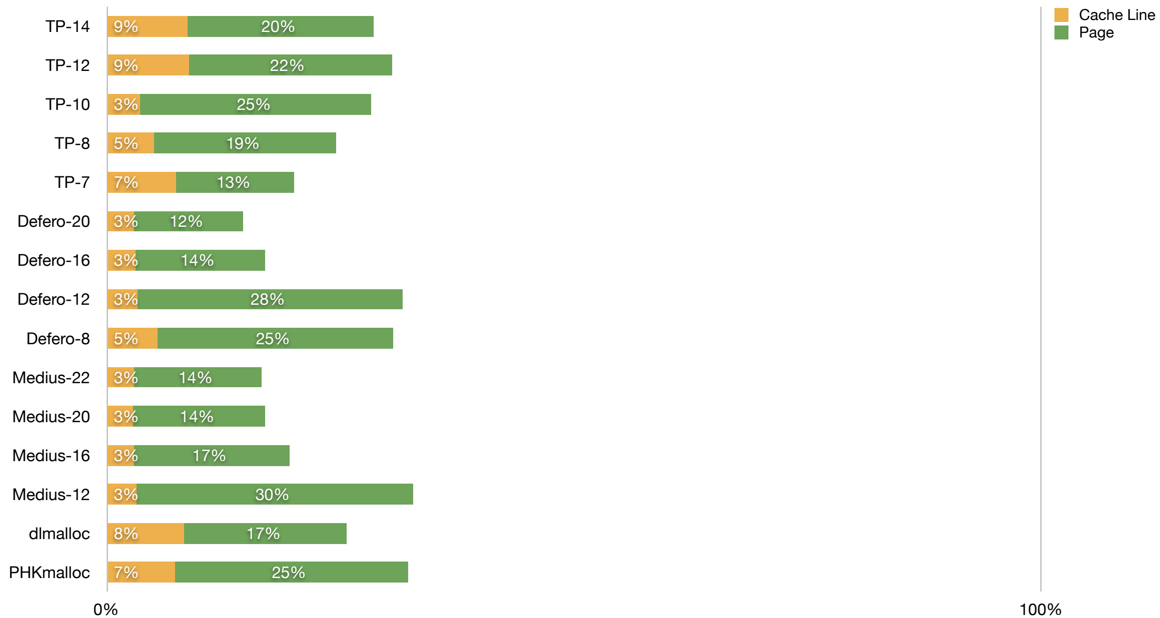


Fig. 43. Average page and cache accuracies for Hintless Defero, Medius and TP as locality precision varies

is between 2% to 9% with Defero, while Medius' range is wider, -2% to 9%. TP, in contrast, is the least sensitive to locality accuracy variance, with the smallest range difference of 2% to 7%.

The cache accuracy, depicted as Cache Line in fig. 44, shows a smaller variance to different allocator's configurations and to the hint switch. It is not until K-regions become sufficiently small that the chances of landing in the hint's cache line increase. Their variance to hint switching is also insignificant, with less than 0.1% for values of K larger than 16. Even for values of K larger than 12 the cache accuracy variance on hint switching is very small. Defero-8 increases the cache accuracy by 1% when using hints. The other allocators' cache accuracy remains almost the same. This is also in part because none of the configurations have K-regions as small as a cache line, which on our system is 64 bytes. TP-7 has 128-byte regions, and this configuration has the largest cache accuracy improvement of all TP's configurations, with 0.1%.

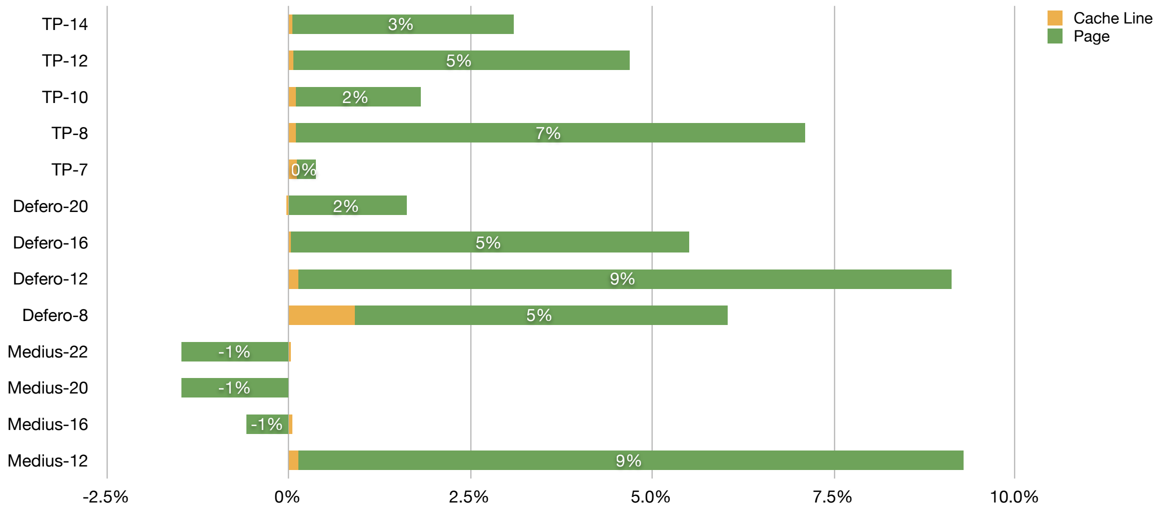


Fig. 44. Average improvement of Hint over Hintless of page and cache accuracies for Defero, Medius and TP as locality precision varies

A finer size distribution for objects less than 128 bytes explains further the poor cache accuracy. Fig. 45 shows the size distribution for these small objects into four categories: 1) less than 24 bytes, 2) less than 32 bytes, 3) less than 64 bytes and 4) less than 128 bytes. The percentages are reported from the total number of objects less than 128 bytes. Please note that only objects less the 32 bytes are subject to cache line collocation, as two larger objects would not fit in the cache line. Deques' cache line collocation potential is close to 90% of their small objects, but it is their deque tables, not their deque arrays - each holding data in 512-byte size- that are collocated in the same cache line. This cache line collocation happens when deques grow and hash tables expand, and it does not really help locality since they do not hold data. Lists and trees have 70% and 57% of their small objects able to be collocated in a cache line, while vectors could collocate 75% of their small objects. However, since lists usage is smaller than 1% and vectors provide hints to less than 20% of its allocations, as discussed in the next section, that leaves trees as the only viable container that could benefit from cache collocation.

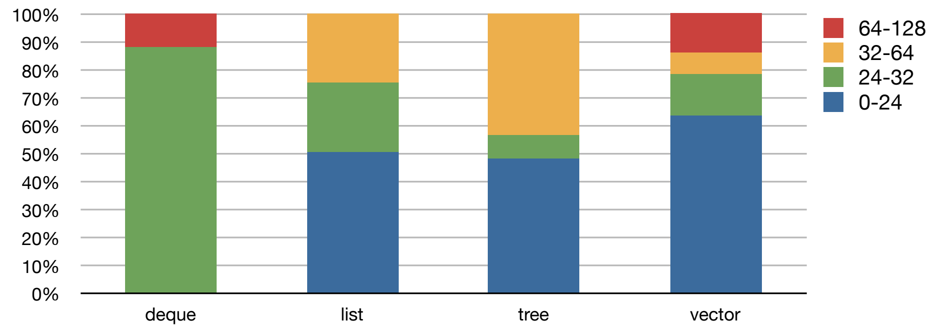


Fig. 45. Size distribution for objects smaller than 128 byes for each container type

Furthermore, the probabilities of cache collocation are much smaller in practice. These statistics show the theoretical cache collocation potential. However, the allocators further reduce these numbers since they do not guarantee cache collocation for cache colocatable objects. For example, Defero-24 has 256-byte regions and assuming that the hint's region is available and found, then the probability to allocate a block within the hint's cache line is 25% with a uniform distribution. To obtain the probability that an object and its hint get collocated in the same cache, one has to multiply this 0.25 factor with the size distributions for objects less than 32 bytes, presented above.

Our findings in the hint vs. hintless comparison can be summarized as:

- hint and hintless allocations perform equally well, in part because allocators run faster without hints
- hints allocation increases page accuracy over hintless
- hints do not affect cache accuracy.
- page sized precision maximizes page accuracy in both scenarios, hint and hintless, as well as their difference

These trends were consistent across all three allocators. Our findings explain why hint and hintless allocations perform equally well from an allocator's point of view. We next shift the focus of the same comparison onto how the STL provides the hints to the allocator in the first place.

C. STL Usage

In this section we examine how applications use STL containers and how these allocate memory. While our findings show that the STL audience is knowledgeable about the locality problem and uses mostly vectors, they also show that memory allocation opportunities for improving STL containers' locality are mostly constricted to trees.

1. STL Containers' Memory Profile

Tables XI and XII present the STL containers' allocation statistics for the seven benchmarks. They both show the number of instantiated containers, the amount of requested memory, the maximum memory in use, the number of allocations, the maximum number of objects, the number of allocations per container and the average allocation size. These statistics are detailed for each application and container type. Neither the total maximum memory in use nor the maximum number of objects are reported as sums of containers' values since these values might occur at different times. Since STL sets, multisets, maps and multimaps are all implemented as red-black trees, we bundled their values in one, trees[54]. Only the number of instantiated containers is kept individually for these containers. Fig. 46 shows the instantiated containers depicted as percentage from the total number of instantiated containers. The most used containers are undisputedly vectors, which account for 84.5% of all STL containers. The second most used containers are maps, with 14%, followed

by sets with 1.3%. Lists and deques account for the remaining 0.1%. This container usage is hardly surprising given vectors' simplicity and performance. Fig. 47 shows the percentage of allocations originated from each container type from the total number of allocations issued by all STL containers. Vectors initiate 62.5% of all allocations, followed by trees with 37%, while lists and deques claim only 0.5%.

Fig. 48 shows the amount of memory allocated by each container type as percentage from the total memory allocated by containers. Vectors use 73% of the total amount of memory allocated by STL containers, followed by trees with 25%, while lists and deques use the remaining 2% of the memory. This memory distribution is roughly similar to the allocation distribution, which implies that the amount of memory used by each STL container is proportional to its activity with its memory allocator. Fig. 49 shows the container participation into the size distribution. Perhaps surprisingly, all objects allocated by trees and lists are less than 128 bytes, and thus making them more susceptible to locality issues. Half of the objects allocated by deques is for deque arrays, which have a fixed size of 512 bytes in GNU 4.1.2 STL, while the other half is for deque entry arrays, which are less than 128 bytes. Vectors are the only containers to allocate all sizes. However, the majority are small, with 82% of their allocations less than 128 bytes.

