Doc No: P0213R0
Date: 2016-01-24
Author: Graham Bleaney (graham@bleaney.ca)

Validation of Memory-Allocation Benchmarks

Abstract

Memory allocation is a fundamental operation that can have a large impact on the run time of a program. This paper performed a series of benchmarks to measure the performance of a selection of custom allocators in different memory usage scenarios. This paper was built on the results reported in P0089R0 “On Quantifying Memory-Allocation Strategies (Revision 1),” but independently recreated the benchmark code to verify the data, provided additional conclusions for each benchmark, and added an additional benchmark.

In each benchmark performed, it was found that at least one of the custom memory allocation strategies provided performance benefits over the default `std::allocator<>` or `new`/`delete` operations. It is recommended that the benchmarks in this report be used as a guide to identify subsystems where custom memory allocation strategies could be used to improve run-time performance.

All of the benchmarking code written for this paper is publically available at <https://github.com/gbleaney/Allocator-Benchmarks>

Contents

Validation of Memory-Allocation Benchmarks 1

1 Introduction 4

2 Disclosure 4

3 Navigating this Paper Quickly 4

4 Glossary 5

5 Allocators Used: Monotonic and Multipool 6

6 Allocation Strategies 7

7 Benchmarking Strategies 9

8 Platform 9

9 Benchmark I: Creating/Destroying Isolated Basic Data Structures 9

Benchmark Overview 9

Benchmark Presentation 12

DS1, vector<int> 12

DS2, vector<string> 14

Further Benchmarks 15

Analysis 15

Conclusions 16

10 Benchmark II: Variation in Locality (Long Running) 16

Benchmark Overview 16

Examining the Effect of Diffusion on the Global Allocator 18

Examining the Effect of Diffusion on a Local Multipool Allocator 20

Comparing the Performance of the Global vs Local Allocator 21

Conclusions 22

11 Benchmark III: Variation in Utilization 23

Benchmark Overview 23

Benchmark Presentation 23

Total Allocated Memory $T = 2^{30}$ 24

Total Allocated Memory $T = 2^{31}$ through 2^{35} 26

Conclusions 26

12 Benchmark IV: Variation in Contention 27

Benchmark Overview 27

Benchmark Presentation 28
Number of Iterations N = $100*2^{15}$, Size of Allocation S = 2^6 28
Further Tests 31
Conclusions 31
13 Benchmark V: Creating and Destroying Data Structures with Varied Locality and Fragmentability 32
Benchmark Overview 32
Benchmark Presentation 33
DS1, vector<int> 33
DS2, vector<string> 35
Further Benchmarks 36
Analysis 36
Conclusions 36
14 Conclusion 37
Appendix 1: Corrected Benchmark I Results from P0089 39
Appendix 2: Elided results from Benchmark I 44
Appendix 3: Absolute Run Times for Benchmark II 49
Appendix 4: Absolute run times for Benchmark III 53
Appendix 5: Elided results for Benchmark III 57
Appendix 6: Absolute Run Times for Benchmark IV 60
Appendix 7: Absolute Run Times for Benchmark IV, with static buffer removed .. 64

1 Introduction

This paper looks at a series of allocator performance benchmarks and attempts to provide guidance about the appropriate allocator for different memory usage scenarios. This paper is built on the results reported in “On Quantifying Memory-Allocation Strategies (Revision 1)” (doc number P0089R0), but independently recreates the benchmark code to verify the data, and provides additional conclusions for each benchmark. P0089R1, a revision of P0089R0, will be published at the same time as this paper. Unless the revision number is relevant, “P0089” will be used to refer generally P0089R0 and its future revisions, with the reader encouraged to find and read the most recent one. Allocators from Bloomberg’s open-source distribution of the BDE library at <https://github.com/bloomberg/bde> were used for the benchmarks.

The benchmarks in P0089 had some unexpected and hard to explain results. To validate or refute these results, the algorithms described in P0089 were re-implemented and the results were compared. It is important to note that while the original benchmarking code is available at <https://github.com/bloomberg/bde-allocator-benchmarks>, the benchmarking code in this paper was recreated from scratch. This was done to address the possibility of bugs in the original code being responsible for the unexpected results. The code used to generate the benchmarks in this paper can be found at <https://github.com/gbleaney/Allocator-Benchmarks>.

2 Disclosure

I, Graham Bleaney, am a former intern of the BDE (“Basic Development Environment”) team at Bloomberg L.P., and was hired as a contractor for the BDE team to produce this paper and the benchmarks presented within. The authors of P0089 are current members of the BDE team.

This paper, the benchmarks in this paper, and the code used to generate the benchmarks, are entirely my own work, with the exception of some of the explanatory boilerplate carried over from P0089R0 to this paper.

3 Navigating this Paper Quickly

This paper analyzes a series of benchmarks, each intended to investigate the behavior of different allocator implementations under particular conditions. Each benchmark section is broken down into the following subsections:

- Benchmark Overview: Summary of the benchmark’s purpose and algorithm
- Data & Analysis: In depth data and explanations
- Conclusions: Final results and recommendations

Readers bypassing the raw data and analysis may need to use the Glossary (Chapter 4) to help make sense of the language used in the conclusions.

The following is a quick summary of each benchmark:

- Benchmark I: Designed to investigate the effects of different allocation strategies on the run time of creating and destroying basic data structures.
- Benchmark II: Designed to show the effects of local allocators on long running programs, where fragmentability (**F**) is high and locality (**L**) is minimal.
- Benchmark III: Designed to examine the effects of memory utilization (**U**) on a system.
- Benchmark IV: Designed to examine the effects of contention (**C**) a multithreaded environment on memory allocation performance.
- Benchmark V: Designed to investigate the run time of different allocation strategies in a more realistic version of Benchmark I, where the fragmentability (**F**) is greater than zero.

4 Glossary

This section provides the user with some useful terminology that will help in understanding the benchmarks in this paper.

Allocation Density (**D**) - A measure of the relative number of allocation instructions (allocate and deallocate) to the total number of instructions executed.

Churn – A measure of the number of allocation and deallocations that together result in no net change to the amount of memory allocated by the system (i.e., deleting four elements from a list then adding four more results in no net change in the total amount of allocated memory, but creates churn).

Diffusion – The distribution of a subsystem's memory throughout the processes memory. Data structures that use only one chunk contiguous memory, such as a `std::vector<int>`, cannot experience diffusion.

Fragmentability (**F**) – A measure of the potential of a subsystem's allocated memory to become diffused throughout physical memory, as a result of the interference of other subsystems' memory allocation. If a subsystem *is* fragmentable (i.e., other subsystems are present in the process and the subsystem allocates more than one chunk of memory), (**F**) is greater than zero.

Global Allocator – Used to generically refer to the system's default allocator, accessed via `new`/`delete` or `std::allocator`. These two methods of allocation are defined as AS1 and AS2 respectively in Chapter 6: "Allocation Strategies"

Local Allocator – An allocator that is scoped to provide memory to a *proper* subset of objects (one or more), rather than the entire process. The monotonic and multipool allocators, outlined in Chapter 5: "Allocators Used: Monotonic and Multipool," are examples of local allocators.

Locality (**L**) – A measure of how physically and temporally close a subsystem's memory is to the current execution state (i.e., when the memory was last accessed and where it is relative to the memory currently being accessed). Approximated as $L = \frac{I}{M*T}$ with the terms defined as:

I – The number of **instructions** executed in the subsystem over the duration of interest

M – The size of the **memory** footprint of the subsystem accessed for the duration of interest

T – The number of context **transitions** out of the subsystem during the duration of interest

Utilization (**U**) - The maximum amount of memory that was actively in use by the system, divided by 'total' amount of memory that has ever been allocated in a system.

Variation (**V**) – The extent to which the size of memory allocated in a system varies over time. An example of the lowest possible variation would be when chunks of only one given size were allocated throughout an entire subsystem's execution.

Winking Out – The process of destroying objects in a data structure *en masse* by releasing the memory they occupy, along with all the memory they manage, via their allocator's `release` method. This is defined behavior according to the standard (see section 3.8 Object lifetime [basic.life]).

5 Allocators Used: Monotonic and Multipool

The allocators used in this paper are the same as those used in P0089: the "monotonic" and "multipool" allocators. An edited version of the explanation from P0089R0 is included here for the reader's convenience.

A *monotonic allocator* supplies memory from a contiguous block, sequentially, until the block is exhausted, and then dynamically allocates new blocks of geometrically increasing size, typically from the global allocator. Returning memory to a monotonic allocator is a no-op: Any returned memory remains unavailable until the monotonic-allocator object itself is destroyed. Bloomberg's `bdlma::BufferedSequentialAllocator` was the implementation that was used for the benchmarks in this paper.

A *multipool allocator* consists of an array of (adaptive) pools, one for each geometrically increasing request size in a range up to some specified maximum. Each time memory is requested, the memory is provided from the most appropriately sized pool. Freed memory is returned to the pool it came from. When the pool has no free memory, the allocator delivers memory from increasingly larger blocks obtained from the backing allocator (by default, the global allocator). This growth may be capped at some (empirically determined) limit, after which allocated blocks are all of the maximum size. Requests that exceed the maximum pool size pass directly through to the backing allocator. Bloomberg's `bdlma::MultipoolAllocator` was the implementation that was used for the benchmarks in this paper.

The combination of a multipool allocator backed by a monotonic allocator forms the third allocator candidate that we consider in this paper.

Both monotonic and multipool allocators are "managed". A *managed allocator* is an allocator that, in addition to its `allocate` and `deallocate` methods, has a `release` method that can be used to summarily return all of the memory it manages to its backing allocator. The `release` method is called implicitly upon destruction of a managed allocator.

For objects placed in memory obtained from a managed-allocator object, and managing no non-memory resources themselves, we can avoid running the objects' destructors. Instead, they can be "winked out" *en masse* by releasing the memory they occupy, along with all the memory they manage, via their allocator's `release` method.

The runtime benefits of bypassing individual destruction of each element in a container can be significant, as deallocating memory can sometimes be costlier than allocating it. Note that this "winking out" technique requires `new`-ing the container object itself into the managed allocator it is to use, so that (1) its destructor is not called, and (2) its footprint is also released when the allocator goes out of scope. Also note that this behavior is fully defined in the current standard, so long as no "winked-out" object is subsequently accessed (see section 3.8 Object lifetime [basic.life]).

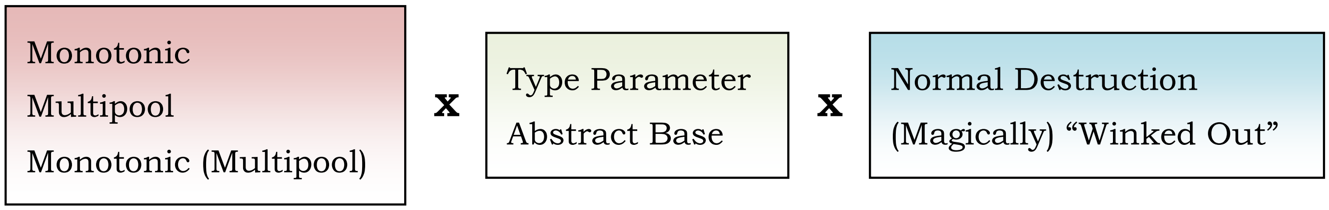
6 Allocation Strategies

The allocation strategies used in this paper are the same as those used in P0089. An edited version of the explanation from P0089R0 is included here for the reader's convenience.

In this paper, up to 14 different allocation strategies are considered for each of the benchmarks to be presented. The first of these strategies will be the default global allocator (`std::allocator`, bound at compile time) which will form the baseline for each successive comparison. The *same* object code will be produced, regardless whether the default allocator is explicitly or implicitly specified, so these are treated as the same case and used interchangeably in the benchmarks.

The second allocation strategy is the *new delete* allocator supplied via an abstract base class. This allocator will demonstrate the additional overhead on compilers that do not elide runtime dispatch. Bloomberg's `bslma::NewDeleteAllocator` is the implementation used for the benchmarks in this paper.

The remaining 12 allocation strategies are comprised of all possible combinations of the following three categories:



The first column represents the allocators themselves. The first entry is a monotonic allocator, the second is a multipool allocator, and the third is a multipool allocator backed by a monotonic allocator. The second column indicates whether the allocator is invasively bound into the type of the container or is (non-invasively) passed via an abstract base class. The third column indicates whether the container was destroyed naturally or, instead, “winked out” by virtue of letting the supplied managed allocator go out of scope.

Label	Allocator type	Allocator binding	Destruction of allocated objects
AS1	Default Global Allocator	Type Parameter	Normal Destruction
AS2	New/Delete Allocator	Abstract Base	Normal Destruction
AS3	Monotonic,	Type Parameter	Normal Destruction
AS4	Monotonic	Type Parameter	“Winked Out”
AS5	Monotonic	Abstract Base	Normal Destruction
AS6	Monotonic	Abstract Base	“Winked Out”
AS7	Multipool	Type Parameter	Normal Destruction
AS8	Multipool	Type Parameter	“Winked Out”
AS9	Multipool	Abstract Base	Normal Destruction
AS10	Multipool	Abstract Base	“Winked Out”
AS11	Monotonic(Multipool)	Type Parameter	Normal Destruction
AS12	Monotonic(Multipool)	Type Parameter	“Winked Out”
AS13	Monotonic(Multipool)	Abstract Base	Normal Destruction
AS14	Monotonic(Multipool)	Abstract Base	“Winked Out”

Table 1: Allocation Strategies

7 Benchmarking Strategies

Benchmarking very quick operations, such as allocating memory for a single data structure, requires the operation to be repeated many times in order to provide a reasonably consistent and noise-free sample. This technique creates a problem because allocating and then immediately deallocating memory for a data structure has no explicitly programmed effect beyond the scope of the loop running the allocation. A good optimizer may be able to spot this “no op” and elide the entire benchmark. The benchmarks described in this paper use the `escape` and `clobber` functions described by Chandler Carruth in his CppCon 2015 talk titled "Tuning C++: Benchmarks, and CPUs, and Compilers! Oh My!" These functions essentially trick the compiler into thinking that the allocated memory is used, by passing a pointer to the data into a piece of empty assembly code that has been labeled as `volatile`. These `escape` and `clobber` methods are used in lieu of various strategies such as writing to elements “using `memset` via a pointer-to-`volatile`,” which were used in P0089R0.

Benchmarks are also sensitive to other processes running on the machine. Wall time measures the span of time between when a program starts and when it finishes, which would include the time that the CPU spent running other processes. CPU time measures the amount of time that a program actually spends executing on the CPU. In order to more accurately determine the amount of time the benchmark code alone spends executing, CPU time was used[*]. CPU time was determined using `std::clock`.

Another important aspect of benchmarking is ensuring that each benchmark is run in a consistent environment. In order to keep the environment as consistent as possible, each entry of each table in this paper was run in its own process. All of the initial boiler plate was set up in the parent process. For each table entry, `fork()` was called, duplicating the environment and creating a child process. The benchmark was run on the child process, the result was outputted, the child process was exited, and then the parent kicked off another child process to produce the next table entry.

8 Platform

The benchmarks in this paper were compiled using Clang 3.6 and run on Amazon Web Services (AWS) r3.2xlarge virtual servers, with High Frequency Intel Xeon E5-2670 v2 (Ivy Bridge) Processors, 8 vCPUs, and 61 GiB of memory.

9 Benchmark I: Creating/Destroying Isolated Basic Data Structures

Benchmark Overview

This benchmark was designed to investigate the effects of different allocation strategies on the run time of creating and destroying basic data structures.

In this experiment, a variety of isolated composite data structures were created, filled with data, and then destroyed. The set of data structures specified by P0089 is used

[*] CPU time could not be used in Benchmark IV, for reasons expanded upon in Section 12

here. This set consists of twelve representative standard-library data structures – the fifth through twelfth (Table 3) being, respectively, `std::vector`s and `std::unordered_set`s of elements containing each of the first four data structure types (Table 2). The full list is shown in the combined Table 2 and Table 3:

DS1	`vector<int>`
DS2	`vector<string>`
DS3	`unordered_set<int>`
DS4	`unordered_set<string>`

Table 2: The four basic data structures used by Benchmark I

DS5	`vector<vector<int>>`
DS6	`vector<vector<string>>`
DS7	`vector<unordered_set<int>>`
DS8	`vector<unordered_set<string>>`
DS9	`unordered_set<vector<int>>`
DS10	`unordered_set<vector<string>>`
DS11	`unordered_set<unordered_set<int>>`
DS12	`unordered_set<unordered_set<string>`

Table 3: The 8 composite data structures used in Benchmark I

The algorithm used in this benchmark can be illustrated as:

1) **Allocate**: Allocate the outer data structure for DS## (where DS## could be DS1-DS12).
2) **Reserve**: Reserve space for `E` elements in the data structure.
3) **Populate**: Fill the allocated data structure with `E` elements, allocated using the same allocator.
4) **Deallocate**: Deallocate the data structure normally or via the “wink out” technique.

5) **Repeat**: Perform steps 1 through 4, a total of 2,560 times.

The values presented in the tables in this chapter are the run time of all five steps.

In the case of DS1 through DS4, the `E` elements that were inserted into the data structure were `int`s or `string`s. For DS5-DS12, these `E` elements were data structures of types DS1-DS4, containing exactly 2^7 = 128 leaf nodes (`int`s or `string`s).

This process of creating and destroying each data structure was repeated many times to allow for meaningful measurements. In order to allow for comparisons across data structures of different sizes, the product of the data structure's size (in terms of leaf elements) and the number of creation and destruction iterations was held constant at an arbitrarily chosen value of 2^{27}. That is, the data structure associated with row 2^8 of any of the first four data structures (DS1-DS4) will be created and destroyed $2^{27-8} = 2^{19}$ times during the benchmark. Note that for data structures DS5-DS12, where the number of leaf elements being constructed per immediate element is increased by a constant factor (e.g., 2^7), a corresponding drop in iterations occurs, thereby keeping the benchmarks roughly comparable in terms of total number of leaf elements created (see below).

The "Reserve" step of the algorithm explicitly pre-sizes the data structure to have the capacity required to store all the elements to be inserted. This pre-sized capacity means that no additional memory allocation will occur to resize the instances of `std::vector` or rehash all the elements of the instances of `std::unordered_set` into new buckets.

Each `string`'s length was chosen randomly over a uniform distribution between 33 and 1000[†], which is deliberately outside the range where the short-string optimization pertains. The container implementations are the native ones for the platform. For container elements that required an allocator, such as `string`, the root container's allocator was explicitly passed to them to prevent the default allocator from being used. The monotonic allocators used in AS3 to AS6 and AS11 to AS14 were supplied with a statically allocated buffer of 2^{30} bytes, just as was done in P0089.

Benchmarks were run on every combination of the 12 data structures above, employing each of the 14 allocation strategies discussed in Chapter 6, for data sizes ranging from 2^6 to 2^{16} nodes in the outer data structure (recall that for DS5-DS12, each "node" is actually a data structure containing 2^7 elements). Results that differed from the results in P0089, and those that display interesting behavior are included in the body of this report. During the development of these benchmarks, it was discovered that some of the columns of data in the Benchmark I section of P0089R0 were transposed. The comparisons in this paper will be made against the corrected data, which is available in "Appendix 1: Corrected Benchmark I Results from P0089".

[†] Note that P0089R0 mistakenly specified 33-100 here; however, the benchmarks actually used 33-1000

Benchmark Presentation

As with P0089, all tables for this benchmark are presented as heat maps in terms of run times in seconds. The first column, 2^6 through 2^{16}, indicates the size of the data structure constructed – e.g., for data size 2^8, the outermost data structure is built up to have $2^8 = 256$ elements before being destroyed.

Note that, in each of the tables below, green indicates substantially shorter run times whereas yellow, orange, and especially red indicate longer run times. Dark red is anchored at the maximum value in the table, and dark green is anchored at the minimum value in the table. When reading the results, be aware that heat maps can be misleading when comparing between data sets with differing spread sizes or with outliers.

DS1, vector<int>

This section presents the results of Benchmark I run using a `vector<int>` (DS1).

Table 4 shows the DS1 results from this paper and Table 5 shows the corrected results from P0089. Some aspects of these results are similar, such as the decrease in run time corresponding to the increase in data size (i.e. moving from top to bottom in the tables). This observation makes sense when considering that all the memory for a `vector` of `int`s can be allocated in one contiguous chunk. As data size increases, the amount of memory being allocated remains the same, but the allocations are being done in larger chunks, resulting in fewer total operations and thus lower run times. Note that this inverse relationship between runtime and data size is an exception rather than the rule; further benchmarks in this chapter show run time increasing or staying the same as the data size increases.

There are two major differences between the results presented in this paper and those in P0089. First, the results presented in P0089 show around 3x worse performance for the global allocator when compared to the results generated for this paper. Secondly, the results presented in P0089 show the non-wink and non-virtual columns (AS3, AS7, and AS11) performing much better relative to the other columns for each local allocator. This observation is in stark contrast to the results presented in this paper, where all four variations for each local allocator perform fairly comparably.

The differences between the results presented in these two papers can likely be attributed to the different platforms used. When the code written for P0089 was run on the platform outlined in Chapter 8 of this paper, the results matched the ones generated here (Table 4).

The main result from this section is that all of the allocation strategies performed comparably, with the exception of the multipool allocator. At lower data sizes, the multipool allocator represented a clear pessimization compared to the others. On the other hand, the monotonic allocation strategies had a slight edge over the others.

Quantitatively, the results in this paper showed that the monotonic allocator, across its various configurations (AS3-6), ran in 81%-100% (average 96%) of the time taken by the global allocator (AS1). Running in less than 100% of the time of AS1 indicates that the monotonic allocator was an optimization relative to the global allocator. The multipool allocation strategies (AS7-10) ran in 100%-275% (average 130%) of the time taken by the global allocator (a sizable pessimization). Finally, the multipool + monotonic allocation strategies (AS11-14) ran in 100%-175% (average 114%) of the time taken by the global allocator (another pessimization). On average, monotonic allocator offered performance improvements over the default global allocator in this test. The other two allocators resulted in an average decrease in performance.

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	0.42	0.44	0.35	0.34	0.34	0.35	0.91	1.13	0.90	1.15	0.68	0.73	0.69	0.73
2^{7}	0.35	0.36	0.31	0.31	0.31	0.32	0.59	0.71	0.59	0.72	0.48	0.51	0.48	0.50
2^{8}	0.32	0.32	0.30	0.29	0.30	0.30	0.44	0.50	0.44	0.49	0.38	0.39	0.38	0.39
2^{9}	0.30	0.30	0.29	0.29	0.29	0.29	0.36	0.39	0.36	0.39	0.33	0.34	0.33	0.34
2^{10}	0.29	0.29	0.28	0.28	0.28	0.28	0.32	0.33	0.32	0.34	0.30	0.31	0.31	0.31
2^{11}	0.28	0.29	0.28	0.28	0.28	0.28	0.30	0.30	0.30	0.31	0.29	0.29	0.29	0.29
2^{12}	0.28	0.28	0.28	0.28	0.28	0.28	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29
2^{13}	0.28	0.28	0.28	0.28	0.28	0.28	0.29	0.29	0.28	0.29	0.28	0.28	0.28	0.28
2^{14}	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
2^{15}	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28
2^{16}	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28	0.28

Table 4: DS1, `vector<int>`, from the benchmarks written for this paper

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.18	1.86	0.27	0.37	0.42	0.44	0.80	1.01	0.90	1.06	0.62	0.69	0.77	0.67
2^{7}	0.92	1.59	0.25	0.39	0.41	0.41	0.51	0.70	0.64	0.70	0.45	0.51	0.60	0.52
2^{8}	0.81	1.00	0.25	0.38	0.38	0.35	0.35	0.59	0.51	0.61	0.31	0.46	0.51	0.47
2^{9}	0.75	0.95	0.22	0.36	0.39	0.35	0.31	0.46	0.45	0.46	0.26	0.43	0.42	0.40
2^{10}	0.74	0.94	0.21	0.34	0.38	0.36	0.24	0.43	0.43	0.42	0.25	0.38	0.40	0.38
2^{11}	0.75	0.94	0.21	0.33	0.36	0.32	0.22	0.39	0.40	0.41	0.24	0.38	0.37	0.38
2^{12}	0.74	0.94	0.21	0.34	0.38	0.36	0.22	0.37	0.39	0.40	0.22	0.37	0.37	0.37
2^{13}	0.76	0.93	0.20	0.32	0.36	0.40	0.21	0.38	0.39	0.37	0.24	0.36	0.37	0.37
2^{14}	0.77	0.93	0.20	0.33	0.39	0.39	0.21	0.38	0.36	0.38	0.20	0.36	0.39	0.37
2^{15}	0.77	0.94	0.20	0.32	0.37	0.37	0.21	0.39	0.36	0.39	0.21	0.36	0.38	0.36
2^{16}	0.78	0.94	0.21	0.36	0.36	0.37	0.21	0.36	0.36	0.39	0.20	0.36	0.37	0.36

Table 5: DS1, `vector<int>`, from P0089

DS2, vector<string>

This section presents the results of Benchmark I run using a `vector<string>` (DS2). These results can be found in Table 6.