These statistics show that vectors are the most popular containers in instantiations, allocations, and amount of used memory. Perhaps surprising, they also show that most vectors are small, less than 128 bytes. The second most popular containers are trees, which use about a third of the resources, in all categories. Another unexpected statistic is that lists and deques are minimally used, which makes trees the only STL containers practically adequate for locality improving, since vectors are already compact. Containers' profile establishes, in part, the locality improving opportunities for a memory allocator. However, another important aspect is the STL

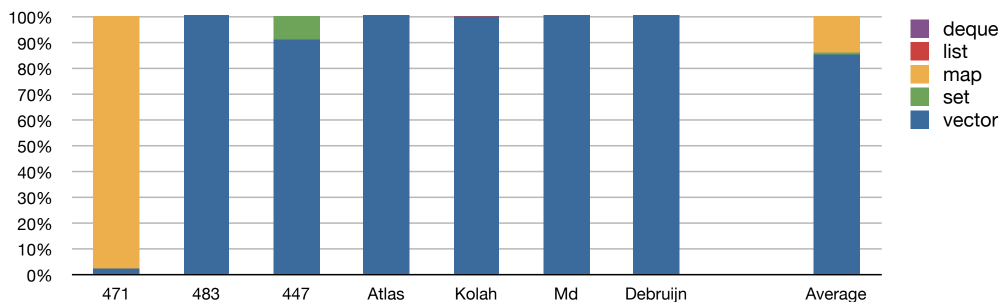


Fig. 46. Percentage of instantiated STL containers

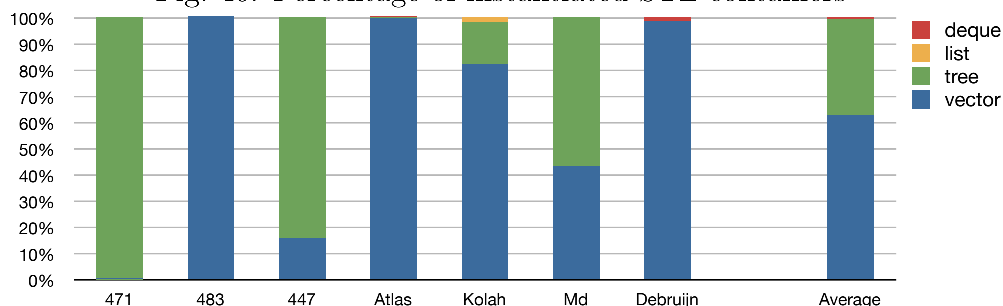


Fig. 47. Percentage of allocations originating from each container type

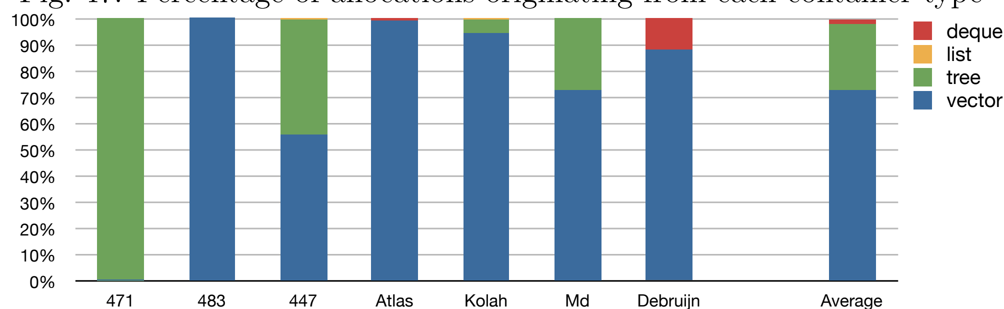


Fig. 48. Percentage of memory used by each container type

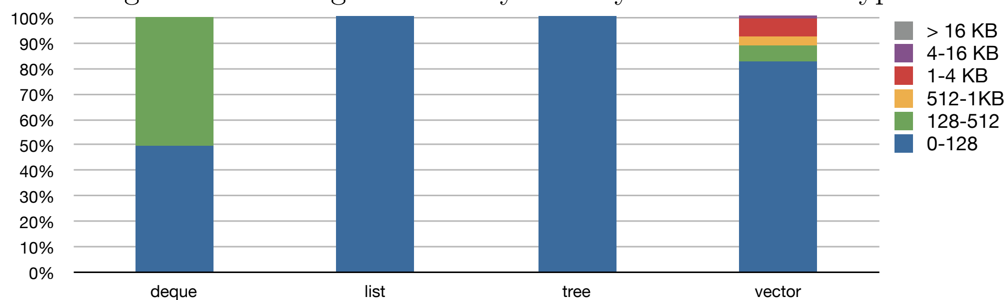


Fig. 49. Size distribution per container type

containers' ability to provide hints to their allocators. The next section addresses this issue.

2. Containers' Effectiveness at Providing Hints

The ability to *use* allocation hints efficiently belongs to allocators and we discuss it in section B, but the ability to *supply* the hints belongs to STL containers and we discuss it in this section. We define the *containers' effectiveness* as the percentage of STL allocations with non-zero hints from the total STL allocations. Fig. 50 shows this ability for the seven benchmarks and their average. The benchmarks span the whole range of effectiveness, from very effective, such as 471.omnetpp, which supply hints to 98% of its allocations, to least effective, such as Debruijn and 483.xalancbmk, which supply hints to less than 2% of their allocations. On average, one in two allocations have a hint.

Fig. 51 shows the containers' effectiveness broken down for each container type. The average was computed only for the applications that use that container type. Containers vary in their abilities to provide hints from very effective, such as lists and trees with 94% and 88% effectiveness, to less effective, such as vectors with 18%. This is because trees and lists provide hints for all their allocations except their header's, while vectors, in contrast, provide hints only for copying operations[1]. Deques are in the middle with a 60% effectiveness.

The containers' effectiveness also shows their usage behavior. Consider vectors for example, with an effectiveness of 18%, a weak context when compared to others'. Since only the copy constructor and equal operator provide hints, a 20% context

[1]Vectors could provide hints for all their allocations, but these hints would be addresses to blocks of different sizes, which are not exploited by allocators with a size segregation, which is what we used in this article.

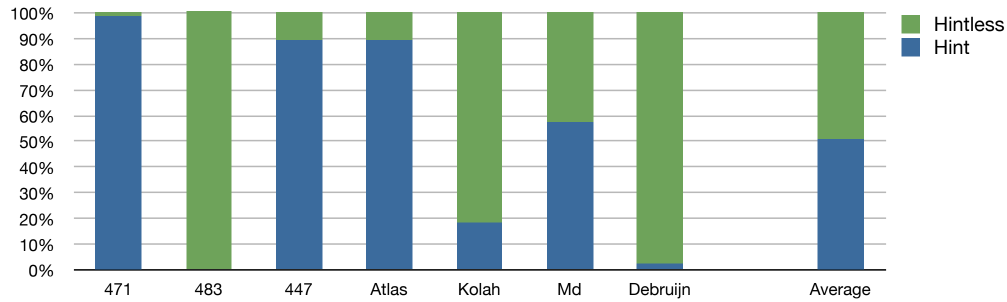


Fig. 50. STL containers' effectiveness per application

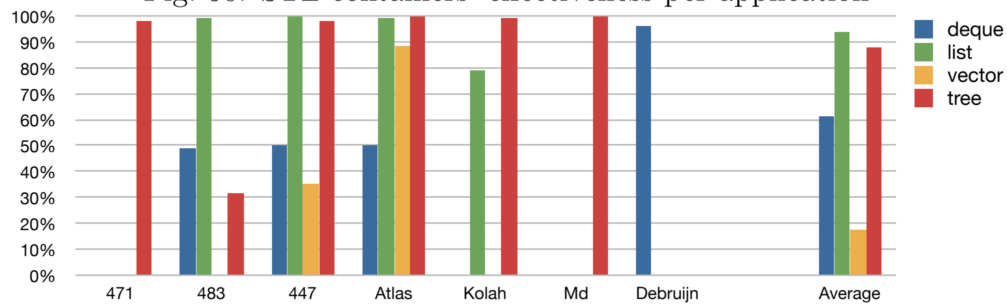


Fig. 51. Effectiveness for STL deques, lists, vectors and trees

effectiveness means that 20% of all vector allocations were for copying purposes, such as `a=b` or parameter passing `foo(a)`, while the remaining 80% for initializations or changing the size of the vector, such as `a.push_back(7)`. The same rationale lets us infer that Debruijn allocates deque arrays extensively, due to a high deque's effectiveness of 95%, while the 50% deque's effectiveness for the remaining applications implies a small number of allocated arrays.

Containers' effectiveness reflects the potency of STL containers to supply allocation hints coupled with their operations' usage by an application. Lists and trees are the most effective, while vectors the least. Containers' effectiveness also provides a glimpse into how they are used by an application. While containers' profile establishes, in part, the locality improving opportunities for a memory allocator, it also describes only half of the picture. The other half, which directly affects the impact of

such opportunities and techniques, is how these containers are used by applications. The next section addresses this issue.

3. Containers' Dynamism

This section presents a metric that measures how important spatial locality is for a container. The metric characterizes the dynamic property of containers: a vector that is allocated in memory once and traversed thousands of times is more static (or less dynamic) than a list whose elements are rapidly inserted and deleted, with very few traversals. The proposed metric characterizes containers' and applications' behavior to better understand their spatial locality needs. With this metric, an allocator could speed up allocations for containers or applications that are not as dynamic, and concentrate more on spatial locality for those who are.

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, denoted by D, as a measurable metric. The dynamism of a container represents the ratio between the modifying and non-modifying operations:

$$D = \frac{M}{NM}, with D, M, NM \geq 1^3 \tag{5.1}$$

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

[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 also at least 1 NM operation: if it didn't, the container would never be accessed and thus would be dead code.

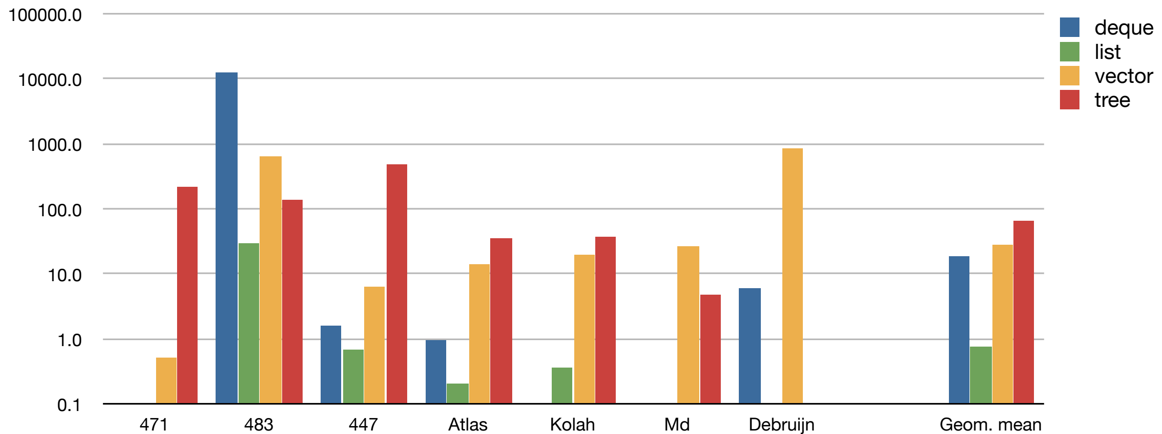


Fig. 52. Dynamism of STL containers

Container's dynamism is highly correlated memory behavior. The higher the dynamism, the more important the allocation speed, while data locality diminishes in its importance. The lower the dynamism, the more important locality, while memory allocation's speed becomes less dominant.

Fig. 52 shows the dynamism for each container type and for each of the seven benchmarks in a logarithmic scale. This dynamism measures only the iterator operations as non-modifying, and leaves out the other non-modifying container operations, such as `operator[]` in vector. Lists are the most dynamic containers that are traversed few times before they change. Deques and trees are relatively more static and are traversed more times before their structures change. We surmise that vectors are the most static containers, despite fig. 52 showing that they are less static than trees. This is because our measurement includes only the iterator traversals as non-modifying operations, while the most used operation of accessing vector's elements, `operator[]`, is left out. The same argument applies for deques.

Our findings in the analysis of STL usage can be summarized as:

- vectors are the most used containers
- STL containers provide hints to half of their allocation
- trees and lists are the most effective at providing hints, while vectors the least
- while lists are the most dynamic, trees, deques, and vectors are more static

D. Discussions

Hint allocation performs as well as hintless allocation. From a cost perspective, hint allocation is more expensive. The costs to locate a spatial vicinity compete directly with the benefits, not only in the extra instructions but also in the memory latencies these instructions impose. From a locality perspective, hint allocation improves page accuracy over hintless, for all allocators and in all configurations. The locality improvement depends on the effort invested. Medius-12 has the highest page accuracy improvement switching from hint to hintless, but it also spends the largest allocation effort of the three allocators. TP reduces this page accuracy difference between hint and hintless, but spends the least allocation effort.

From a container perspective, trees are the only viable containers for locality improvement from hint allocation. They have a high effectiveness and cover one third of the allocations and the memory. Deques, in contrast, have a lower effectiveness and are the least popular in terms of allocations. Lists and vectors fall in between. Despite a high container effectiveness and page accuracy improvement with hint allocation, lists take only a minute role in the locality improving technique, which downplays their strengths. Vectors have the weakest effectiveness, and only their copy operations benefit from hint allocation, reducing the opportunities for explicit locality improvement. All these results are further amplified by their dynamism. Vectors and

deques are the most static and thus need and already have the most compact data layout. Trees and lists, which are more dynamic, benefit more from locality improving techniques.

From an allocation strategy perspective, the three allocators we use in our study stand as potent techniques for hint allocation. They improve locality and execution time over one of the best allocators available, in dlmalloc[53, 8], and over other state-of-the-art allocators, such as FreeBSD's allocator[46]. These allocators show the same trends, though some with more pronounced values than others.

And finally, from an application's locality perspective, hint allocation has an edge over hintless if the application exhibits poor temporal locality. Hint allocation is unassuming about the application's locality of reference and increases locality for all kinds of applications. For example, in 483.xalancbmk containers' effectiveness is almost zero, yet the overall execution time is reduced by 5%. We attribute this reduction entirely to the application's temporal locality overlapping the allocators' locality of reference. In another example, Atlas, the 20% improvement in execution time is attributed to the same circumstances.

For hint allocation to outperform hintless allocation, several things have to hold:(1) high use of dynamic containers, such as trees or lists, (2) an application should have limited temporal locality, and (3) the time spent in these dynamic data structures should be considerable. Lack of any of these conditions impedes an allocator from beneficially use allocation hints to increase locality, and a careful design of allocators' temporal locality suffices.

At a first glance, one could hastily infer that using address based memory management is unprofitable. However, a closer look shows the contrary. First, the performance of hintless allocation is predicated by high locality of reference, which requires a diligent partition of free blocks into spatial regions. This efficient partitioning

is possible only if the memory management uses addresses as organizational markers. Despite ignoring allocation hints, hintless Defero, Medius, and TP still use the blocks' addresses to keep them partitioned. The second argument is that while allocation performs equally well with and without hints, the deallocation benefits from address based allocation, since it uses exactly the same technique to return a block to its partition as allocation uses it to locate a certain vicinity. A fast address search benefits *both* allocation and deallocation. Thus, hint based allocation is needed to improve the performance of 1) locality of reference and 2) deallocation.

E. Summary

This chapter presents a cost-benefit analysis of locality improving with memory allocation, by directly comparing two allocation techniques. The first one, hint allocation, collocates a new block within a hint's vicinity. The second one, hintless allocation, only uses the allocator's locality of reference and ignores the allocation hints. In this comparison, the allocators change only the allocation hints, and keep the other parameters the same. This setup allows for a fair comparison between hint and hintless allocation. While hint allocation improves page accuracy over hintless by up to 40%, its costs offset the benefits. Careful considerations for locality of reference within an allocator performs on average as well as explicit locality improving allocation.

Hint-based locality improving techniques require integration into standard modules, such as STL containers, which automatically supply spatial allocation hints to their memory allocators, avoiding the error-prone and impractical approach of manual intervention. However, we show that STL audience is knowledgeable about locality issues and uses mostly vectors, limiting the opportunities for explicit locality improving through memory allocation. This study raises a further question: are STL containers

effective enough at providing hints to compensate for the costs of exploiting them? It could be that STL design favors temporal locality and inherently hintless allocation. Nevertheless, further analysis on more applications is required before excluding STL as a prolific context for proving allocation hints.

However, regardless of the effectiveness of the STL context, the locality improving techniques presented in chapter IV, namely, TP, Defero and Medius, benefit both scenarios of hint and hintless allocation.

Application	Containers Instantiated	Memory Requested (KB)	Max. Memory in use (KB)	# Allocations	Max. Objs.	Allocations / Container	Average Size (bytes)
Debruijn							
deque	45	4,367.5	112.9	8,743	222	194.28	511.54
vector	60,003	30,937.7	16,172.1	420,006	60,003	6.99	75.42
Md-sandia							
rb-tree	1	90,869	312	4,652,503	16,009	4,652,503	20.00
vector	2,168,142	235,269	3,727	3,495,035	40,005	1.61	68.93
Atlas							
deque	313,408	168,336.1	67.3	630,558	135	2.01	273.37
list	1	4.0	0.53	128	17	128	32.00
rb-tree	747	15,354.8	4,940.2	464,860	109,116	622.30	33.82
vector	379,681,296	14,629,286.4	683,451.3	155,533,234	2,482,109	0.40	96.31
Kolah							
list	61,216	4,218	2,812	180,000	120,000	2.94	24.00
rb-tree	139	58,065	26,228	1,928,500	1,025,106	13,874.10	30.83
vector	19,952,875	1,023,020	657,917	9,623,849	3,228,360	0.48	108.85

Table XI. Statistics of STL allocations

Application	Containers Instantiated	Memory Requested (KB)	Max. Memory in use (KB)	# Allocations	Max. Objs.	Allocations / Container	Average Size (bytes)
471							
rb-tree	5170	10,378.5	92.5	269,629	27,656	52.15	39.41
vector	84	4.5	0.046	252	2	3	18.66
483							
deque	801	426.9	11.5	1,605	42	2.00	272.37
list	4	9.5	9.5	409	409	102.25	24
rb-tree	2,070	7.4	2.9	203	76	0.09	37.12
vector	43,289,464	58,432,617.6	192,841.1	85,177,616	365,250	1.96	702.47
447							
deque	2	1.0	1.0	4	4	2	272
list	217	30,426.1	3,444.0	649,254	73,475	2991.95	47.98
rb-tree	2,005,382	2,408,774.3	256,603.3	121,808,088	13,138,091	60.74	20.24
vector	19,513,486	2,988,145.7	27,497.3	22,180,460	82,893	1.13	137.95

Table XII. Statistics of STL allocations (cont'd)

CHAPTER VI

GENERIC MEMORY ALLOCATION

Existing techniques are concrete rather than generic, and limit the exploration of allocation strategies to a traditional way of looking at memory blocks: based only on their size. A generic approach allows viewing and exploring the memory allocation problem from various angles, such as allocation call site, thread identifier and memory address. Generic basic building blocks are thus required to assist programmers in exploring novel strategies.

In this chapter we present a theoretical formalism and an accompanying software library, named Allotehque, as a generic approach to the memory allocation problem. Allotheque is to memory allocation what C++ Standard Template Library is to programming: it allows the users to focus on the allocation strategies rather than the implementation. Fig. 53 shows two examples of the allocation and deallocation requests using the traditional way and our proposed generic approach. Traditional

```
// Traditional allocation & deallocation requests
1. int *a1 = new int;    ...   delete a1;
2. void *a2 = malloc(8);  ...free(a2);

// Generic allocation & deallocation requests
3. int *a3 = Allocator::allocate(Request<CallSiteType,int>(cs,4));
4. Allocator::deallocate(Request< CallSiteType,int>(cs,4));
```

Fig. 53. Example of generic memory allocation and deallocation requests

allocations consider 'size' as the only relevant parameter[1], lines 1-2, while generic allocations accept any parameters in a request, such as call site and size, line 3-4. These generic allocations, however, represent only the application's interface to an allocator. For allocators to satisfy generic requests, they first need to manage their memory based on these generic parameters. Allotheque helps users build such allocators. For example, Defero's internal structure can be implemented in two lines

```
1.typedef rb_tree<kbit<K>,list<void>,match> tk;
2.typedef hash_table<size,tk,match, hf_segregated_d> Defero;
3.typedef hash_table<size,list<void>,match, hf_poweroftwo>
                                              Kingsley;
```

Fig. 54. Defero's and Kingsley's memory management schemes

of code, Fig 54, line 1-2. Or, the internal structure of Kingsley's allocator used in BSD 4.3[34, 55] can be implemented in a single line of code, Fig 54 line 3. Having the problem reduced to several lines of code allows one to focus on the allocation strategy.

While Allotheque helps building allocators, our theoretical formalism to generic memory allocation allows designing their blueprints. It also allows them to effortlessly compare different schemes and understand the different data structures and allocation strategies. The generic approach is also intended to be an intermediate language that consistently describes allocators. With the multitude of memory allocation mechanisms developed over the years, it is getting harder to manage and

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

compare existing mechanisms. Our generic approach glues them together and coherently organizes them. If we consider a new memory allocation scheme as a *vertical* development, then the generic comparison is a *lateral* development that connects the vertical ones. We show how to build and compare several well-known memory allocation schemes, such as sequential fits, buddy systems, or region-based, along with our newly developed ones.

A. Generic Memory Management and Allocation

We generalize the memory management and allocation problem by breaking down the problem into three independent components:

1. block attributes
2. memory partitions to store them
3. allocation predicates to search them

We argue that any memory allocator can be decomposed into these three components.

1. Attributes of a Block

Each memory block is described by attributes, such as size (used by all traditional allocators), address (used by TP, Defero,and Medius), the program call site (used as a lifetime predictor by Barret et at. [6]), or the stack pointer (used by Seidl et at. [64]). An attribute is any relevant property of a memory block[2]. These attributes act as handles with which an allocator manages the blocks.

[2] There are attributes that are not relevant to memory allocation, such as the contents of a memory block. We refer only to attributes that are pertinent to the memory allocation problem.

For an allocator to satisfy a requested attribute A, it needs to manage its memory based on A. For example, the popular size segregation mechanism manages its blocks based on size alone, and cannot satisfy a request based on allocation's call site, for example. Hence, a memory request is limited by the attributes an allocator uses to manage its blocks. Traditional memory allocators use size as the only block attribute, which is reflected in the request's interface: `malloc(size_t)`.

a. Types of Attributes

We classify attributes into two categories, based on how they can be inferred from a block: 1) *implicit* or 2) *explicit*. This classification decides whether an attribute gets stored with the block. For example, the virtual page or the cache line attributes can be inferred implicitly from the block's address. Implicit attributes need not be stored. Size, on the other hand, is an explicit attribute that cannot be inferred directly from the block's address. Explicit attributes need be stored[3].

In addition, attributes can be classified as *mandatory*, such as size, and *optional*, such as call site or address. Mandatory attributes affect program's correctness. Size is the only mandatory attribute for uniprocessors. Optional attributes do not affect program's correctness but do affect program's performance. They are byproducts of computer's design and can be used by a memory allocator to improve performance. In our generic memory allocation approach, all concrete attributes are instances of a generic attribute concept. Free memory blocks are handled generically using this concept.

[3]Size can be stored in each block as in the sequential-fits [49, 69] or once implicitly in a hash table for all blocks in the bucket, as in the segregated approaches[20, 59].