Qualitatively, the results presented in this section appear to show similar patterns to the corrected results from P0089 (found in Appendix 1: Corrected Benchmark I Results from P0089); the monotonic allocator performed the best, with the multipool + monotonic allocator coming in as a close second. Both the monotonic and multipool + monotonic allocators were an optimization over the global allocator. In both tests, all of the allocators experienced performance degradations as the data size increased.

The one place where the benchmarks differ is the global allocator. The global allocator performed significantly worse, relative to the other allocators, in P0089 than in this paper. This worse performance makes the difference between the multipool being an optimization over the global allocator in P0089 and a pessimization over the global allocator in this paper. Irrespective of the multipool's performance relative to the global allocator, it is consistently the worst performing of the three local allocator combinations tested in this benchmark.

Quantitatively, the results in this paper showed that the monotonic allocator, across its various configurations (AS3-6), ran in 28%-67% (average 48%) of the time taken by the global allocator (an optimization). The multipool allocator (AS7-10) ran in 84%-255% (average 155%) of the time taken by the global allocator. Finally, the multipool + monotonic allocator (AS11-14) ran in 32%-77% (average 59%) of the time taken by the global allocator.

	← global →		← Monotonic →				← multipool →				← multi + mono →			
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	10.77	11.28	6.75	6.39	6.74	6.37	9.53	9.44	9.55	9.46	8.06	7.75	8.07	7.76
2^{7}	10.63	11.43	6.30	5.95	6.30	5.88	9.44	9.09	9.32	8.98	7.76	7.35	7.75	7.38
2^{8}	13.55	14.00	9.12	8.72	9.12	8.69	28.95	28.00	28.86	28.32	10.65	10.53	10.69	10.60
2^{9}	14.76	15.30	9.76	9.41	9.73	9.39	34.16	33.67	34.25	33.60	11.12	10.74	11.16	10.75
2^{10}	14.99	15.55	9.63	9.28	9.64	9.27	36.84	36.00	36.98	36.17	11.18	10.63	11.21	10.67
2^{11}	15.30	32.13	9.70	9.35	9.70	9.34	38.93	37.93	39.01	37.97	11.23	10.60	11.25	10.63
2^{12}	32.62	33.06	9.75	9.38	9.75	9.39	40.39	39.02	40.43	39.13	11.29	10.65	11.31	10.67
2^{13}	33.13	33.70	9.76	9.40	9.76	9.41	38.22	36.66	38.36	36.84	11.31	10.66	11.34	10.68
2^{14}	33.49	34.05	9.76	9.41	9.77	9.41	36.55	34.99	36.95	35.23	11.32	10.65	11.34	10.68
2^{15}	36.01	36.60	10.51	10.02	10.57	10.11	47.38	45.44	48.04	45.92	16.60	16.54	16.59	16.85
2^{16}	46.45	47.33	20.16	19.92	20.04	20.12	55.82	53.98	56.61	54.78	26.36	26.94	26.61	27.67

Table 6: DS2, `vector<string>`, from the benchmarks written for this paper

	← global	→	←	Monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	68.90	67.30	12.90	12.80	13.30	12.90	18.10	17.80	18.20	17.70	15.50	14.80	15.60	14.80
2^{7}	68.80	68.20	12.80	12.90	13.20	12.90	20.60	20.20	20.60	20.40	15.10	14.30	15.00	14.40
2^{8}	70.80	68.90	13.20	12.80	13.60	12.90	30.80	30.40	30.70	30.30	15.30	14.60	15.40	14.70
2^{9}	73.10	71.20	13.50	13.50	13.90	13.50	38.20	37.60	38.00	37.30	15.90	15.10	15.90	15.10
2^{10}	75.40	74.30	13.60	13.50	14.00	13.70	41.10	40.30	41.60	40.90	16.00	15.10	15.90	15.00
2^{11}	76.90	74.50	13.60	13.50	14.10	13.60	43.90	43.20	43.70	42.60	16.00	15.00	16.00	15.10
2^{12}	76.10	74.80	13.70	13.50	14.00	13.60	41.20	38.80	40.60	39.40	15.90	14.90	15.80	15.00
2^{13}	76.10	74.80	13.60	13.60	14.00	13.60	41.40	39.20	41.30	39.90	15.90	15.00	15.80	14.90
2^{14}	78.30	76.50	13.60	13.60	14.00	13.60	45.80	42.30	44.80	44.00	16.10	15.20	16.20	15.40
2^{15}	90.40	91.00	20.20	20.10	20.50	20.10	62.20	58.70	62.20	58.20	26.00	25.00	26.00	24.90
2^{16}	103.00	103.00	21.50	21.30	21.80	21.30	66.50	59.20	65.10	59.90	27.00	25.30	27.10	25.20

Table 7: DS2, `vector<string>`, from P0089

Further Benchmarks

The rest of the benchmarks reproduced for this chapter show similar performance to the corrected results from P0089, with the exception of the degraded performance of the global allocator.

Throughout the remaining benchmarks, the monotonic (AS3-AS6) and multipool + monotonic (AS11-AS14) allocation strategies consistently offered performance improvements over the global allocator. The monotonic allocation strategies took between 24%-92% of the time taken by the global allocator, with an average of 48.7%. The multipool + monotonic allocation strategies took between 31%-93% of the time taken by the global allocator, with an average of 59.7%. The multipool allocation strategies (AS7-AS10) were sometimes better and sometimes worse than the global allocator, but were on average worse. The multipool allocation strategies took between 43%-292% of the time taken by the global allocator, with an average of 118%.

For those interested, the results of the tests for DS3-DS12 have been included in “Appendix 2: Elided results from Benchmark I”.

Analysis

This benchmark produced a lot of data, which can be overwhelming when examined in its entirety. This section covers some aggregate numbers extracted from the benchmark results.

The first feature of the data examined was the cost of virtual function calls. Over all the combinations of allocators, data structures, and data sizes, the average overhead of accessing an allocator via a virtual function call was 0.78%.

The next feature examined was the “winking out” technique. Over all the combinations of allocators, data structures, and data sizes, “winking out” resulted in an 8.4% reduction in run time.

Finally, the run time of the tests used each allocation strategy was calculated as a percentage of the run time of AS1 for a given row (i.e. if AS1 took 10s and AS2 took 11s, the percentage for AS2 would be 110%). These ratios were then averaged across every data structure and are presented in Table 8. Clearly, the monotonic (AS3-AS6) and monotonic + multipool (AS11-AS14) allocation strategies caused an overall performance improvement relative to the global allocator. The multipool (AS7-AS10) allocation strategies caused an overall performance degradation.

global	←	Monotonic		→	←	multipool		→	←	multi + mono		→
virtual			← virtual →				← virtual →				← virtual →	
		(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
105.06	54.86	50.23	54.90	50.52	125.73	118.97	125.69	118.97	66.69	61.72	66.60	61.66

Table 8: Average run times of allocation strategies as a percentage of AS1

Conclusions

Overall, the monotonic allocator provided the largest performance improvement in this benchmark, when given a static buffer. Given the demonstrated and theoretical properties of the monotonic allocator, it would be advisable to use a monotonic allocator in situations similar to this benchmark, where large amounts of memory are being allocated, used, and then deallocated. Note that this recommendation does not hold in situations dissimilar from this benchmark, such as in Benchmark IV where there is high churn.

A second conclusion that can be drawn is that the “winking out” technique provides a sizable runtime benefit (8.4% reduction in run time) and should be considered when possible. Finally, accessing an allocator through a virtual function call has a small, but measurable, runtime performance overhead (0.78% increase in run time in this benchmark). This slight overhead can likely be mitigated some optimizers. Whether or not the convenience is worth the overhead of a virtual function call will vary from use case to use case, and platform to platform.

10 Benchmark II: Variation in Locality (Long Running)

Benchmark Overview

This benchmark was designed to show the effects of local allocators on long running programs, where subsystems have a high potential for fragmentation (**F**) and temporal or physical locality (**L**) is low.

This benchmark emulates a long running system through three major steps:

1) **Creation**: A collection of subsystems, represented as `std::vector<std::list<int>>`, is created and populated

2) **Shuffling**: The contents of the subsystems are swapped around to emulate the diffusion of each subsystems memory in a long running program

3) **Usage**: The values in the subsystems are accessed and modified. This step is timed to measure the effect of the shuffling step

During the creation step, an `std::vector` is instantiated and filled with `k` entries of `std::list<int>`. Each list is filled with `S` `int`s, of increasing values. The total system size is characterized by `G = k * S`. The `vector` of `list`s is illustrated in Figure 1.

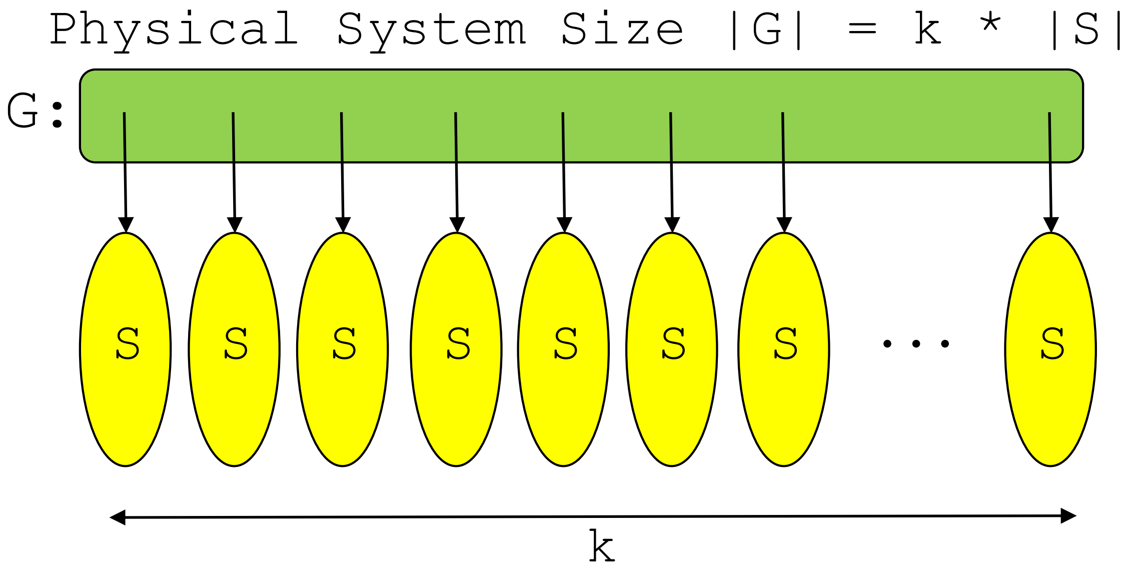


Figure 1: Diagram of Benchmark II system configuration

During the shuffle step, the data in the `k` lists is shuffled by visiting each list in turn, popping from the front, and pushing onto the back of a randomly chosen list. This process is repeated `S` times, so that every element has been popped and pushed at least once. This entire shuffling process of popping and pushing `k * S` elements is then repeated 5 times. Note that during the additional shuffles, some lists may be encountered that have no elements left in them. In these cases, the empty list is skipped.

During the usage step, each of the lists is then accessed, according to an access factor, `af`, and a repeat factor, `rf`. The access factor (`af`) determines how many times each list is traversed before moving on to the next one. For example, with an `af` of 2, the first list would be traversed from beginning to end twice, and then the process would move on to the next list. Every time an element is touched during iteration, its value is incremented (in part ensure that the access is not elided during the optimization phase).

The repeat factor (`rf`) determines how many times the entire vector of lists is traversed. For example, with a `rf` of 2, the test would travel through the vector, accessing each list the number of times specified by the `af`, and then it would travel through the vector a second time and access each list again. The product of `af` and `rf` is held constant at 2560, meaning a multiplicative increase in `af` is matched by a corresponding decrease in rf.