2. Memory Management with Partitions

The second component of our generic approach is the memory partition. We regard the memory address space as a collection of generic attributes and we use set theory [37] to partition it. This partition helps an allocator to search for specific attributes. For example, the segregated lists approach organizes same sized blocks into a linked list, which speeds up the search for size.

a. Equivalence Classes

We use algebraic equivalence relations to partition the memory space into equivalence classes[36]. An example of an equivalence relation is 'modulo 2', which partitions a set into two equivalent classes, each holding elements of the same parity. An example that is not an equivalence relation is '$\leq$', since it lacks symmetry. Memory blocks that have equivalent attributes are stored together in the same equivalence class. This partition is strictly maintained throughout the entire execution of the program, regardless of the allocation/deallocation pattern. For example, Defero, Medius and TP keep their blocks organized in K-regions regardless of the application's allocation pattern. This strict organization allows an allocator to exploit locality of reference, as we discuss it in chapter V. The equivalence partitions inherently posses this property. Equipped with a partition, an allocator can search it for a certain attribute.

Well known memory management approaches, such as the segregated lists or the buddy systems, can be expressed with equivalence relations. We describe the most popular ones in section D. The algebraic partition creates a generic management of blocks. For example, partitioning the memory into virtual pages requires the use of address attribute and an equivalence relation R_{page}, which creates an equivalence class for all blocks within a certain page. The K-regions used by our allocators are

equivalence classes, created by R_{K-bit} relation (two addresses are equivalent iff they share the most significant K bits).

b. Storing Blocks in Containers

The equivalence classes are stored in a class container, which we denote by *C-container*. For example, the hash table used by Defero and TP to store the blocks of a certain size class is a C-container. The equivalent elements are stored in an element container, which we denote by *E-container*. For example, the list used by Defero to store blocks within a K-region is E-containers. An E-container should favor insertion and deletion, which are expected to be the dominant operations. A C-container should favor fast search operations.

c. Partitions

A memory space can have more partitions. For example, Defero partitions the space into both size classes and K-regions. Multiple partitions are required to satisfy requests for multiple attributes. For our locality improving allocators, the requests had two attributes, a size and an address. We identified two types of multiple partitions: (i) *recursive* and (ii) *simultaneous*.

d. Recursive Partitions

In recursive partitioning, each equivalence class is further partitioned using a new equivalence relation. For example, Defero partitions the memory based on size and then each size segregated class is further partitioned using the R_{K-bit} relation. The recursive procedure creates a chain of nested partitions. The path to reach the most nested one goes through all partitions. We use the symbol "$\wedge$" to denote the recursive composition of equivalence relations. For example, suppose we have a set of elements

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

Fig. 55. Example of recursive partition

$S = \{0, 1, \ldots 5\}$ and the equivalence relation 'modulo 3', $R_{\%3}$. This relation creates three disjoint subsets, S_0, S_1 and S_2 as described fig. 55. Each subset can further be partitioned using the parity relation $R_{odd-even}$ (two elements are equivalent iff they have the same parity). $R_{odd-even}$ partitions each of the three subsets into two new $R_{\%3} \wedge R_{odd-even}$ equivalence classes, $S_{i,j}, 0 \leq i \leq 2, j =' odd',' even'$.

Composing equivalence relations is commutative, but the recursive partition is not commutative. That is $R_1 \wedge R_2 \neq R_2 \wedge R_1$. For example, Defero and Medius use the same two equivalence relations, one based on size and the other on address. The difference between these two allocators is that they commute the same two relations: Defero uses $R_{size} \wedge R_{K-bit}$ and Medius uses $R_{K-bit} \wedge R_{size}$. Yet their composition leads to very different management techniques.

e. Simultaneous Partitions

The second type of partitioning, simultaneous partitioning, organizes the same memory space into independent partitions, such as in TP's dual partition. This allows different partitions to coexist on the same memory space and at the same time. We use the symbol "$\vee$" to denote the simultaneous composition of equivalence relations.

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

Fig. 56. Example of simultaneous partition

For example, the two equivalence relations, $R_{\%3}$ and $R_{odd-even}$ described above, can divide S into two simultaneous partitions, $R_{\%3} \vee R_{odd-even}$, in which every element of S simultaneously belongs to two equivalence classes, as depicted in fig. 56.

In contrast to recursive partitioning, which requires the traversal of the recursion chain to reach a certain partition, simultaneous partitioning allows direct access to any of its partitions. For example, in TP's simultaneous partition, its memory can be accessed directly either by size or by address, and each method is independent of the other. We also use simultaneous partitioning to describe splitting and coalescing in section § D.

3. Memory Allocation

The third and last component of our generic approach is the actual process of allocating a memory block. Together with generic attributes and partitions, the generalized allocation can now be conceived as a request of N attributes as in "return a memory block whose attribute$_i$ has value $attr_i$ for every 1≤ i ≤ N". For example, a 3-attribute request might look like: "return a memory block of 8 bytes, in the same virtual page as x, but in different cache set than x".

We identified two types of allocations: (i) *prioritized* and (ii) *non-prioritized*, depending on the type of partition. Recursive partitions impose priority while simultaneous partitions do not.

a. Prioritized Allocation

An allocation request becomes prioritized when a memory allocator searches in a recursive partitioning. The search first finds the class of blocks whose attribute A have value a, and within this class the search then finds all blocks whose attribute B have value b, and so forth until the last partition in the chain is reached. This search prioritizes A over B and so forth. In the example presented on page 103, 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, and thus, the prioritization of 'modulo-3' over parity. We denote the prioritized relation with the symbol "$\succ$", where $a \succ b$ means that a takes priority before b.

Prioritized allocation, just like recursive partition, is not commutative: $a \succ b \neq b \succ a$. Defero prioritizes size over address. It first selects the right size class and within this class class it looks for a certain address. Even if there were a closer address in a different size class, Defero's allocation would not look for it, since size takes priority over address.

b. Non-Prioritized Allocation

An allocation request becomes non-prioritized when a memory allocator searches in a simultaneous partition. The search can select any partition and attribute. Thus, it does not prioritize one attribute over another. Once a block is taken out from a partition, it is taken out from the remaining partitions as well. TP's allocation is an example of non-prioritized allocation. It can allocate either based on size or address. TP with hints uses address first and size second, while hintless TP allocates based on size alone.

The strength of simultaneous partitions is that it allows different attributes to

be considered for different allocations. This strength is what makes TP fast, allowing it to quickly revert from address based allocation to size based.

A deallocation request returns a block back into its equivalence classes, for both recursive and simultaneous partitions. The deallocation request must contain all the attributes of that block.

c. Allocation Predicates

The actual search for a generic block can be described as a generic search in a container of attributes. We identified four components in this generic search:

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

The first component, the search traversal, is imposed by the C-container type, e.g. linear search for lists or top-down for trees. The remaining three components - start, target and the terminate predicate - are independent of the container type. We group these last three components together in what we denote by an *allocation predicate*. An allocation predicate informs the container where to start the search from, what to look for and when to stop looking for it. This process is very similar to an STL algorithm `for_each ( begin,end, bind1st(binary_predicate,target))`[56, 1]. The search can start either from the beginning of the C-container or the last visited element, which can be cached by the C-container. The allocation predicate encapsulates this decision and decouples the search from the starting position.

Container	Search Traversal	Description
List	linear	Beging to end
Tree	top-down	Top to bottom
Hash	lookup, linear	index, begin to end

Table XIII. Search traversals types used in memory allocation

Table XIII shows the most common search traversals, while table XIV shows several common termination predicates, such as equal or max. Defero uses a top-down traversal with an equal predicate for address, while Medius uses a lookup for hint's region and last recently used K-region, followed by a linear search in its hash table. In chapter V, the hintless allocation for Defero is implemented by replacing the 'equal' termination predicate used for hint allocation with 'true'. Our allocators do not use the linear search in a list, by other allocation strategies do, such a first-fits. PHKmalloc also searches its address-order lists linearly upon deallocation of a block[46].

d. Allocation = Search + Predicate

Conceptually, an allocation process consists of a traversal together with an allocation predicate. The decoupling between the traversal pattern and the allocation predicate allows one to change the C-container type in a partition, or to replace the allocation predicate, without affecting the design.

Most of the allocation strategies in the literature can be described using the generic allocation predicate. Table XV shows the most popular ones. For example, the classic 'first-fit' strategy is a linear search in a list from the beginning to end, together with the termination predicate of 'greater' [69, 77]. The search stops at the

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 XIV. Termination predicates

first block whose size is greater than the target. Another example is the 'worst-fit' strategy that selects the largest size in the list [77]. This strategy is a linear search and a 'maximum' predicate.

'Next-fit' strategy is a type of search that does not start from the beginning of the container [49]. This strategy is similar to 'first-fit', but it starts searching from where the last search left off, instead of the beginning of the list as in 'first-fit'. This strategy is implemented using an allocation predicate that signals the container to store the last visited element and start the subsequent search from this element. More details about how this is implemented are provided in appendix A.

Changing the search strategy is equivalent to replacing an allocation predicate. The modular separation between the partition containers and allocation strategies allows for experimenting with various schemes at the implementation 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 block
first-fit	(begin-end)(greater)	First greater than target
next-fit	(last-end)(greater)	First greater than target
best-fit	(begin-end)(min)	Smallest larger fit
worst-fit	(begin-end)(max)	Largest block
match	(begin-end)(equal)	First equal to target

Table XV. Allocation predicates

B. A Formalism for the Generic Memory Management and Allocation Problem

We now formalize the memory management and allocation problem using the three components we previously described, namely attributes, partitions and allocation predicates. We argue that any memory allocation problem can be formulated using this formalism. This generic approach allows an allocator to service any contextual attributes deemed relevant.

1. Generic Management

Let $A_1, A_2, \ldots A_n$ be n attributes that describe a memory block for memory allocation purposes. These attributes need not be unique. Let $R_1, R_2, \ldots R_n$ be n equivalence relations defined on a memory space $M = i, 0 \leq i \leq 2^{32}$, one for each attribute. The partitions created by these n relations form the *memory management* of an allocator.

2. Generic Allocation

The partitions created by R_i are stored in a container C_i, with the allocation predicates $P_1, P_2, \ldots P_n$ serving each container. The allocation predicates form the *memory*

allocation strategy of an allocator.

a. Prioritized Allocation

A prioritized allocation request for $(a_1, a_2, \ldots a_n)$ with the allocation predicates $P_1, P_2, \ldots P_n$ returns a block $= (ra_1, ra_2, \ldots ra_n)$, with $a_1 \overset{P_1}{=} ra_1 \succ a_2 \overset{P_2}{=} ra_2 \succ \ldots \succ a_n \overset{P_n}{=} ra_n$, where $x \overset{P}{=} y$ means that binary predicate P with x and y evaluates to true. If priority is not needed for an specific attribute A_k, it can be expressed using 'any' as P_k allocation predicate.

An allocation request $(a_1, a_2, \ldots a_n)$ executed in recursive partitions denoted by $R_{1,2,\ldots,n} = R_1 \wedge R_2 \wedge \ldots Rn$ and stored in recursive containers $C_1 < C_2 < \ldots C_n < R_n, A_n >> \ldots >$, becomes prioritized by the partition, as discussed in section 3.

b. Non-Prioritized Allocation

An allocation request (a_i) executed in simultaneous partitions denoted by $R^{1,2,\ldots,n} = R_1 \vee R_2 \vee \ldots Rn$ and stored in independent containers $C_1 < R_1, A_1 > \vee C_1 < R_2, A_2 > \ldots \vee C_n < R_n, A_n >$, with the allocation predicate P_i returns a block $=(ra_i)$, with $a_i \overset{P_i}{=} ra_i$ for any $0 \leq i \leq n$.

c. Deallocation

A deallocation request returns a block with attributes $(a_1, a_2, \ldots a_n)$ back into its corresponding partitions. For recursive partitions, deallocation follows the recursive partition chain in returning a block to its partitions. For simultaneous partitions, deallocation returns the blocks to each partition, in any order.

Focus on Management and Allocation Strategies

We implemented this formalism in Allotheque, which is to memory allocation what C++ Standard Template Library is to programming: it allows the users to focus on the allocation strategies rather than the implementation. We share Wilson et al.'s opinion that "... higher-level strategic issues are still more important, but have not given much attention."[77] Allotheque allows experimentation with new dynamic memory allocation strategies in a generic and concise manner. Appendix A describes in more details the design and implementation of Allotheque and appendix B shows the implementation of Kingsley's allocator with Allotheque, which is less than twenty lines of code.

C. Designing TP, Defero, and Medius

We now show how TP, Defero, and Medius use the generic memory allocation framework to implement their internal memory management schemes. Fig. 57 show the implementation of these schemes.

1. TP

We use R_{size} relation to represent TP's size segregation, as described in chapter IV, page 39, and R_{K-bit} relation for the most significant K bits. With these two, relations TP's memory partition can be formally expressed as $R_{TP} = R_{K-bit} \vee R_{size}$. TP has simultaneous partitions formed of two independent partitions, based on address and size. The address partition is depicted in line 1 in fig. 57 as a K-region based on address with lists within and 'any' as allocation predicates for these lists. The size partition is depicted in lines 2-3 in fig. 57, and it has a hash table of lists of lists. The outer lists are lists corresponding to all K-regions which hold blocks of that size.

```
//TP
1. typedef region<K, list<void>,any> tp_address;
2. list<void, tp_address, any> lreg;
3. hash_table<size, lreg, match, hf_segregated>  tp_size;
//Defero
4. typedef rb_tree<kbit<K>, list<void>,match> tree_k;
5. hash_table<size, tree_k, match, hf_segregated>  defero;
// Medius
6. typedef list<size, list<void>,match_segregated> list_size;
7. hash_table<address, list_size, match_last_first, hf_kbit>
                        medius_partition;
```

Fig. 57. TP's, Defero's, and Medius' generic partitions

2. Defero

Defero has a recursive partition that can formally be described as $R_{Defero} = R_{K-bit} \wedge R_{size}$. Fig 57, lines 4-5, shows Defero's memory partition and allocation strategy. A hash table stores the size based partition, which is indexed using a hash function that maps sizes of the same size classes into the same entry. This hash table uses 'match' as allocation predicate, which selects only the hashed entry for searching.

In line 4, each size class is further partitioned based on the R_{K-bit} relation and the resulting partition is stored in a red-black tree that uses 'match' to find the hint's K-region. For hintless allocation this predicate is replaced with 'first', which returns the K-region at the root of this tree. A K-region's elements are stored in a list, whose default is void and its allocation predicate is 'any'.

3. Medius

Medius has a recursive memory partition can be formally described as $R_{Medius} = R_{K-bit} \wedge R_{m-size}$, with R_{m-size} describing Medius' size segregation scheme described on page 39. Line 6 in fig. 57 declares a list of lists, with the nested list using a void attribute and 'any' allocation predicate and the outer list using size as attribute and 'match_segregated' allocation predicate. This predicate matches two blocks if their size match 'segregately', i.e. their computed size classes are the same.

Line 7 declares a hash with lists of lists as buckets, and address' most significant K bits as attribute. A hash function uses the most significant bits as index in the hash table. The allocation predicate 'match_last_first' shows the strategy used in this hash table, checking the hashed index first, followed by the last recently used index and then searches each entry in an increasing order starting with the first index.

4. Other Strategies

We experimented with other allocation policies and strategies. For example, we tried to store the Medius' size classes in a tree for faster searching, but we observed that the list was faster.

We also ran across prioritized allocations that resulted in incorrect allocation. For example, the configuration for Medius' partition that uses 'match' allocation predicate for the hash table and 'match' for the size lists within, attempts to locate the target K-region to find a specific size within it. When a K-region is 'matched', a right size is searched within this class. If the right size does not exist in that K-region, Medius' 'match' allocation predicate determines that there is no more blocks of that size and allocates more memory. The second time it fails again since the newly acquired memory is inserted in a different K-region than target's. At this point Medius

raises an exception and interrupts the application. Using a 'match_first' instead for the hash table fixes the bug by allowing the search to look into all entries instead of only target's.

D. Designing Existing Allocation Strategies

Existing allocation strategies, such as sequential fits, segregated fits, buddy systems or region based, can also be formulated with our generic framework. In this section we illustrate how to design several well known allocation strategies. We already described sequential fits in generic terms in section A. The pictures shown in the remainder of this section use blue[4] to denote memory in the allocator's possession and orange[5] to denote memory in the application's possession. Because most of these mechanisms use splitting and coalescing, we first show how this technique is expressed with our generic framework.

1. Splitting & Coalescing

Splitting and coalescing are common operations in memory allocation used in various allocation policies[49, 52, 77]. *Splitting* takes a large block and splits it into two smaller ones: one of them is returned to the application and the other is inserted back into the allocator's storage. *Coalescing* takes two adjacent blocks and merges them into one contiguous block. The idea of splitting and coalescing is to reduce memory waste and fragmentation. We do not discuss here the benefits and strategies of splitting and coalescing as they have been researched and documented extensively[27, 49, 77].

---

[4] Lighter color for black and white printouts

[5] Darker color for black and white printouts

To express this technique in generic terms, consider the following equivalence relation $R_{adjacency}$ that involves two attributes: size and address. Two blocks are $R_{adjacency}$ equivalent iff they are adjacent in memory. That is if the address of the one plus its size is equal to the address of the other one. $R_{adjacency}$ partitions the memory into contiguous blocks. In the partition created by $R_{adjacency}$, 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 XIV, finds a block large enough and splits into two. The block of the requested size is returned to the application and the remainder inserted back in the partition. A deallocation inserts the block back into its equivalence class and merges it with the equivalence class's block.

Both splitting and coalescing can be represented as allocation and deallocation in the partition created by $R_{adjacency}$, with 'large-enough-to-split' as allocation predicate. They can be implemented in several ways, e.g. using either a bit flip as in binary buddy systems[48] or using boundary tags as described by Knuth in[49].

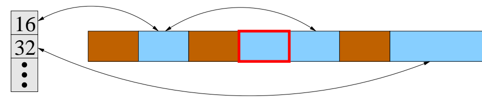
a. Boundary Tags

Boundary tags is a mechanism that delimits blocks in order to facilitate coalescing. Each free block has a header and a footer field, both of which record the size of the block [49, 69]. A status bit is also used to indicate whether the block is free or in use. A deallocated block is coalesced with the previous block, if the latter is free. The application is oblivious to the boundary tags and receives the address after the header.

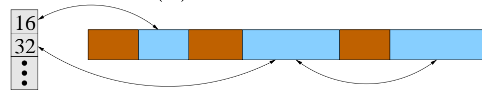
Doug Lea's allocator, dlmalloc, uses a combination of segregated lists and boundary tags within its lists[53, 52]. Fig. 58(a) shows its generic simultaneous partitions, one for splitting and coalescing and the other for size segregation. Lea's allocator uses

```
1. hash_table<size, list<void>,match_next, segregation_dl> seg_lists;
2. split_coalesce<address, void, match, adjacency> boundary_tags;
```

(a) Generic partitions for segregated lists with coalescing



(b) Deallocation



(c) Coalescing

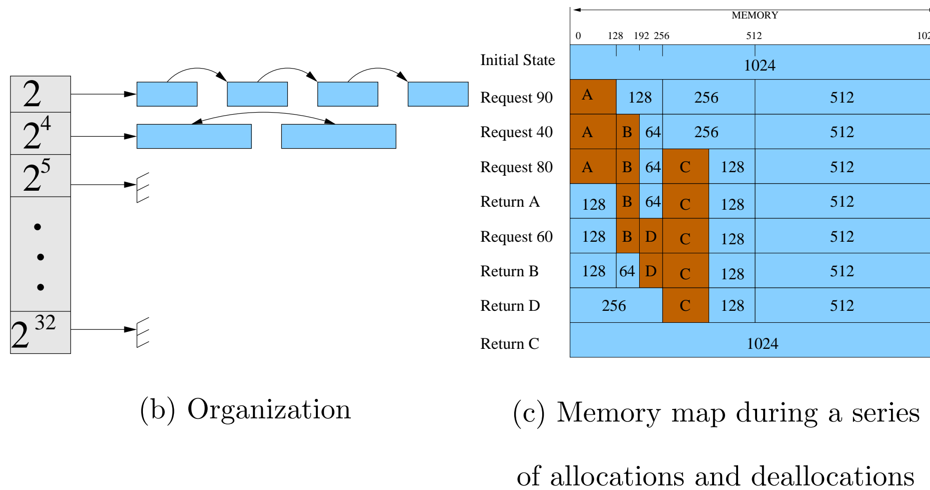
Fig. 58. Coalescing

different strategies for small, medium and large objects, and fig. 58(a) describes only the small object partition. Fig. 58(b) shows the deallocation of a block and fig. 58(c) shows the state after its coalescing.

2. Buddy Systems

Buddy systems are a variant of segregated lists that use splitting and coalesing for every allocation and deallocation[48]. A buddy system conceptually splits the memory into two areas, named buddies, which in turn are split into four smaller areas and so on. Two buddies have the same power of two size. An area can easily find its buddy by flipping its nth bit, where 2^n is the size of the block. Blocks of the same size are stored together in a list designated for that size. All the list sizes are stored in a hash table. Fig. 59(a) shows this organization.

Fig. 59(b) shows the memory map of the binary buddy system during a series of allocation and deallocation requests. Upon allocation, the requested size is rounded



(b) Organization

(c) Memory map during a series of allocations and deallocations

```
1. hash_table<size , list <void >,match_next , hf_poweroftwo>  seg_lists ;
2. split_coalesce<address , void , match , adjacency>  buddies ;
```

(a) Generic partition

Fig. 59. Binary buddies

to the nearest power of two size, which is used as index into the hash table from where the first block in the indexed list is returned. If a block of requested size does not exist, the next largest block is split recursively into two, until requested size is reached. Upon deallocation, if two buddies are free they are coalesced into a larger block. The coalescing process is applied recursively.

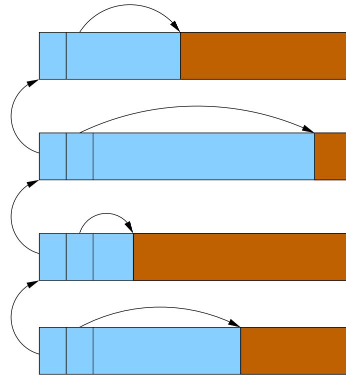
There are various flavors of buddy systems, depending of the way a block is split. Fibonacci buddy system splits the buddies according to the Fibonacci series[32]. Weighted buddy system splits the buddies according to the series: $2, 3, 4, 6, ...2^k, 3 * 2^{k-1}$, while double buddy systems use the series of power of two 2^k interleaved with the series $2^k + 2^{k-1}$[65, 77]. Fig. 59(c) shows the simultaneous partitions of a binary buddy system, while the other buddy systems can be implemented by substituting

their appropriate size segregation.