To measure the effect the shuffling step had on the usage step, two tests were run for each table presented in this section. The first test omitted the shuffling step and was timed, to get a baseline for how long the usage step took, without any induced diffusion in the subsystems. Note that some diffusion may have occurred naturally due to the way the system allocated memory. The second test executed the shuffling step as specified above, and the usage step was then timed. Table 9 and Table 10 in this chapter depict the *ratio* of the shuffled usage to unshuffled usage. The absolute run times for each table, with and without the shuffling step, have been included in Appendix 3: "Absolute Run Times for Benchmark II".

Examining the Effect of Diffusion on the Global Allocator

Table 9 depicts the ratio of the time the usage step took when the shuffling step *had* been executed, to the time the usage step took when the shuffling step *had not* been executed. The lists in this table all used the global allocator (AS1). The higher (and redder) a number is, the worse the system performance was post-shuffle.

	Access Factor (`af`)									
List Length (`S`)	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Number of Lists (`k`)
2^{21}	1.01	1.00	0.99	1.00	1.01	0.99	1.01	1.01	1.00	2^0
2^{20}	8.29	8.56	8.33	8.26	8.38	8.44	8.18	8.33	8.39	2^1
2^{19}	19.52	20.73	19.92	19.19	21.86	19.04	20.04	18.36	17.28	2^2
2^{18}	10.75	10.63	11.58	11.63	12.35	13.56	15.35	17.11	17.77	2^3
2^{17}	8.74	8.55	8.94	9.69	10.29	11.76	12.85	15.89	18.09	2^4
2^{16}	6.39	6.58	6.74	6.96	7.66	8.80	10.95	14.23	18.18	2^5
2^{15}	6.61	6.65	6.76	6.91	7.55	8.36	10.31	13.52	17.95	2^6
2^{14}	6.65	6.68	6.78	6.90	7.52	8.61	10.15	13.38	18.13	2^7
2^{13}	7.87	7.95	7.90	8.09	8.87	9.73	11.37	14.22	18.14	2^8
2^{12}	6.48	6.64	6.72	6.87	7.73	8.38	10.84	13.68	18.22	2^9
2^{11}	4.06	4.14	4.82	5.11	5.58	6.76	9.20	13.22	18.15	2^{10}
2^{10}	4.54	4.84	4.99	5.27	6.31	7.66	10.31	13.77	18.29	2^{11}
2^9	3.34	3.85	3.94	4.38	5.27	6.83	9.75	13.48	18.54	2^{12}
2^8	2.07	2.58	2.44	3.05	3.81	5.81	8.99	13.45	18.24	2^{13}
2^7	1.86	1.98	2.15	2.88	3.79	5.50	8.66	13.41	18.02	2^{14}
2^6	1.47	1.67	1.98	2.50	3.47	5.45	8.60	13.10	18.01	2^{15}
2^5	1.42	1.59	1.93	2.75	4.21	6.51	10.76	15.76	16.32	2^{16}
2^4	1.35	1.61	2.11	3.01	4.50	7.60	12.09	16.60	17.96	2^{17}
2^3	1.40	1.73	2.26	3.35	5.39	8.38	13.62	16.44	17.87	2^{18}
2^2	1.33	1.74	2.19	3.35	5.35	9.16	14.33	16.59	16.02	2^{19}
2^1	1.40	1.69	2.25	3.24	4.86	8.75	12.31	13.16	10.66	2^{20}
2^0	1.26	1.51	1.98	2.86	4.22	6.45	8.41	8.17	6.87	2^{21}

Table 9: Problem size `G` = 2^{21} with global allocator, ratio of test with shuffle to test without

Qualitatively, the results produced in Table 9 of this report show similar patterns to the equivalent results in Table 16 of P0089R0. The one major exception to this is that the data presented in POO89 showed an increase in relative run time in the `af=`2^3 through `af=`2^1 entries of the final row. Conversely, in this benchmark, these cells show a decrease in relative run time, when looking through the table top to bottom.

The first row of Table 9 is essentially all ones, which means that shuffling the lists had no effect on the run time. This observation makes sense – especially when one considers that the first row corresponds to a test where there is only one list. The "shuffling" in first row of the table consists of elements being taken off the front of the list, and pushed directly on to the back of the list. Thus, the "shuffled" list consists of a series of nodes that were allocated sequentially, which is essentially the same as an unshuffled list.

Moving down Table 9, the second and third rows show worsening post-shuffling performance, relative to the performance of the test without shuffling. This degradation makes sense because the system has been broken up into a few lists, each of which are still quite long. The hypothesis is that, as the test iterates over these lists, the memory being accessed is likely diffused throughout the process's memory, thus requiring more time to retrieve and increment.

Beyond the third row and moving down Table 9, the performance of tests with higher `af` (left side of the table) improves relative to the baseline, however, the performance of the lower `af` tests (right side of the table) do not. Recall that a lower `af` means that tests further left in the table iterate over each list more times before moving on to the next list in the vector. This observation may explain the improved performance on the left side of the table: Once lists are small enough, more of the list would be able to fit into various system caches, allowing subsequent iterations over the list to benefit from the caching. On the right side of the table, even if the whole list could fit into the cache, each list is accessed fewer times before moving on, resulting in less of a benefit. In the most extreme case, each list in the tests for the right-most column of the table is accessed exactly once before moving on.

On the right side of Table 9, in the rows corresponding to list lengths 2^5 through 2^2, performance once again degrades relative to the unshuffled baseline. It may be that the benefits of caching are felt less when the lists are short, however, this is a tenuous explanation.

The final row of Table 9 shows the best relative performance for each column, with the exception of the first row. The effect of caching appears to still play a part, since the tests with higher access factors (left side) performed better than the tests with lower access factors (right side). The gap between the lower `af` and higher `af` columns appears to have narrowed in the final row. Recall that in this final row, each list contains (on average) only one element, which means the test is not hopping around the process's memory to retrieve multiple elements in the list. The relative performance gain from not having to jump around the process's memory retrieving shuffled elements would apply across the board. It was suspected that the performance gain, relative to entries higher in the columns, would be more obvious on the right side of the table where it is assumed caching would be playing a smaller part. This hypothesis is consistent with the results seen in the final row of the table.

Examining the Effect of Diffusion on a Local Multipool Allocator

Table 10 shows the results of a test identical to the one shown in Table 9, with the exception that a local allocator is used for each list, rather than the global allocator (shared by all the lists). The local allocator used was a multipool allocator. The monotonic allocator was not used in any form, because it never gives up any memory. The behavior of not giving up any memory is particularly ill suited to this benchmark, because the shuffle process would result in a large amount of allocated memory that would then be no longer in use.

	Access Factor (`af`)									
List Length (`S`)	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Number of Lists (`k`)
2^{21}	1.00	1.01	1.00	1.00	1.00	1.00	1.00	0.99	1.01	2^0
2^{20}	1.59	1.59	1.57	1.57	1.58	1.58	1.59	1.58	1.60	2^1
2^{19}	1.52	1.61	1.73	1.71	1.61	1.72	1.77	1.68	1.78	2^2
2^{18}	1.57	1.58	1.59	1.59	1.61	1.62	1.72	1.79	1.86	2^3
2^{17}	1.48	1.48	1.49	1.50	1.52	1.56	1.62	1.75	1.92	2^4
2^{16}	1.48	1.49	1.49	1.50	1.53	1.56	1.65	1.78	1.97	2^5
2^{15}	1.49	1.50	1.50	1.51	1.54	1.59	1.66	1.81	2.04	2^6
2^{14}	1.49	1.50	1.50	1.51	1.54	1.59	1.69	1.84	2.10	2^7
2^{13}	1.50	1.50	1.51	1.53	1.56	1.62	1.73	1.92	2.18	2^8
2^{12}	1.65	1.66	1.67	1.70	1.73	1.82	1.96	2.15	2.37	2^9
2^{11}	1.63	1.64	1.66	1.69	1.76	1.88	2.09	2.38	2.73	2^{10}
2^{10}	1.57	1.58	1.61	1.66	1.75	1.89	2.18	2.52	2.92	2^{11}
2^9	1.07	1.09	1.15	1.25	1.46	1.75	2.21	2.69	3.15	2^{12}
2^8	1.02	1.06	1.12	1.24	1.43	1.79	2.28	2.80	3.21	2^{13}
2^7	1.04	1.08	1.15	1.28	1.52	1.85	2.28	2.77	3.12	2^{14}
2^6	1.03	1.08	1.14	1.30	1.49	1.81	2.18	2.55	2.80	2^{15}
2^5	1.18	1.23	1.31	1.47	1.68	1.98	2.19	2.44	2.33	2^{16}
2^4	1.14	1.18	1.30	1.41	1.59	1.88	1.96	1.90	2.01	2^{17}
2^3	1.19	1.25	1.36	1.52	1.67	1.95	2.15	2.12	2.28	2^{18}
2^2	1.20	1.30	1.49	1.80	2.10	2.21	2.09	2.60	1.91	2^{19}
2^1	1.25	1.36	1.52	1.81	1.98	2.02	2.08	2.14	1.23	2^{20}
2^0	1.16	1.28	1.43	1.67	1.91	1.85	1.73	1.64	1.06	2^{21}

Table 10: Problem size `G = ` 2^{21} with multipool, ratio of test with shuffle to test without shuffle

Table 10 shows behavior similar to that seen in Table 9, with similar likely explanations. One particular aspect of this table that does stand out is the bottom right corner. This improvement of the corner, relative to entries to the left (higher `af`) exists in Table 9 as well, the heat map just highlights it less due to a wider range of values. This corner case was unexpected at first, and was investigated further. Figure

2 shows the raw run times that were used to produce the ratio shown in the rightmost column of Table 10. As can be seen in the figure, the run time of the test that included the shuffling plateaued around the `S=2^2` point, while the run time for the unshuffled test continued to increase. The unexpected performance increase in the bottom right corner of Table 10 becomes a lot more believable once it is clear that this is a *relative* performance increase, rather than a drop in absolute run times.

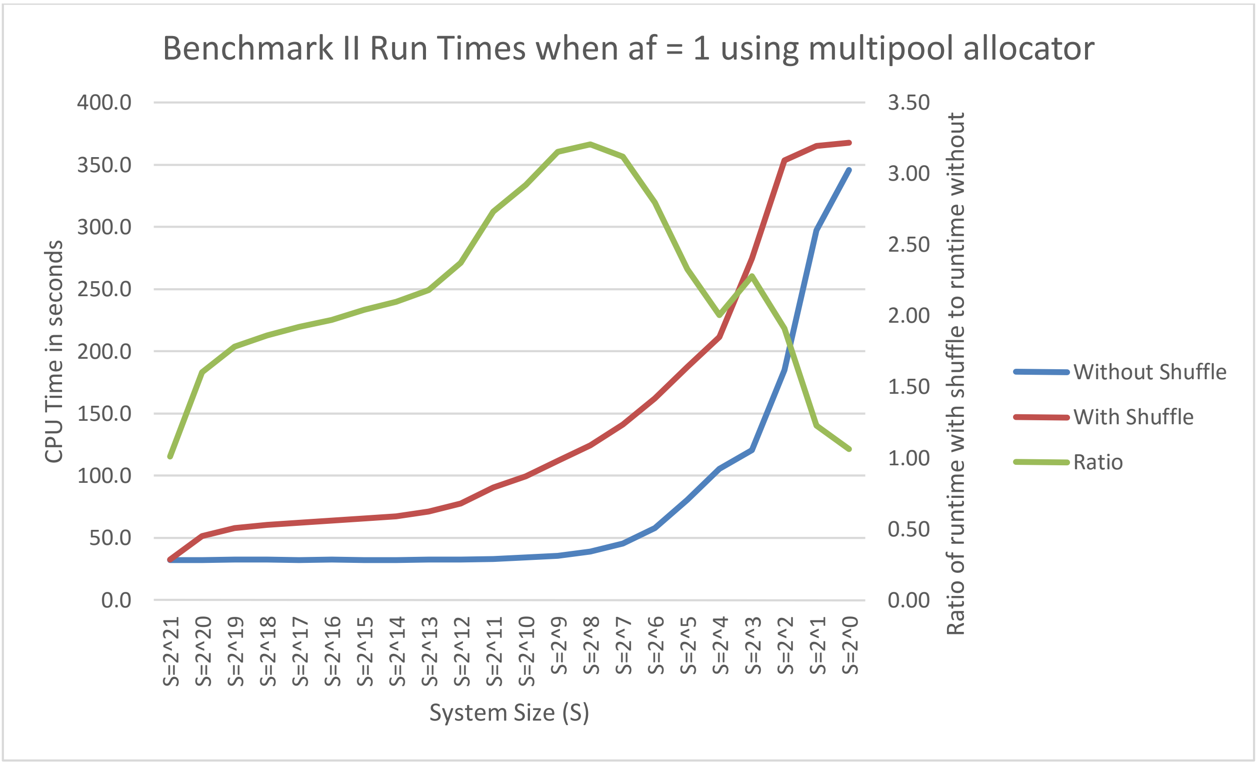


Figure 2: Comparison of run times with and without shuffle for the right column of Table 10

Comparing the Performance of the Global vs Local Allocator

A quick comparison between Table 9 and Table 10 clearly shows that systems using the multipool allocator experience a significantly smaller performance degradation when shuffling has occurred, which suggests that the allocator has helped reduce diffusion within the fragmentable (**F**) lists. One thing that Table 9 and Table 10 do not take into account is that there may be a performance overhead for using a local allocator in the first place. To examine this possibility, Table 11 shows the ratio of the absolute run times of the test with allocators to the test without allocators (Table 46 / Table 44, from Appendix 3: Absolute Run Times for Benchmark II), after each respective system has undergone the shuffling step.