3. Regions

Region based memory allocation was first introduced by Tofte and Talpin for functional languages, in which all variables within a scope are allocated in a contiguous region using a pointer-bumping technique [73]. Upon exiting the scope, the whole region is deallocated. Each scope has one or more regions associated with it. All regions are stored in a list of regions that mimics the scope nesting. Fig. 60 shows the structure of region based memory management, as well as its generic partition.

(a) Region-based allocator's internal structure



(b) Generic partition

```
1. list<void,region<region_size>,first> stack_region;
```

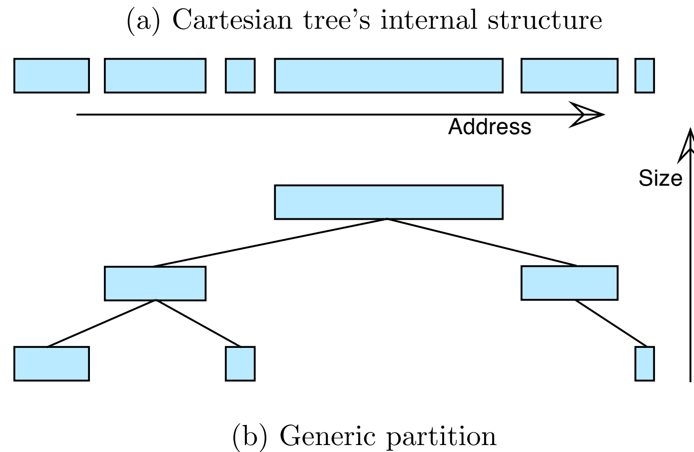
Fig. 60. Regions

4. Cartesian Trees

Cartesian trees store free blocks in a binary tree ordered based on both size and address [70]. A block is smaller than any of the blocks in higher levels, and larger

than any of the blocks in lower levels. Similarly, a block's address is larger than any of the blocks located in its left sub-tree and smaller that any of the blocks located in its right-tree. Fig. 61 shows the structure of such cartesian tree.

(a) Cartesian tree's internal structure



(b) Generic partition

```
1. typedef cartesian_tree<SizeAndAddress> ct_allocator;
2. ct_allocator::allocate<leftmost>(n);
3. ct_allocator::allocate<better>(n);
```

Fig. 61. Cartesian tree

Leftmost-fit and better-fit are two allocation policies used with a cartesian tree. The leftmost-fit policy traverses the tree top-down, until the size of the block is smaller than the requested size, and moving left, selecting the lowest address between two blocks. The better-fit is also a top-down traversal, which selects the best fit block at each level until the size of the block is smaller than requested. Fig. 61, lines 1-3, depict the generic partition of such cartesian tree, together with its two allocation predicates, leftmost and better fit.

Cartesian tree was introduced as a remedy to the best-fit's linear search time. However, the tree can get unbalanced, in which case the complexity is linear in the

number of blocks, as in best-fit's case.

E. Comparison

Table XVI formalizes all the allocators we mentioned in this dissertation in generic terms. The table presents a lateral comparison between the core components of these allocators. It shows that the cartesian tree based allocator is the only one that stores its elements in a container based on two attributes, size and address. Its allocation predicates, leftmost and better, also distinguish themselves from the other allocation predicates.

We can observe, for example, that Vam, PHKmalloc and TP have similar structures, with the difference in the size segregation schemes, and that only TP allocates based on address among the three. This is a structural comparison and there are several details that are not included in this generic description, such as the fact the PHKmalloc stores its pages in additional memory, or that Vam uses a page directory for fixed size regions, or that TP stores the region's list in the upmost address in the region. Nonetheless, this comparison points out the major structural similarities and differences.

A quick scan shows which allocators use coalescing or which schemes use power of two size segregation. Another quick scan shows which allocators have recursive partitions and which allocators have simultaneous ones, or both.

Defero stands out as a locality refinement of segregated trees, while Medius stands out as the allocator that indexes memory in a hash table based on address.

F. Summary

We propose a generic approach to the memory allocation problem that decomposes an allocator into three independent dimensions: attributes, partitions and allocation predicates. These three components describe different strategies and managements in the same language. The generic framework was implemented into a software library, Allotheque, which is to memory allocation what C++ Standard Template Library is to programming: it allows the users to focus on the allocation strategies rather than the implementation.

This generic approach allows memory to be partitioned and allocated based on any contextual attribute. It can also be used as a comparison tool between allocators. We show how existing allocators as well as new allocators are described in this generic approach and then we draw a comparison amongst all allocators.

Allocators	Storage (cont), attribute + relation(ar) and allocation predicate (ap)								
	Level 1			Level 2			Level 3		
	cont	ar	ap	cont	ar	ap	cont	ar	ap
sequential fits	1. list	size	first, best, next worst	-	-	-	-	-	-
	2. split-coalesce	address	match	-	-	-	-	-	-
segregated lists	hash	size-seg	match_next	list	-	any	-	-	-
buddy systems	1. hash	size-pow2	match_next	list	-	any	-	-	-
	2. split-coalesce	address	match	-	-	-	-	-	-
cartesian	cart-tree	size+address	leftmost,better	-	-	-	-	-	-
region-based	list	-	first	region	size	first	-	-	-
Vam	1. hash	size-vam	match_next	list	-	any	list	-	any
	2. region	address	first	list	-	any	-	-	-
PHKmalloc	1. hash	size-pow2	match_next	list	-	any	list	-	any
	2. region	address	-	bitmap	-	any	-	-	-
Defero	hash	size-d	match	tree	kbit	match	list	-	any
Medius	hash	kbit	match_last_first	list	size-m	match	list	-	any
TP	1. hash	size-t	match	list	-	any	list	-	any
	2. region	kbit	match	list	-	any	-	-	-

Table XVI. Generic partitions of various allocation schemes

CHAPTER VII

CONCLUSIONS

Dynamic memory allocators improve an application's performance by optimizing its data layout. Adjustable and flexible techniques are needed to explicitly address data locality. While considering locality as an important goal, an allocator must not abandon the existing constraints of speed and fragmentation, which complicate its design. Memory allocators can use an application's spatial hints to improve its locality. However, a practical approach needs to be automatic, without user intervention. Nevertheless, an allocator with high locality of reference can be as competitive as one using an application's spatial feedback. Hence, users need tools to build new memory allocation strategies that explore the existing and emerging issues.

A. Contributions

The contributions of this dissertation are:

three novel locality improving allocators with hints - we develop the three novel memory allocators that are locality conscious: Defero, which prioritizes size over location, Medius, which prioritizes location over size and TP, which can do both. With a simple yet adjustable locality accuracy, $K \in [0, 32]$, our allocators can exploit locality on all fronts: cache, page and page clustering. Because they manage memory efficiently based on size and address, our allocators improve locality when used both with and without allocation hints. Performance results on seven large, real world applications, show all three allocators improving execution time over state-of-the-art allocators, such as Doug Lea's and FreeBSD's allocators. We further show that in locality improving allocator speed, frag-

mentation and locality circularly compete with each other in a game of rock, paper, scissors: efforts to increase one hurt others.

automatic technique to supply allocation hints from C++ STL - we develop an automatic technique to supply allocation hints from C++ Standard Template Library (STL) directly to their allocators. This process requires no user intervention, but only an application re-compilation.

explicit vs. implicit locality comparative study - we compare our allocators' performance when used with and without STL automatic hints. This study measures the potency of STL automatic hints as well as the efficiency of our allocators. While STL hints increase page accuracy, the costs of exploiting them offset the benefits, which results in hint allocation performing as well as hintless allocation. Our detailed analysis corroborates these findings and shows that the STL audience is knowledgeable about locality issues and uses mostly vectors, which limits the locality enhancing potential for STL containers.

generic memory allocation - a theoretical and generic formalism that allow exploration of unconventional memory allocation strategies. We implement this formalism into a software tool that is to memory allocation what the C++ Standard Template Library is to programming: it allows users to focus on allocation strategies, rather than their implementation. We use it to explore new allocation strategies and compare the most common allocation mechanisms.

The source code is available at the following URL:
http://parasol.tamu.edu/resources/downloads.php

B. Future Work

In this thesis we develop memory allocation techniques that improve data locality, and can collaborate with an application to enhance performance. However, there are still many open research avenues. For example, a specialized allocator can tailor its functionality based on the specific behavior of each container type, as we showed that STL containers behave differently. Other containers that are not STL standard, such as graphs and hash tables, can also supply contextual information to their allocators for performance improvement.

The hint selection process can be also refined. Instead of a fixed spatial strategy, a container can change it hints and inherently its data layout to better fit each computational phase's traversal pattern. A compiler can automate such a process through its compile or run-time analysis and can shape the container's layout as the computation requires.

A memory allocator can use different contextual information to improve performance. For parallel systems, an allocator can use the thread identifier or the shared memory address division to solve problems specific to parallel systems, such as the producer-consumer problem.

We implemented automatic hints for C++ Standard Template Library, yet the scheme is applicable to other languages and tools that use dynamic memory allocation, such as object oriented and functional languages. Their specific contexts provide opportunities for further performance improvement.

As computer hardware design evolves and memory latencies change, the balance between speed, allocation and fragmentation needs to be re-evaluated. Possibly, new constraints may be added to memory allocation. Nevertheless, we believe that each new dynamic memory allocation strategy needs to be flexible and automated, as its integration into today's large applications requires adaptability and can no longer be performed manually.

REFERENCES

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

[2] Aneesh Aggarwal. Software caching vs. prefetching. In *ISMM '02: Proceedings of the 3rd ACM International Symposium on Memory Management*, pages 157–162, Berlin, Germany, 2002.

[3] American Institute of Mathematics. Representations of E_8, www.aimath.org/e8.

[4] 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, 1994.

[5] John Backus. Can programming be liberated from the von Neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, 1978.

[6] 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, Albuquerque, New Mexico, 1993.

[7] 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, Snowbird, Utah, 2001.

[8] 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, Seattle, Washington, 2002.

[9] Shekhar Borkar, Pradeep Dubey, Kevin Kahn, David Kuck, Hans Mulder, Steve Pawlowski, and Justin Rattner. Platform 2015: Intel processor and platform evolution for the next decade, http://www.intel.com/technology/magazine/computing/platform-2015-0305.pdf. *Technology@Intel Magazine*, March, 2005.

[10] Brad Calder, Chandra Krintz, Simmi John, and Todd Austin. Cache-conscious data placement. In *ASPLOS-VIII: Proceedings of the Eighth ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 139–149, San Jose, California, 1998.

[11] Roland Cheng and Chen Ding. Measuring temporal locality variation across program inputs. Technical Report *UR CSD; TR 875*, University of Rochester,Computer Science Department, 2005.

[12] Sigmund Cherem and Radu Rugina. Region analysis and transformation for Java programs. In *ISMM '04: Proceedings of the 4th ACM International Symposium on Memory Management*, pages 85–96, Vancouver, British Columbia, Canada, 2004.

[13] Trishul Chilimbi, Richard Jones, and Benjamin Zorn. Designing a trace format for heap allocation events. In *ISMM '00: Proceedings of the 2nd ACM International Symposium on Memory Management*, pages 35–49, Minneapolis, Minnesota, 2000.

[14] Trishul M. Chilimbi, Bob Davidson, and James R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 13–24, Atlanta, Georgia, 1999.

[15] 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, Atlanta, Georgia, 1999.