	Access Factor (af)									
List Length (S)	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Number of Lists (k)
2^{21}	1.52	1.53	1.53	1.53	1.52	1.53	1.51	1.52	1.53	2^0
2^{20}	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29	0.29	2^1
2^{19}	0.14	0.14	0.15	0.14	0.14	0.14	0.15	0.15	0.15	2^2
2^{18}	0.22	0.21	0.20	0.20	0.20	0.18	0.16	0.16	0.16	2^3
2^{17}	0.24	0.25	0.24	0.22	0.21	0.19	0.18	0.16	0.16	2^4
2^{16}	0.34	0.33	0.32	0.31	0.29	0.26	0.22	0.18	0.16	2^5
2^{15}	0.32	0.32	0.32	0.31	0.29	0.27	0.23	0.20	0.17	2^6
2^{14}	0.32	0.32	0.32	0.31	0.29	0.26	0.24	0.20	0.17	2^7
2^{13}	0.32	0.32	0.32	0.31	0.29	0.27	0.24	0.21	0.18	2^8
2^{12}	0.32	0.31	0.31	0.31	0.28	0.28	0.24	0.22	0.20	2^9
2^{11}	0.50	0.50	0.43	0.42	0.40	0.36	0.31	0.26	0.23	2^{10}
2^{10}	0.53	0.50	0.49	0.48	0.42	0.38	0.32	0.28	0.25	2^{11}
2^9	0.32	0.29	0.30	0.30	0.30	0.30	0.29	0.29	0.28	2^{12}
2^8	0.51	0.43	0.48	0.44	0.42	0.37	0.34	0.32	0.31	2^{13}
2^7	0.57	0.56	0.56	0.48	0.46	0.43	0.39	0.37	0.36	2^{14}
2^6	0.72	0.67	0.62	0.58	0.52	0.47	0.43	0.43	0.41	2^{15}
2^5	0.85	0.81	0.74	0.63	0.56	0.52	0.49	0.48	0.48	2^{16}
2^4	0.88	0.80	0.72	0.62	0.60	0.56	0.53	0.52	0.54	2^{17}
2^3	0.89	0.81	0.76	0.68	0.63	0.65	0.64	0.64	0.68	2^{18}
2^2	0.93	0.84	0.86	0.82	0.80	0.82	0.82	0.85	0.90	2^{19}
2^1	0.97	0.98	0.96	0.98	1.03	1.02	1.04	1.07	1.18	2^{20}
2^0	1.05	1.09	1.16	1.22	1.31	1.29	1.21	1.21	1.34	2^{21}

Table 11: Problem size G = 2^{21}, ratio of multipool to global allocator runtimes after shuffle

As can be seen in Table 11, a local multipool allocator provides a performance benefit over the global memory allocator, with two exceptions. The first exception occurs when the system consists of only one large data structure. In this case, the system is not fragmentable (**F**) because no other subsystems exist to allow the subsystem's memory to diffuse. Without being able to help improve fragmentability (**F**), it appears that having a multipool for every subsystem adds an extra overhead without any chance to provide a benefit. The other exception is in systems consisting of data structures containing only one element. In this case, the fragmentability (**F**) is minimal, since each subsystem (`list`) allocates only one element. Thus, each local multipool allocator has no chance to help prevent diffusion. Again, this likely means that the multipool is adding an overhead, but without any countervailing benefit.

Conclusions

In this chapter, the local multipool allocator offered a large performance benefit over the global allocator, which suggests that local allocators can help improve performance in subsystems with higher fragmentability (**F**). One important caveat is that the multipool allocator does appear to incur a performance overhead that can result in a pessimization when diffusion is not occurring. For this reason, one should consider whether diffusion is actually affecting the performance of a system, before employing a local multipool allocator.

The benchmarks also showed that more rapid context switches between subsystems caused worse performance (right sides of Table 9 and Table 10, which suggests that the loss of temporal locality (**L**) does indeed degrade the performance of a system.

11 Benchmark III: Variation in Utilization

Benchmark Overview

This benchmark was designed to examine the effects of memory utilization on a system. P0089R0 defines a term, "Utilization" (**U**) as "the maximum fraction of the 'total' amount of allocated memory 'actively' in use at any one time."

The benchmark has three parameters: chunk size, `S`, amount of active memory, `A`, and the total amount of memory allocated, `T`, all of which are measured in bytes. The algorithm was as follows:

1) **Initial Allocation**: Chunks of size `S` are allocated until the desired amount of active memory, `A`, has been achieved
2) **Churn**: A chunk is deallocated, and then a new one is immediately reallocated
3) **Repeat**: The Churn step is repeated until the total amount of memory allocated by the system reaches `T`

All three steps were timed for this benchmark. Note that the first byte of each allocated chunk was incremented to deliberately access it.

Allocation strategies AS4, AS6, AS8, AS10, AS12, and AS14 were not considered because the "winking out" technique circumvents the destruction of the active memory at only the end of the test. In all the tests performed in this benchmark, the total amount of memory deallocated *before* the data structure could be winked out, `T – A`, is orders of magnitudes larger than the amount of memory that would be winked out. Hence, even if the entire run time of the benchmark were a result of the deallocation operations, saving the overhead of `A` deallocations out of a total of `T` would be barely noticeable, if that.

Benchmark Presentation

The results presented for allocation strategies AS2, AS3, AS5, AS7, AS9, AS11, and AS13 in this chapter are displayed as percentages of the run time for AS1. For example, if AS2 took 13s to run and AS1 took 10s to run, the entry for AS2 would be presented as 130%. The absolute run times can be found in "Appendix 4: Absolute run times for Benchmark III". All of the tables are colored as heat maps, with the midpoint (yellow) fixed at 100%. Thus, if a result is green it represents an improvement over AS1, if it is red, it represents a longer run time than AS1, and if it is yellow it took about the same time to run as AS1.

Total Allocated Memory `T` *= 2^{30}*

This section shows the relative run times of AS2, AS3, AS5, AS7, AS9, AS11, and AS13 to AS1 for a system having a total of `T`=2^{30} bytes allocated. The other parameters `A` and `S` were varied as show in Table 12.

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{30}	2^{15}	2^{10}	0.0521s	105	434	435	44	46	46	46
2^{30}	2^{16}	2^{10}	0.0520s	106	435	435	46	46	46	46
2^{30}	2^{17}	2^{10}	0.0515s	105	438	439	46	46	46	47
2^{30}	2^{18}	2^{10}	0.0515s	105	439	439	47	47	46	47
2^{30}	2^{19}	2^{10}	0.0516s	105	438	439	47	47	47	47
2^{30}	2^{20}	2^{10}	0.0519s	105	435	436	47	47	47	47
2^{30}	2^{20}	2^{11}	0.0264s	104	822	824	47	47	47	47
2^{30}	2^{20}	2^{12}	0.0129s	104	1657	1656	54	53	49	49
2^{30}	2^{20}	2^{13}	0.0065s	103	3245	3247	118	118	3281	3279
2^{30}	2^{20}	2^{14}	0.0034s	103	6113	6115	117	117	6143	6143
2^{30}	2^{20}	2^{15}	0.0018s	104	11901	11901	114	116	11910	11912

Table 12: Total Allocated Memory `T` = 2^{30}

First, it can be observed that increasing the amount of active memory, `A`, had no effect on any of the allocators (for both relative and absolute run times). This makes sense, given that `A` started at 2^{15} bytes (32KB) maxed out at 2^{20} bytes (1MB). The system had more than enough memory to support this 32KB – 1MB range of active memory.

The next observation is that the monotonic allocator (AS3 and AS5) performed incredibly poorly (4x-119x slower than AS1). This poor performance is to be expected, since the monotonic allocator can never reclaim any of the memory it has handed out. Hence, the tests using a monotonic allocator would have had a full 2^{30} bytes (1GB) in use by the end of the test.

A third observation is that, until the last few rows, the multipool allocator performs incredibly well, irrespective of its backing (AS7, AS9, AS11, and AS13). This observation also makes sense, given that this test has a certain affinity with the multipool allocator. Once the multipool allocates the initial `A` bytes from whichever backing allocator, the subsequent series of deallocation and reallocations will simply be popping one chunk of memory at a time on to and then off of the multipool's free list. This popping on and off of the free list should be expected to be fast. Note that, depending on the implementation, the remainder of the current chunk of pooled nodes may be consumed before the free list is examined. After the remaining chunk has been consumed, however, the allocator would still reach the steady-state behavior described above.

One final observation is that the multipool allocator appears to take on the characteristics of its backing allocator after the 2^{12} chunk size. This behavior is consistent with the statement made by P0089 indicating that the backing pool is set up to handle chunks up to only 2^{12} bytes; any request exceeding this threshold size would be passed on to the backing allocator, giving the behavior seen in this test.

Qualitatively, the results presented in P0089 (copied in Table 13) look similar to the results presented in Table 12 and the other table in this chapter. There were, however, counter-intuitive results presented in P0089. All of the tables presented in Chapter 9, “Benchmark III: Variation in Utilization,” of P0089R0 had some data points where the global allocator performed better when accessed through a virtual function call than when accessed directly. These results bear further investigation and should probably not be taken at face value. While some optimizers may see through the virtual function call and elide it, there does not seem to be a scenario where this should be an *improvement* over a direct call. Additionally, if the improvement was a few percent, this could be attributed to noise. In some cases, however, the virtual function call ran in as little as 54% of the time the direct call did. These counter-intuitive results did not manifest in the benchmarks run for this paper. Instead, a consistent 3-5% overhead was seen for the virtual function call to the global allocator (AS2).

Another counter-intuitive result in Table 13 is that, in the final row of the multipool column (AS7), the performance gets significantly better (110% of AS1 to 60% of AS1) relative to the row above. In the column representing the multipool accessed through a virtual function (AS9), the exact opposite happens (58% of AS1 in the row above compared to 111% of AS1 in the row below). The only difference between the AS7 and AS9 columns should be the virtual function call, so this sudden change in the relative performance of AS7 vs AS9 does not make sense. This may be an issue with transposed values, or perhaps a deeper issue with the Benchmark III in P0089. Regardless, this unexpected change does not occur in the results presented in this paper.

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{31}	2^{15}	2^{10}	0.063s	103	440	435	46	43	46	47
2^{31}	2^{16}	2^{10}	0.069s	102	401	395	42	42	41	45
2^{31}	2^{17}	2^{10}	0.064s	110	435	428	46	44	47	46
2^{31}	2^{18}	2^{10}	0.063s	102	440	434	46	39	54	47
2^{31}	2^{19}	2^{10}	0.063s	104	439	434	51	46	47	47
2^{31}	2^{20}	2^{10}	0.064s	110	433	430	46	42	46	52
2^{31}	2^{20}	2^{11}	0.035s	125	758	747	54	37	49	37
2^{31}	2^{20}	2^{12}	0.022s	101	1216	1206	51	31	52	32
2^{31}	2^{20}	2^{13}	0.013s	60	1985	1961	110	67	1996	1979
2^{31}	2^{20}	2^{14}	0.008s	77	3356	3304	110	58	3276	3314
2^{31}	2^{20}	2^{15}	0.004s	74	5985	6288	60	111	6016	6057

Table 13: Total Allocated Memory `T` = 2^{30}, as shown in P0089

Total Allocated Memory `T` = 2^{31} through 2^{35}

Subsequent tests with larger total memory usages (`T`=2^{31} through 2^{35}) resulted in patterns similar to those seen in Table 12. The tables displaying the results for `T`=2^{31} through 2^{35} have been elided from the body of the report because they do not convey any new information. For those interested, the tables can be found in “Appendix 5: Elided results for Benchmark III”.