[16] Trishul M. Chilimbi and James R. Larus. Using generational garbage collection to implement cache-conscious data placement. In *ISMM '98: Proceedings of the 1st ACM International Symposium on Memory Management*, pages 37–48, Vancouver, British Columbia, Canada, 1998.

[17] Trishul Madhukar Chilimbi. *Cache-conscious Data Structures: Design and Implementation*. PhD thesis, University of Wisconsin-Madison, 1999.

[18] Wei-Ngan Chin, Florin Craciun, Shengchao Qin, and Martin Rinard. Region inference for an object-oriented language. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 243–254, Washington DC, 2004.

[19] Yoo Chung and Soo-Mook Moon. Memory allocation with lazy fits. In *ISMM '00: Proceedings of the 2nd ACM International Symposium on Memory Management*, pages 65–70, Minneapolis, Minnesota, 2000.

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

[21] Morgan Deters and Ron K. Cytron. Automated discovery of scoped memory

regions for real-time Java. In *ISMM '02: Proceedings of the 3rd ACM International Symposium on Memory Management*, pages 25–35, Berlin, Germany, 2002.

[22] David Detlefs, Al Dosser, and Benjamin Zorn. Memory allocation costs in large C and C++ programs. *Softw. Pract. Exper.*, 24(6):527–542, 1994.

[23] Yi Feng and Emery D. Berger. A locality-improving dynamic memory allocator. In *MSP '05: Proceedings of the 2005 ACM Workshop on Memory System Performance*, pages 68–77, Chicago, Illinois, 2005.

[24] 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, Montreal, Quebec, Canada, 1998.

[25] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, and Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In *Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 291–310, Portland, Oregon, 2006.

[26] Dirk Grunwald and Benjamin Zorn. CustoMalloc: efficient synthesized memory allocators. *Softw. Pract. Exper.*, 23(8):851–869, 1993.

[27] 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, Albuquerque, New Mexico, 1993.

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

[29] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, California, USA, third edition, 2002.

[30] John L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, 2006.

[31] John L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Comput. Archit. News*, 35(1):84–89, 2007.

[32] Daniel S. Hirschberg. A class of dynamic memory allocation algorithms. *Commun. ACM*, 16(10):615–618, 1973.

[33] Martin Hirzel. Data layouts for object-oriented programs. *ACM SIGMETRICS Perform. Eval. Rev.*, 35(1):265–276, 2007.

[34] http://www.freebsd.org/. *Documentation of BDS 4.4*.

[35] http://www.wikipedia.org. http://en.wikipedia.org/wiki/Deque.

[36] http://www.wikipedia.org. http://en.wikipedia.org/wiki/Equivalence_classes.

[37] http://www.wikipedia.org. http://en.wikipedia.org/wiki/Set_theory.

[38] 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, Vancouver, British Columbia, Canada, 2004.

[39] Innovative Computing Laboratory,University of Tennessee - Knoxville. Performance Application Programming Interface.

[40] Mark S. Johnstone and Paul R. Wilson. The memory fragmentation problem: solved? In *ISMM '98: Proceedings of the 1st ACM International Symposium on Memory Management*, pages 26–36, Vancouver, British Columbia, Canada, 1998.

[41] J.R. Minkel. Mathematicians Team Up with Supercomputer to Crack 248-Dimensional Object. Scientific American, March, 2007. http://www.sciam.com/article.cfm?id=mathematicians-team-up-wi.

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

[43] Alin Jula and Lawrence Rauchwerger. How to Focus on Memory Allocation Strategies. Technical Report *TR07-003*, Texas A&M University, Department of Computer Science, College Station, 2007.

[44] Alin Jula and Lawrence Rauchwerger. Balancing Speed, Locality and Fragmentation in a Memory Allocator. Technical Report *TR08-002*, Texas A&M University, Department of Computer Science, College Station, 2008.

[45] Sheetal V. Kakkad, Mark S. Johnstone, and Paul R. Wilson. Portable run-time type description for conventional compilers. In *ISMM '98: Proceedings of the 1st ACM International Symposium on Memory Management*, pages 146–153, Vancouver, British Columbia, Canada, 1998.

[46] Poul-Henning Kamp. Malloc(3) revisited. http://phk.freebsd.dk/pubs/malloc.pdf.

[47] Scott F. Kaplan, Lyle A. McGeoch, and Megan F. Cole. Adaptive caching for demand prepaging. In *ISMM '02: Proceedings of the 3rd ACM International Symposium on Memory Management*, pages 114–126, Berlin, Germany, 2002.

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

[49] Donald E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1 (3rd ed.). Addison Wesley Longman Publishing Co., Inc., Redwood City, California, USA, 1997.

[50] Per-Ake Larson and Murali Krishnan. Memory allocation for long-running server applications. In *ISMM '98: Proceedings of the 1st ACM International Symposium on Memory Management*, pages 176–185, Vancouver, British Columbia, Canada, 1998.

[51] 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, Chicago, Illinois, 2005.

[52] Doug Lea. A memory allocator. *The C++ Report*, 1989. http://gee.cs.oswego.edu/dl/html/malloc.html.

[53] Doug Lea. Some storage management techniques for container classes. *The C++ Report*, 1989. ftp://g.oswego.edu/pub/papers/C++Report89.txt.

[54] Tech Libraries. SGI - Standard Template Library - Allocator Design. Silicon Graphics, Inc., 2006. http://www.sgi.com/tech/stl/alloc.html.

[55] 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.

[56] 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, California, USA, 1995.

[57] Gor V. Nishanov and Sibylle Schupp. Garbage collection in generic libraries. In *ISMM '98: Proceedings of the 1st ACM International Symposium on Memory Management*, pages 86–96, Vancouver, British Columbia, Canada, 1998.

[58] Princeton University. Olden Benchmarks. http://www.cs.princeton.edu/˜mcc/olden.html.

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

[60] Feng Qian and Laurie Hendren. An adaptive, region-based allocator for java. In *ISMM '02: Proceedings of the 3rd ACM International Symposium on Memory Management*, pages 127–138, Berlin, Germany, 2002.

[61] Sven G. Robertz. Applying priorities to memory allocation. In *ISMM '02: Proceedings of the 3rd ACM International Symposium on Memory Management*, pages 108–118, Berlin, Germany, 2002.

[62] J. M. Robson. An Estimate of the Store Size Necessary for Dynamic Storage Allocation. *J. ACM*, 18(3):416–423, 1971.

[63] J. M. Robson. Bounds for Some Functions Concerning Dynamic Storage Allocation. *J. ACM*, 21(3):491–499, 1974.

[64] Matthew L. Seidl and Benjamin G. Zorn. Segregating heap objects by reference behavior and lifetime. In *ASPLOS-VIII: Proceedings of the Eighth ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 12–23, San Jose, California, 1998.

[65] Kenneth K. Shen and James L. Peterson. A weighted buddy method for dynamic storage allocation. *Commun. ACM*, 17(10):558–562, 1974.

[66] 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, Seattle, Washington, 2002.

[67] Jeremy G. Siek and Andrew Lumsdaine. C++ concept checking. *Dr. Dobb's J.*, 26(6):64–70, 2001.

[68] Standard Performance Evaluation Corporation - SPEC. CPU2006.

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

[70] 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, Bretton Woods, New Hampshire, 1983.

[71] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[72] Mads Tofte. A brief introduction to regions. In *ISMM '98: Proceedings of the 1st ACM International Symposium on Memory Management*, pages 186–195, Vancouver, British Columbia, Canada, 1998.

[73] Mads Tofte and Jean-Pierre Talpin. Region-based memory management. *Inf. Comput.*, 132(2):109–176, 1997.

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

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

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

[77] 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 Springer-Verlag International Workshop on Memory Management*, pages 1–116, London, UK, 1995.

[78] Michael Wong. C++ benchmarks in SPEC CPU2006. *SIGARCH Comput. Archit. News*, 35(1):77–83, 2007.

[79] Peng Wu, Paul Feautrier, and David Padua. Analyzing java arrays: Combness and synchness analysis. Technical Report CSRD Technical Report No. 1570,

University of Illinois at Urbana-Champaign, 1999.

[80] Wm. A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *SIGARCH Comput. Archit. News*, 23(1):20–24, 1995.

APPENDIX A

ALLOTHEQUE'S DESIGN & IMPLEMENTATION

This appendix provides more details on Allotheque's design and implementation. The library is implemented using C++ templates and all class methods are static. Thus, there is no memory overhead for instantiated objects.

Attributes

Each attribute is designed as a separate class that stores whether the attribute is explicit or implicit, as well as the size required to represent this attribute. An attribute class has two operations: (i) read and (ii) write. Fig. 62 shows the implementation of

```
struct CacheSetAttribute {
  typedef implicit_attribute_tag attribute_tag;
  //! Number of bytes it required
  enum {Attribute_Span = 0};
  //! Read the cache-set of the memory block.
  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) {}  };
```

Fig. 62. Cache set attribute

'cache-set' attribute for a 2 MB L2 cache with 64-byte cache line. The method 'read'

Recursive Partitions

Container's structure	Explicit attribute	Nested container's header	Last element

Simultaneous Partitions

Container 1	Container-2	Container-3	Container-4

Fig. 63. Block's structure for recursive and simultaneous partitions

extracts the cache set index from the address while 'write' is empty because cache set attribute is an implicit attribute and implicit attributes need not be written.

Blocks

Blocks in Recursive Partitions

The block's structure for recursive partitioning is depicted in Fig. 63(top). The first item stored in a block 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 block is the explicit attribute, or is empty is implicit. The third item stored in a block is a pointer to the next container in the partition recursion, if any. And finally, the fourth item a block stores is a pointer to the nested container's last allocated from equivalence class, if the allocation predicate the cached value.

The recursive partition requires a minimum block size of $min_{size} = max\{P_i, 0 \leq i \leq N\}$, where P_i is the size required to store the block in the ith container. Thus, containers that have a high memory overhead, such as hash tables, are not appropriate in a recursive partition, since their overhead is propagated to every block. Except when the hash is the first container in the recursion chain, in which case the overhead

appears only once.

Blocks in Simultaneous Partitions

The block's structure for simultaneous partitioning is depicted in Fig. 63(bottom). Each block carries the internal structure of every partition. As a result, the number of simultaneous partitions is limited by the block size. For example, if a partition requires two pointers, the number of partitions for blocks of size 16 bytes is limited to two.

The minimum block's size to represent N simultaneous partitions is: $min_{size} = \sum_{i=0}^{N} size_i$, where $size_i$ is the size required to store the ith container's internal structure. This makes simultaneous partitions more desirable for larger blocks and less desirable for smaller ones. As for the container memory overhead, this is paid only once for each partition not for every nested equivalence class as in the recursive partition, which makes hash table suitable for storing simultaneous partitions.

Containers

Each partition keeps its blocks in a container that has four template parameters: (i) block's attribute, (ii) allocation predicate, (iii) nested container type, and (iv) offset within a block 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. 64.

The allocation is called recursively on the nested container. Since blocks can belong to nested containers, the NextContOffset offset, line 1, carries the displacement from the beginning of the block where the data structure of the current nested container is stored.

We implemented five containers in our framework: singly linked lists, doubly

```
1. template<class Attr,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)
```

Fig. 64. Container's interface

linked lists, a generic hash table and several hash functions, a red-black tree and the identity container, which ends the recursion. This container collection is not 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 [70].