There was one notable difference between the benchmarks omitted from this section and the equivalent ones in P0089: The machine used to run these benchmarks had sufficient memory, so the latter benchmarks did not fail due to the monotonic allocator’s exhausting all available memory.

Conclusions

The results in Table 12 (as well as those in the elided tables in Appendix 5: Elided results for Benchmark III) clearly show that the multipool allocator ran in less than 50% of the time of the global allocator (AS1). This result is (at least in part) because the multipool allocator handled churn incredibly well. For the case when a system has a high level of churn (and objects that are within the multipool’s size limits) the multipool allocator offers a potential performance improvement. What’s more, the version of the multipool allocator used in this experiment was unsynchronized – another advantage that local allocators have over global ones. Even for modern global allocators that create separate thread-specific pools, having a dedicated unsynchronized local allocator eliminates needleless “bookkeeping” overhead in this single-threaded scenario.

12 Benchmark IV: Variation in Contention

Benchmark Overview

This benchmark was designed to examine the effects of contention (**C**) with respect to memory allocation and deallocation in a multithreaded environment.

The algorithm was as follows:

1) **Thread Creation**: Spawn `W` threads
2) **Allocation**: On each thread, allocate a chunk of size `S` bytes and increment the value of the first byte (to access that chunk)
3) **Deallocation**: Deallocate the previously allocated chunk
4) **Repeat**: Repeat the “Allocation” and “Deallocation” steps `N` times for each thread
5) **Wait**: Wait for all `W` threads to complete

Steps 1-5 were timed together for this benchmark. Note that for this benchmark wall time (`std::chrono::system_clock::now()`) was used because the benchmark needed to track the time required for *all* threads to finish. It was simpler to measure the wall time on the `main` thread from the first thread start to the last thread exit, rather than individually tracking and summing the CPU time of each thread. The “wink out” allocator strategies were omitted in this test, for the same reasons presented in Chapter 11: “Benchmark III: Variation in Utilization”.

One change that was made from the benchmark described in Chapter 10: “Benchmark IV: Variation in Contention” from P0089R0, was that the number of iterations, `N`, was increased by a factor of 100 for each test. This change was introduced to decrease the noise relative to the productive work being measured. Note that for each of the 100 repetitions of the original N iterations, a new allocator was created and used. For example, in the test with $N=100*2^{15}$ iterations, the old allocator was destroyed and a new allocator was created and used after every 2^{15} allocations. Creating this new allocator object meant that each allocator allocated the same amount of memory as in the original test. Keeping the memory per allocator object constant prevented allocators such as the monotonic allocator from being unfairly penalized by the increased memory usage that would have otherwise resulted.

It is important to note that, in this test, a separate allocator was created on each thread. In the case of AS1 and AS2, the global allocator must handle concurrent access. Allocators AS3, AS5, AS7, AS9, AS11, and AS13 are local allocators that are not designed for concurrent access, and then don’t suffer any performance penalties to support it. The one twist is that these local allocators will still have to pay the overhead of concurrency support when they (comparatively rarely) employ the backing global allocator to get additional chunks of memory.

Benchmark Presentation

The results in this section are presented in a similar fashion to those in Chapter 11: "Benchmark III: Variation in Utilization". The results presented for allocation strategies AS2, AS3, AS5, AS7, AS9, AS11, and AS13 in this chapter are displayed as percentages of the run time for AS1. The absolute run times can be found in "Appendix 6: Absolute Run Times for Benchmark IV". All of the tables are colored as heat maps, with the midpoint (yellow) fixed at 100%. Thus, if a result is green it represents an improvement over AS1, if it is red, it represents a longer run time than AS1, and if it is yellow it took about the same time to run as AS1.

Number of Iterations $N = 100*2^{15}$, Size of Allocation $S = 2^6$

This section shows the relative run times of AS2, AS3, AS5, AS7, AS9, AS11, and AS13 to AS1 for a system with a total of `N=100*2`^{15} allocations and deallocations of chunks of `S=2`^{6} size. The number of threads was varied as shown in Table 14.

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{15}$	2^6	1	0.103	102	25	25	47	48	48	47
$100*2^{15}$	2^6	2	0.103	102	50	25	48	48	48	47
$100*2^{15}$	2^6	3	0.103	102	25	25	47	48	48	47
$100*2^{15}$	2^6	4	0.103	102	25	25	76	48	48	47
$100*2^{15}$	2^6	5	0.181	105	28	29	44	45	43	45
$100*2^{15}$	2^6	6	0.181	105	28	28	44	44	45	44
$100*2^{15}$	2^6	7	0.186	103	32	29	44	44	43	43
$100*2^{15}$	2^6	8	0.192	100	30	28	43	50	47	44

Table 14: Number of Iterations $N = 100*2^{15}$, Size of Allocation $S = 2^6$

The first observation is that all of the local allocators consistently offer a performance improvement over the default global allocator. The monotonic allocator typically ran in 25%-32% of the time of the global allocator (with one outlier at 50%), the multipool allocator typically ran in 44%-50% of the time of the global allocator (with one outlier at 76%), and the multipool + monotonic allocator ran in 43%-48% of the time. These observations make sense, given that none of the local allocators have to be instrumented for (nor handle) concurrent usage.

The second observation is that it does not seem to make a difference in this test whether the multipool allocator was backed by a monotonic allocator or the global allocator. This result also makes sense, since the multipool allocator needs to make only one request to its backing allocator, for enough memory to hold a single chunk of size S (or perhaps some typically small multiple of S, depending on the growth strategy). After this initial allocation from the backing allocator, the multipool will (soon) simply be popping one chunk on to and off of its free list as the test deallocates and reallocates one chunk at a time.

A third observation is that there was a minimal overhead of 0%-5% of AS1's run time, when accessing the global allocator through a virtual function call (AS2).

One strange result, which is more visible in the raw numbers (see "Appendix 6: Absolute Run Times for Benchmark IV"), is that performance consistently degraded when the number of threads, W, was 5 or more. This phenomenon is not noticeable in the relative numbers presented in Table 14 because all of the allocation strategies experience the same performance degradation. This phenomenon was also not observed in the numbers presented in P0089, nor in smaller tests run on a local machine, so it is likely that this degradation is a quirk of how Amazon Web Services (AWS) allocates compute resources to virtual servers. This suspected quirk was further confirmed when the code written for P0089 was run on an AWS server, and the jump in runtime at W=5 was seen.

The equivalent results from P0089 are presented below in Table 15. Note that, while the chunk size, S, and number of threads, W, are exactly the same between Table 14 and Table 15, the number of iterations, N, in Table 14 are 100x higher. The rationale for this deviation from R0089 was discussed in the previous section. Despite the difference in the number of iterations, the relative run times should still be comparable. Additionally, note that Table 15 was recolored according to the coloring strategy outlined in the first section of this chapter. This coloring strategy means that the midpoint of the coloring, yellow, was fixed at 100% of AS1, rather than halfway between the minimum and maximum of the values in the colored range. Because of this recoloring, any cells that are green indicate better performance relative to AS1, and any that are red similarly indicate worse performance.

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{15}	2^6	1	0.041s	91	40	39	26	26	24	24
2^{15}	2^6	2	0.037s	100	42	43	27	26	26	29
2^{15}	2^6	3	0.038s	105	41	43	15	16	17	16
2^{15}	2^6	4	0.032s	93	56	58	31	32	25	24
2^{15}	2^6	5	0.032s	91	46	52	26	23	22	24
2^{15}	2^6	6	0.030s	95	51	53	24	27	26	27
2^{15}	2^6	7	0.033s	96	47	49	23	28	21	26
2^{15}	2^6	8	0.029s	96	71	63	33	30	31	25

Table 15: Number of Iterations N = 2^{15}, Size of Allocation S = 2^6, from P0089

The results from P0089 are somewhat noisy, so comparing the results on a point-by-point basis would be futile. Utilizing the heat map colorization, however, macro scale patterns for the whole benchmark can be observed. The main pattern seen in the P0089 numbers is that all of the local allocators performed better than the global allocator. The allocation strategies using a multipool (AS7, AS9, AS11, and AS13) performed the best. The monotonic allocator ran in 39%-71% of the time taken by AS1, the multipool allocator ran in 15%-32% of the time taken by AS1, and the multipool + monotonic allocator ran in 16%-31% of the time taken by AS1. This improvement of the local allocators over AS1 was also seen in the results from P0089.

One counter-intuitive result from P0089 is that the global allocator more often than not performed better when accessed through a virtual function call (AS2) than when accessed directly (AS1). It is possible that the virtual function call could be elided, resulting in no performance overhead, but there does not seem to be a clear explanation for why performance would *improve* when accessed through a virtual function call. This strange result was not seen in the tests run for this paper.

Where the results from P0089 and the results in this paper diverge the most is in the performance of the monotonic allocator (AS3 and AS5). Investigation into the benchmarking code for this paper and P0089 revealed the difference: P0089 relied in a monotonic allocator that had no initial buffer, and grew geometrically. The monotonic allocator used in this paper was given a statically allocated buffer of 2^{30} bytes, from which to distribute memory (which matches what was done in Benchmark I for both this paper and P0089). The benchmarking code for this paper was re-run without the static buffer and the results are presented in Table 16. All of the absolute run times for Benchmark IV, re-run without the static buffer, can be found in “Appendix 7: Absolute Run Times for Benchmark IV, with static buffer removed”.

The performance of the monotonic allocator (AS3 and AS5) degraded significantly when the static buffer was removed. This degradation was most likely because the allocator had to go to the backing global allocator multiple times, to feed its geometric growth. Because the monotonic allocator never gives up memory, the tests in columns AS3 and AS5 would have had to maintain much more globally allocated memory from the global allocator than the tests in AS1 and AS2. As the number of iterations, `N`, or chunk size, `S`, increases in further tests, the amount of globally allocated memory held by the monotonic allocator will increase. This increase would likely result in further performance degradations.

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{15}	2^{6}	1	0.103	102	73	73	47	48	47	47
2^{15}	2^{6}	2	0.104	101	76	76	47	48	47	47
2^{15}	2^{6}	3	0.103	102	78	77	47	48	47	47
2^{15}	2^{6}	4	0.104	102	78	78	47	48	47	47
2^{15}	2^{6}	5	0.180	106	67	68	43	44	44	40
2^{15}	2^{6}	6	0.192	100	64	65	41	41	42	42
2^{15}	2^{6}	7	0.182	105	70	73	45	45	44	45
2^{15}	2^{6}	8	0.188	104	74	72	49	49	44	46

Table 16: Number of Iterations `N` = 2^{15}, Size of Allocation `S` = 2^{6}, without static buffer

Ultimately, the results presented in Table 14 (results from this paper, with static buffer), Table 15 (results from P0089, without static buffer), and Table 16 (results from this paper, without static buffer) do not quite agree, however, the following broad conclusions can be drawn:

- All of the local allocation strategies performed significantly better than the global allocator (AS1)
- The monotonic allocator (AS3 and AS5) performed the best, when given a pre-allocated static buffer
- The multipool based allocation strategies (AS7, AS9, AS11, and AS13) performed the best when the monotonic allocator (AS3 and AS5) did not benefit from a static buffer

Further Tests

Further tests were performed on variations of the number of iterations, `N`, and size of allocations, `S`, to match those done in P0089. The results when the monotonic allocator *was* supplied a static buffer did not change from those presented in Table 14. Thus, those results were elided for the sake of brevity.

The results when the monotonic allocator *was not* supplied a static buffer did show a degradation in performance as the number of iterations, `N`, and the size of the allocated chunks, `S`, were increased. This result is as expected and predicted in the previous section. This degradation in performance matches the degradation seen in the equivalent results in P0089.

Conclusions

In this multithreaded benchmark, thread-local allocators improved performance by up to 6x over the global allocator. If allocated memory does not need to be accessed from other threads, a multipool allocator offers significant and consistent performance benefits over the global allocator. If there is enough memory to supply a local monotonic allocator with a sufficiently large static buffer, it can offer the most significant performance improvements. The monotonic allocator, however, should be used with caution because it can consume much more memory than the object it is backing requires.

13 Benchmark V: Creating and Destroying Data Structures with Varied Locality and Fragmentability

Benchmark Overview

This 5th benchmark was created in an attempt to more closely mimic real world programs. All of these benchmarks have been done in isolation, which ignores the reality of most software, where many different subsystems are using the (same) global allocator. In a real program, before a subsystem of interest runs, it is likely that memory has already been allocated and deallocated. Additionally, it is likely that some of this memory is currently held by other subsystems.

This benchmark is essentially a combination of Benchmark I and II. The algorithm can be described as:

1) **Global Allocator Usage**: The global allocator is used to simulate a real world system
    a. **Allocate**: Allocate 2^{16} randomly sized chunks of memory
    b. **Deallocate**: Randomly deallocate `D` of the `N` chunks
2) **Testing**: The allocation under test is performed
    a. **Allocate**: Allocate the outer data structure for DS## (where DS## could be DS1-DS12)
    b. **Reserve**: Reserve space for `E` elements in the data structure
    c. **Populate**: Fill the allocated data structure with `E` elements, allocated using the same allocator
    d. **Deallocate**: Deallocate the data structure normally or via the “wink out” technique
    e. **Repeat**: Perform the Allocate through Deallocate steps 2,560 times

The testing was performed on DS1 through DS12, just like in Benchmark I. The number of randomly sized chunks, 2^{16}, was chosen arbitrarily. The random sizes for the chunks of memory were taken from a uniform distribution on the range [1, 1024]. Four tables were produced for each data structure, corresponding to `D=N`, `D=N/2`, `D=N/4`, and `D=N/8`. To further explain: all, one half, one quarter, and one eight of the memory was respectively deallocated, while the rest remained allocated as the “Testing” step proceeded.

Using the language of P0089, this test will have the same density (**D**) and variation (**V**) as originally characterized for DS1 through DS12. Where this test differs from Benchmark I is in diffusion potential, a.k.a., *fragmentability* (**F**). The "Global Allocator Usage" step of the algorithm introduces another subsystem to the process, opening up the potential for diffusion to occur during the "Testing" step. Note that data structures that exist in one contiguous chunk of memory (DS1: `vector<int>`, and DS5: `vector<vector<int>>`), do not allocate multiple chunks of memory that can diffuse – i.e., the *fragmentability* (**F**) is zero.

Benchmark Presentation

The hypothesis is that local allocators will reduce the data structure elements' diffusion throughout memory. Thus, changes in the performance of the local allocators relative to the global allocator are of interest. This change in performance can be expressed mathematically as a ratio of ratios:

$$R = \frac{\left(\frac{Run\,Time\ of\ AS\#\#\ in\ Benchmark\ V}{Run\,Time\ of\ AS1\ in\ Benchmark\ V}\right)}{\left(\frac{Run\,Time\ of\ AS\#\#\ in\ Benchmark\ I}{Run\,Time\ of\ AS1\ in\ Benchmark\ I}\right)}$$

If the ratio `R` is less than one, it indicates that the run time of the allocation strategy under examination has been less affected by the diffusion of system memory than the global allocator. This improved ratio indicates an opportunity to preserve runtime performance over the status quo.

The tables in this section present the ratio `R` for each data point. All of the tables are colored as heat maps, with the midpoint (yellow) fixed at 1. Thus, if a result is green it represents an improvement relative to the global allocator (AS1) when system memory is diffused. If a value is red, it represents worse performance relative to the global allocator. If the value is yellow, there was no significant change. Note that, because this ratio incorporates four different data points, it also incorporates 4x the noise, resulting in less smooth looking results than desired. Also note that the AS1 column has been omitted because, mathematically, it will always be unity.

DS1, vector<int>

Table 17 presents the ratio of Benchmark V to Benchmark I for DS1 (`vector<int>`), when `D=N`. Recall that `D=N` means that 2^{16} randomly sized chunks of memory were allocated and then deallocated before the timed portion of the benchmark was run. Provided that the system reclaims the deallocated memory completely and efficiently, it would be as if there was no other subsystem in the test, meaning that there was no diffusion potential during the "Testing" step. Thus, there should be no difference between the results from Benchmark V and Benchmark I, resulting in all the ratios in Table 17 being unity. Inspecting Table 17, this is indeed the case. Note that the heat map is misleading, because the "extreme" values it highlights in red and green are within 2% of the neutral (yellow) value of 1.

	global	←		monotonic	→	←		multipool	→	←		multi + mono	→
	virtual				← virtual →				← virtual →				← virtual →
			(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.00	0.99	0.99	0.99	0.99	0.99	1.00	0.99	0.99	0.98	0.99	1.00	0.99
2^{7}	0.99	0.99	0.99	0.99	0.99	0.98	0.99	0.99	0.99	0.99	0.99	1.00	0.99
2^{8}	1.00	1.00	1.00	0.99	0.99	0.99	0.99	1.00	0.99	0.99	0.99	1.00	0.99
2^{9}	1.00	1.00	1.00	0.99	0.99	0.99	1.00	1.00	1.00	0.99	1.00	1.00	1.00
2^{10}	1.01	1.00	1.00	0.99	0.99	1.00	1.00	1.00	0.99	1.00	0.99	0.98	0.99
2^{11}	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.01	1.01	1.00
2^{12}	1.00	1.00	0.99	1.00	1.01	1.00	1.00	1.00	1.00	1.00	1.02	1.01	1.01
2^{13}	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.99
2^{14}	1.00	1.00	1.01	1.01	1.01	1.01	1.00	1.01	1.01	1.01	1.01	1.01	1.00
2^{15}	1.01	1.00	1.00	1.00	1.01	1.00	1.00	1.01	1.01	1.01	1.01	1.01	1.00
2^{16}	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00

Table 17: Ratio of Benchmark V to Benchmark I for DS1 (`vector<int>`), when `D=N`

Table 18 presents the ratio of Benchmark V to Benchmark I for DS1 (`vector<int>`), when `D=N/2`. Recall that `D=N/2` means that 2^{16} randomly sized chunks of memory were allocated and then 2^{15} (half) were deallocated before benchmark was run (the other half remained allocated for the duration of the test).

For a `vector` of `int`s, all of the data resides in one contiguous chunk of memory. Thus, regardless of the allocation strategy, the diffusion potential is minimal. There is no opportunity for a local allocator to prevent diffusion, so it would be expected that allocation strategy performance relative to the global allocator would not change, resulting in a table of all 1s. This is indeed the result seen in Table 18, with the exception of some outliers in the first two rows, which defy explanation.

	global	←		monotonic	→	←		multipool	→	←		multi + mono	→
	virtual				← virtual →				← virtual →				← virtual →
			(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.00	0.99	1.13	1.14	1.13	1.13	1.24	1.07	1.24	1.04	1.12	1.13	1.13
2^{7}	1.00	0.99	1.07	1.07	1.07	1.07	1.24	1.09	1.25	1.08	1.08	1.07	1.07
2^{8}	1.00	1.00	1.03	1.03	1.03	1.02	1.09	1.05	1.10	1.06	1.03	1.03	1.03
2^{9}	1.00	1.00	1.02	1.02	1.03	1.01	1.04	1.03	1.04	1.03	1.02	1.02	1.02
2^{10}	1.00	1.00	1.02	1.01	1.01	1.01	1.01	1.01	1.01	1.00	1.02	1.02	1.00
2^{11}	1.00	1.00	1.00	1.00	0.99	0.99	1.00	1.00	1.00	1.00	1.00	1.00	0.99
2^{12}	1.00	1.00	1.00	0.99	1.00	1.00	1.00	1.00	1.00	1.01	1.00	1.00	1.00
2^{13}	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.00	1.01	1.00	1.00	1.00
2^{14}	1.00	1.00	1.00	1.01	1.01	1.01	1.01	1.02	1.01	1.01	1.01	1.01	1.01
2^{15}	1.00	1.01	1.01	1.00	1.00	1.00	1.01	1.02	1.03	1.02	1.01	1.01	1.00
2^{16}	1.00	1.00	1.00	1.00	1.00	1.00	1.01	1.01	1.01	1.02	1.00	1.00	1.00

Table 18: Ratio of Benchmark V to Benchmark I for DS1 (`vector<int>`), when `D=N/2`

The remaining measurements for DS1 (`D=N/4` and `D=N/8`) are omitted because they are nearly identical to Table 18.

As expected, there is no change between the results in Benchmark I and Benchmark V for DS1, most likely because DS1 does not allocate chunks of memory cable of diffusing, and thus the test has minimal fragmentability (**F**). This was also seen to be the case for DS5 (`vector<vector<int>>`), which also has minimal fragmentability (**F**).

DS2, vector<string>

Unlike DS1, DS2 (`vector<string>`) has a high potential to diffuse because the data for each contained string can be allocated in a different chunk of memory. Table 19 presents the ratio of Benchmark V to Benchmark I for DS2, when `D=N/2`. Recall that `D=N/2` means that 2^{16} randomly sized chunks of memory were allocated and then 2^{15} (half) were deallocated before the benchmark was run (the other half remained allocated for the duration of the test).

	global	←		monotonic	→	←		multipool	→	←		multi + mono	→
	virtual				← virtual →				← virtual →				← virtual →
			(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.00	0.94	0.42	0.42	0.42	0.42	0.53	0.54	0.51	0.55	0.43	0.43	0.42
2^{7}	1.00	0.94	0.51	0.52	0.51	0.52	0.65	0.66	0.64	0.62	0.51	0.51	0.51
2^{8}	1.00	1.17	0.36	0.36	0.36	0.36	0.34	0.32	0.33	0.33	0.36	0.36	0.36
2^{9}	1.00	1.01	0.56	0.56	0.55	0.55	0.55	0.56	0.54	0.54	0.56	0.56	0.56
2^{10}	1.00	0.95	0.55	0.55	0.56	0.56	0.57	0.56	0.54	0.54	0.55	0.55	0.55
2^{11}	1.00	0.38	0.35	0.35	0.35	0.35	0.42	0.38	0.40	0.45	0.35	0.35	0.35
2^{12}	1.00	0.73	0.85	0.85	0.85	0.85	0.78	0.83	0.81	0.99	0.85	0.85	0.85
2^{13}	1.00	0.62	0.81	0.82	0.82	0.82	0.96	0.97	0.77	1.15	0.81	0.81	0.82
2^{14}	1.00	0.92	0.80	0.80	0.84	0.77	0.75	0.69	0.52	0.70	0.80	0.79	0.70
2^{15}	1.00	1.55	0.90	0.67	0.97	1.03	0.88	0.97	0.95	0.93	0.52	0.76	0.66
2^{16}	1.00	1.08	0.89	0.89	0.89	0.88	0.92	0.89	0.86	0.89	0.87	0.89	0.88

Table 19: Ratio of Benchmark V to Benchmark I for DS2 (vector<int>), when `D=N/2`

Table 19 shows the performance of every one of the local memory allocation strategies improving relative to the global allocator (AS1), when subsystem memory has the potential to defuse – i.e., the subsystem has non-zero *fragmentability* (**F**). This observation seems to validate the hypothesis that local allocators help to prevent diffusion of a data structures memory, thus improving performance relative to the global allocator (AS1).

The remaining tables for this DS2 test (`D=N`, `D=N/4` and `D=N/8`) have been omitted. The `D=N` table for DS2 looks incredibly similar to the `D=N` table for DS1 (Table 17), as do all of the `D=N` tables for the remaining tests, DS3-DS12. The explanation remains the same: Allocating and then deallocating a bunch of memory is likely not too different from starting the benchmark in a fresh process, and so the subsystem under test is not fragmentable (**F**). No change was seen from Benchmark I to Benchmark V when `D=N`. The other tables for DS2, where `D=N/4` and `D=N/8`, are omitted because they are nearly identical to Table 18.

This benchmark seems to show that local allocator performance (further) improves relative to the global allocator (AS1) when the data structure under test has the potential for its memory to diffuse – i.e., *fragmentability* (**F**) is greater than zero.

Further Benchmarks

Further results are omitted because they show similar behavior, with likely similar explanations, to DS2 (`vector<string>`). The one exception is DS5 (`vector<vector<int>>`), which was also allocated in one contiguous chunk, providing no opportunity for its memory to be diffuse, which means fragmentability (**F**) for both DS1 and DS5 is minimal. Consequently, DS5 behaved similarly to DS1. DS5 also stood out from all the other tests in one way: The performance of the global allocator (AS1 and AS2) inexplicably improved when the "Global Allocator Usage" step (step 1 of the algorithm) was introduced to allow *fragmentability* (**F**) to manifest.