Recursive Partition is implemented using C++ nested templates. For example, three recursively nested containers, each with a different atrribute, are implemented as: $C_0 < Attribute_0, C_1 < Attribute_1, C_2 < Attribute_2, identity >>>$. 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 block,

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*

Allocation Search and Predicate

The search algorithm for an allocation receives a collection of attributes in the form of a generalized memory request. Fig. 65 shows this generic search algorithm, line 2-8. 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, line 4. This technique allows for the decoupling of memory management from memory allocation, which can be developed independent of each other.

The 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 it either reaches the end 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 recursion reaches the identity container, which simply returns the block found. The allocation predicate is implemented as a separate class. Lines 9-14 in Fig. 65 show the implementation of 'match' allocation predicate.

As for deallocation, each block is returned to its corresponding partitions.

```
1. template<class Attr,class DefaultAP,class NestedAllocator>
                 class container {
...
2. template<class Rqst, class AllocPred=DefaultAP>
3. char* Container<Attr>::allocate(Rqst rqt, char** header) {
4. Attribute *req_attr=reinterpret_cast<Attr*>(rqt);
5. AllocPred ap(req_attr);
6. start= begin(AllocPred::state_tag());
7. while (ap.check(start)) ++start;
8. return NestedAllocator::template allocate<Rqst>(rqt,start);}
//------------------------------------------
9.  struct MatchAllocPred{
10. typedef stateless_tag state_tag; // do not cache last value
11. char* target;
12. static int check(char*  obj)   {
13.     return  obj==target ?  0 : 1 ; }
14.  Match(char* t): target(t) {} };
```

Fig. 65. Generalized allocation algorithm and allocation predicate

APPENDIX B

HOW TO BUILD AN ALLOCATOR WITH ALLOTHEQUE

We now show an example of how to build an allocator using Allotheque. The implementation requires less than 20 lines of code, which is approximately the number of pseudo code lines necessary to describe this allocator.

This scheme was first published by Purdom, Stigler and Cheam in 1970 as a mix between the buddy system and first-fit[59]. 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[77]. It was later developed and integrated in BSD 4.2 by Chris Kingsley[34, 55]. The allocator segregates all blocks whose size fall within powers of 2 in a single list. The allocator rounds the requested size to the nearest power of 2 and allocates the first block 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[40].

Implementation

Fig 66 shows the implementation. The simple segregated storage mechanism is implemented is two lines of code, line 2-3. Line 2 declares a hash table with lists as buckets and with power-of-two as the hash function. The actual hash tabel is declared in line 3. 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 block with

the requested size attribute, line 7. This line 7 contains the recursive search algorithm described in chapter VI.

If there are no blocks with that attribute, line 8, the allocator asks for more memory from the SysAlloc class, which acquires memory from 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 block is not found, even after acquiring more memory from the system, the allocator terminates gracefully with an exception, line 12.

```
1. template<class SysAlloc>class KingsleyAlloc{
        // Partition
2. typedef hash_table<size, list<void>,match_only,pow2> partition;
3. static char* tbl[32];
        // Allocation
4. template<class Request, class AllocPred=First>
5. static char* allocate(Request& attrs){
6. char* r= partition::allocate<Request,AllocPred>(attrs,tbl);
7. if (r==ErrorAttributeMissing) {
8.    size_t s=(size_t) reinterpret_cast<Size>(attrs);
9.   char* nc=SysAlloc::allocate(4096);
10.   for(int i=0;i<4096;i+=s) deallocate(Request(nc+i,s));
11.   r= partition::allocate <Request,AllocPred>(attrs,tbl);}
12.if (!ValidAddress(r)) throw Exception(); else return r;}
        // Deallocation
13.template<class Request>
14.static void deallocate( Request& attrs) {
15.partition::deallocate<Request>(attrs);}};
```

Fig. 66. Kingsley's allocator implementation in Allotheque

APPENDIX C

STL CONTAINERS' GUARANTEE FOR VALID ALLOCATION HINTS FOR TP

In this appendix, we show that TP guarantees its allocation correctness when integrated with our method in STL containers, as described in IV. We first define allocation correctness, then describe TP's correctness guarantee, and finally we show that our method of selecting hints from within STL containers satisfies TP's correctness guarantees.

Memory Allocation Correctness

A memory allocator guarantees its allocation correctness if upon a memory request *allocate(size_t s)*, it returns a valid address of at least size s. By *valid* address we mean an address in the process' virtual address space that can be safely used without compromising the program's correctness.

TP's correctness

The TP's ssumption is that the parameter passed as *hint* to *allocate(size_t s, char* hint)* method needs to point to a block of memory previously allocated by TP and of size s.

1. **Hints point to TP's allocated memory**. If this assumption holds, TP guarantees to return a valid block of the requested size. This assumption is used to locate the hint's K-region's header and is enforced for all values of K. For example, suppose the hint parameter has the value of 0x12345678h, the requested size is 8 bytes and TP uses K=8 as default value of K (TP-8).

According to TP's implementation, this means that block sizes smaller than 120 bytes are stored in 8-regions, block sizes $\in (120, 512]$ in 12-regions, block sizes $\in (512, 1,024]$ in 13-regions, as so forth, as described in chapter IV.

For the requested size of 8 bytes, TP identifies that 8-regions correspond to holding this size and thus assumes that this address belongs to a 8-region in its own management. TP uses bitmasking to compute the header's address of this 8-region as follows: the most significant 24 bits remain unchanged while the last 8 bits are set to FOh, pointing to the last 16 bytes of the 8-region. Thus, TP assumes that the header address of the hint's 8-region is located as 0x123456F0. This 8-region's header stores the header of the list with all available blocks in that region. The blocks in this list have the same size, 8 bytes in our example. The first available block in this list is unlinked and returned to the application.

2. **Hint's K-region is empty**. If the hint's K-region does not have available blocks, the search based on hint is abandoned, and TP returns the first available block from the first 8-region in the list of available 8-regions. Thus, TP guarantees the allocation correctness.

3. **Hint is 0**, TP abandons the search for locality and returns the first available block from the first 8-region in the list of available 8-regions, just like the case when the hint's region has no available blocks. This case also guarantees TP's correctness.

4. **Hints point to memory not allocated by TP**. If a hint does not point to a block of memory previously allocated by TP or points to a block of different size, TP's correctness is no longer guaranteed. There are two cases that cover this scenario, namely, when a hint points to a memory block: a) previously

allocated by TP but of different size than requested, and b) not allocated by TP, with the same or different size. Next, we show what happens in each case.

(a) If the hint points to a memory block previously allocated by TP but of different size than requested. In this case, TP interprets the address 0x123456FO as pointing to the header of a 8-region from which the hint was supposedly allocated from. If the size of the memory block pointed to by the hint and the requested size are in the same size segregation class, e.g. hint block's size is 12 and the requested size is 16, then the allocation luckily proceeds correctly due to the internal size rounding imposed by each size class, and returns a 16-byte block in the hint's 8-region. However, if hey are in different size segregation classes, e.g. hint block's size is 12$\in (8, 16]$ and the requested size is 30$\in (24, 32]$, then the returned block will be of the same size as the block pointed to by the hint instead of the requested size, e.g. 16 bytes instead of the correct 32 bytes.

(b) If the hint points to a memory block not allocated by TP, with the same or different size. In this case, the address 0x123456FO might not be the header of a valid 8-region. TP however assumes that the list's header is stored at 0x123456FO and interprets it as a valid pointer to the first available block in that list. It returns the address found by dereferencing this pointer. This dereferencing operation can lead to a segmentation fault or return a non-valid address.

TP's correctness is guaranteed when used within STL with our integration

We now argue that our integration of TP into STL guarantees TP's correctness. We confirm TP's correctness by showing that all the hints provided to TP by our

integration in STL containers satisfy the correctness conditions imposed by TP: the hint addresses point to memory blocks previously allocated memory by TP and of the same size as the requested parameter. Once we show this property, we conclude that TP's correctness is guaranteed when used within STL with our integration.

Proof

The rationale of the proof is as follows:

1. all STL containers allocate all their elements with the allocator provided in their instantiation (or the default if not specified)

2. all hints point only to STL containers' elements of the requested size

3. It then follows from 1) and 2) that all hints point to memory blocks allocated by an STL container's allocator, which are of the requested size

4. When TP is used as the default allocator, 3) guarantees the TP's correctness assumption described above and thus TP's allocation correctness

Part 1 of the rationale is guaranteed by the implementation of STL. There is no memory used by STL containers that is allocated by a different entity other than their own allocators. We show part 2) next, which allows us to conclude 3) and 4).

Part 2 - all hints point only to STL containers' elements of the requested size. We show this property for each of the four types of STL containers for which we integrated TP to allocate their elements, using the transitive closure principle.

list - all allocations within list occur from the method named *_M_get_node(void* hint)*. This method is invoked by two methods that insert elements in a list: *insert(iterator position, const value_type& x)* and *void _M_insert(iterator position, const value_type& x, char* hint)* .

In the former method, whose semantic is "inserts an element of value x before position", the hint passed to the default allocator is 0 if the list is empty using list's own method *bool empty()*. If the list is not empty, but *position* point to the first element of the list, then the hint passed to the allocator points to the header of the list, which was previously allocated in the constructor by the default allocator and is guaranteed to have the same size as all list's elements. Else, the hint passed to the allocator is the *position*'s previous element address, which is guaranteed to exist and point to a valid element in the list by the STL correctness assumptions (cite C++ STL) and have the same size as the element to be allocated.

The latter method that allocates list's elements, whose semantic is "inserts new element at position given and with value given", passes the parameter hint directly to the allocator. We now analyze all the methods that call this insert method and investigate how they, the callees, provide the hint. There are five callees: list copy constructor *list(const list& x)*, *void push_front(const value_type& x)*, *void push_back(const value_type& x)*, *void _M_insert_dispatch(iterator pos, InputIterator first, InputIterator last, false_type)*, *void _M_fill_insert(iterator pos, size_type n, const value_type& x*. All five of them provide hints using the same hint selection as described above for *insert(iterator position, const value_type& x)*, with parameter *position* taking the values of the method's parameter in the case of *void _M_insert_dispatch* and *_M_fill_insert*, the *begin()* or *end()* for *push_back* and *push_front*, and the address of each of the list's elements in the copy constructor. STL correctness is guaranteed by the correct selection of the parameters passed to its methods. If STL is incorrectly used, then the hints are subsequently not guaranteed to be correct. For example, if

the list container passed to the copy constructor is not valid, i.e. has invalid elements, then the resulting list is not valid either. Piggybacking on STL correctness assumption of its parameters guarantees the validity of parameters in these five callees's, which in turn guarantees that hints point to memory blocks previously allocated by STL containers' allocators and of the requested size in our integration .

red-black tree - similar to list. Hints point either to the parent of the node to be allocated, or 0 if this does not exist.

deque - similar to list. Hints point to either the hash table that holds deque arrays, or to an existing deque array, if they exist, or 0 otherwise.

vector - similar to list. Hints point to the address of the source vector for copy constructor and `operator =`, or 0 otherwise.

Thus, each of the four container types that we modified to supply hints to their allocator guarantees that its hints point to memory blocks previously allocated by its allocator and of the same size. This in turn satisfies part 2) of the rationale described above, and therefore we conclude that TP's correctness is guaranteed when used within STL with our integration.

VITA

Name	Alin Jula
Address	Department of Computer Science
	Texas A&M University
	TAMU 3112
	College Station, TX 77843-3112
email	alin.jula@gmail.com
Education	B.S., Computer Science, "Babes-Bolyai" University, Cluj, Romania, 1997
	M.S., Computer Science, "Babes-Bolyai" University, Cluj, Romania, 1998

The typist for this dissertation was Alin Jula.