The omitted tables are too numerous, even for the appendices. All raw data, as well as the code to generate it, can be found in this GitHub repository:
https://github.com/gbleaney/Allocator-Benchmarks/tree/master/benchmarks/allocators

Analysis

Table 20 shows the average results for Benchmark V across all of the tests for DS1-DS12 where the system had some amount of *fragmentability* (**F**) – i.e. the tests with `D=N/2`, `D=N/4`, and `D=N/8`. Recall that `D` is the number of randomly sized chunks of memory that were deallocated (from a total of `N` allocated) before the benchmark was run. The table shows that the execution time of the local allocation strategies (AS3 through AS14) relative to that of the global allocator improved by an average of ~20%.

global	←		Monotonic	→	←		multipool	→	←		multi + mono	→
virtual				← virtual →				← virtual →				← virtual →
		(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
0.96	0.80	0.81	0.79	0.80	0.78	0.79	0.80	0.80	0.81	0.80	0.80	0.80

Table 20: The average results for Benchmark V across all of the tests with diffuse system memory (`D=N/2`, `D=N/4`, `D=N/8`)

Conclusions

Benchmark I previously demonstrated that local allocators, monotonic (AS3 through AS6) and multipool + monotonic (AS11 through AS14) in particular, can improve upon the runtime of the default global allocator when large amounts of memory are allocated and deallocated with very little churn. This benchmark (Benchmark V) shows that the performance gap can widen even further (an additional 20%) when the overall system allows the memory of subsystems having non-zero *fragmentabili*ty (**F**) to defuse. The results of this benchmark reinforce the hypothesis that local memory allocators can improve system performance by preventing the diffusion of data structure contents throughout global memory.

14 Conclusion

This paper was created to investigate the results presented in P0089R0. Some results in that paper were found to be erroneous and will be updated in P0089R1 and all future revisions.

Benchmarks I through V were designed to examine the effects of various memory allocation strategies on runtime performance. Benchmarks I through IV were implemented from the descriptions provided in P0089, and Benchmark V was created as a hybrid of Benchmark I and Benchmark II. Each benchmark demonstrated situations where one or more allocation strategies using a local allocator (AS3-AS14) improved runtime performance compared to that of the global allocator (AS1).

Benchmark I determined that the monotonic allocator, when given a static buffer, provided the largest performance improvement. It was advised to use the monotonic allocator (or multipool + monotonic allocator) in situations where large amounts of memory are being allocated, used, and then deallocated, without high churn.

Benchmark I also concluded that the “winking out” technique provides a sizable runtime benefit (8.4% reduction in run time) and should be considered when possible. Finally, accessing an allocator through a virtual function call had a small, but measurable, runtime performance overhead (0.78% increase in run time). Whether or not the convenience is worth the overhead will vary from use case to use case and platform to platform. There is every reason to believe that this overhead can be elided as indicated in P0089.

Benchmark II concluded that a local multipool allocator offered a large performance benefit over the global allocator. It was recommended to use a local multipool allocator in situations where the potential for diffusion across subsystems exists, and *fragmentability* (**F**) is high. Evidence was also presented to confirm that a lack of *locality* (**L**) has a negative effect on performance.

Benchmark III saw the multipool allocator run in less than 50% of the time of the global allocator. For the case when a system has a high level of churn (and objects that are within the multipool’s size limits) the multipool allocator was recommended to improve performance, especially in a single-threaded context.

Benchmark IV demonstrated that thread-local allocators offer performance savings of up to 6x over the global allocator. The multipool allocator was recommended in multithreaded situations where allocated memory does not need to be accessed from other threads. The monotonic allocator was also recommended, provided there is enough memory to supply the monotonic allocator with a sufficiently large initial buffer.

Finally, Benchmark V concluded that the benefits offered by the local allocators improve by average of 20% when the process provides the potential to diffuse memory and the *fragmentability* (**F**) of the subsystem in question is non-zero. It is in precisely these cases where local allocators prove most useful compared to the global one — irrespective of how efficient that general-purpose, global allocator might be.

Appendix 1: Corrected Benchmark I Results from P0089

This section contains the corrected results for Benchmark I. The incorrect results were originally presented in P0089R0 with data in columns [AS3 – AS7] and [AS8-AS11] transposed. The corrected results should appear in P0089R1.

	← global →		← monotonic →				← multipool →				← multi + mono →			
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	1.18	1.86	0.27	0.37	0.42	0.44	0.80	1.01	0.90	1.06	0.62	0.69	0.77	0.67
2^{7}	0.92	1.59	0.25	0.39	0.41	0.41	0.51	0.70	0.64	0.70	0.45	0.51	0.60	0.52
2^{8}	0.81	1.00	0.25	0.38	0.38	0.35	0.35	0.59	0.51	0.61	0.31	0.46	0.51	0.47
2^{9}	0.75	0.95	0.22	0.36	0.39	0.35	0.31	0.46	0.45	0.46	0.26	0.43	0.42	0.40
2^{10}	0.74	0.94	0.21	0.34	0.38	0.36	0.24	0.43	0.43	0.42	0.25	0.38	0.40	0.38
2^{11}	0.75	0.94	0.21	0.33	0.36	0.32	0.22	0.39	0.40	0.41	0.24	0.38	0.37	0.38
2^{12}	0.74	0.94	0.21	0.34	0.38	0.36	0.22	0.37	0.39	0.40	0.22	0.37	0.37	0.37
2^{13}	0.76	0.93	0.20	0.32	0.36	0.40	0.21	0.38	0.39	0.37	0.24	0.36	0.37	0.37
2^{14}	0.77	0.93	0.20	0.33	0.39	0.39	0.21	0.38	0.36	0.38	0.20	0.36	0.39	0.37
2^{15}	0.77	0.94	0.20	0.32	0.37	0.37	0.21	0.39	0.36	0.39	0.21	0.36	0.38	0.36
2^{16}	0.78	0.94	0.21	0.36	0.36	0.37	0.21	0.36	0.36	0.39	0.20	0.36	0.37	0.36

Table 21: DS1, vector<int>

	← global →		← monotonic →				← multipool →				← multi + mono →			
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	68.90	67.30	12.90	12.80	13.30	12.90	18.10	17.80	18.20	17.70	15.50	14.80	15.60	14.80
2^{7}	68.80	68.20	12.80	12.90	13.20	12.90	20.60	20.20	20.60	20.40	15.10	14.30	15.00	14.40
2^{8}	70.80	68.90	13.20	12.80	13.60	12.90	30.80	30.40	30.70	30.30	15.30	14.60	15.40	14.70
2^{9}	73.10	71.20	13.50	13.50	13.90	13.50	38.20	37.60	38.00	37.30	15.90	15.10	15.90	15.10
2^{10}	75.40	74.30	13.60	13.50	14.00	13.70	41.10	40.30	41.60	40.90	16.00	15.10	15.90	15.00
2^{11}	76.90	74.50	13.60	13.50	14.10	13.60	43.90	43.20	43.70	42.60	16.00	15.00	16.00	15.10
2^{12}	76.10	74.80	13.70	13.50	14.00	13.60	41.20	38.80	40.60	39.40	15.90	14.90	15.80	15.00
2^{13}	76.10	74.80	13.60	13.60	14.00	13.60	41.40	39.20	41.30	39.90	15.90	15.00	15.80	14.90
2^{14}	78.30	76.50	13.60	13.60	14.00	13.60	45.80	42.30	44.80	44.00	16.10	15.20	16.20	15.40
2^{15}	90.40	91.00	20.20	20.10	20.50	20.10	62.20	58.70	62.20	58.20	26.00	25.00	26.00	24.90
2^{16}	103.00	103.00	21.50	21.30	21.80	21.30	66.50	59.20	65.10	59.90	27.00	25.30	27.10	25.20

Table 22: DS2, vector<string>

	← global →		← monotonic →				← multipool →				← multi + mono →			
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14

2^{6}	10.20	11.00	5.08	4.88	5.62	5.34	7.16	7.12	7.50	7.20	6.19	5.73	6.40	5.81
2^{7}	12.50	13.30	5.04	4.81	5.68	5.24	6.37	6.22	6.71	6.31	5.80	5.46	6.08	5.50
2^{8}	15.80	16.40	4.99	4.79	5.54	5.22	5.95	5.81	6.21	5.92	5.65	5.32	5.82	5.40
2^{9}	18.30	19.00	5.01	4.80	5.53	5.18	5.78	5.56	6.01	5.70	5.56	5.20	5.76	5.21
2^{10}	21.40	22.30	4.99	4.83	5.55	5.20	5.72	5.46	5.95	5.55	5.52	5.27	5.68	5.24
2^{11}	25.50	26.10	4.98	4.81	5.56	5.16	5.67	5.44	5.86	5.65	5.53	5.23	5.69	5.26
2^{12}	27.10	28.00	5.02	4.81	5.55	5.20	6.42	6.10	6.57	6.25	5.51	5.12	5.68	5.27
2^{13}	27.90	28.80	5.03	4.81	5.59	5.21	7.34	6.91	7.46	7.03	5.61	5.16	5.71	5.24
2^{14}	28.50	29.00	5.03	4.80	5.58	5.26	7.03	6.59	7.18	6.68	5.64	5.19	5.80	5.34
2^{15}	28.30	29.20	5.03	4.78	5.56	5.28	7.11	6.65	7.20	6.83	5.68	5.17	5.78	5.24
2^{16}	31.60	31.80	5.02	4.76	5.60	5.22	6.79	6.37	6.93	6.46	5.68	5.17	5.79	5.24

Table 23: DS3, unordered_set<int>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	103.00	120.00	52.20	51.90	52.40	51.20	58.40	57.60	59.70	58.90	55.10	54.10	56.90	55.30
2^{7}	103.00	122.00	52.50	52.10	52.90	51.80	63.30	61.90	64.40	63.80	55.30	54.00	56.80	55.70
2^{8}	109.00	128.00	53.60	53.00	53.70	52.60	76.30	74.70	77.40	75.90	56.50	54.90	57.90	56.70
2^{9}	113.00	134.00	54.50	53.40	54.90	53.00	83.10	81.70	82.80	81.40	57.30	56.70	58.00	56.40
2^{10}	119.00	143.00	56.60	54.90	56.90	54.60	87.60	85.90	88.10	86.50	58.80	56.90	59.20	57.30
2^{11}	122.00	144.00	57.00	55.30	57.70	54.90	90.70	89.20	90.70	88.40	59.40	57.60	60.00	57.80
2^{12}	122.00	146.00	57.90	55.90	58.40	55.70	93.20	90.70	93.20	90.70	60.50	58.30	60.70	58.40
2^{13}	124.00	148.00	58.20	56.30	58.50	55.90	95.10	91.50	94.30	92.00	60.50	58.20	60.70	58.70
2^{14}	139.00	166.00	59.10	57.30	59.60	56.80	98.50	94.10	97.80	95.80	61.80	59.60	62.20	60.00
2^{15}	176.00	211.00	66.00	62.70	66.20	62.40	121.00	115.00	122.00	115.00	76.50	73.30	76.80	74.00
2^{16}	196.00	232.00	78.50	72.00	79.10	71.00	137.00	127.00	136.00	127.00	87.10	82.40	87.80	82.90

Table 24: DS4, unordered_set<string>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	0.97	1.00	0.19	0.13	0.20	0.17	0.24	0.20	0.20	0.21	0.21	0.19	0.20	0.21
2^{7}	0.96	0.96	0.22	0.16	0.18	0.14	0.21	0.20	0.19	0.20	0.16	0.20	0.21	0.19
2^{8}	0.99	1.00	0.19	0.13	0.18	0.17	0.27	0.30	0.27	0.29	0.19	0.19	0.20	0.21
2^{9}	0.99	1.02	0.19	0.13	0.18	0.14	0.36	0.33	0.33	0.36	0.19	0.15	0.20	0.20
2^{10}	1.01	1.04	0.19	0.18	0.19	0.14	0.37	0.36	0.36	0.38	0.22	0.19	0.20	0.22
2^{11}	1.02	1.05	0.19	0.13	0.19	0.14	0.36	0.35	0.36	0.36	0.20	0.15	0.20	0.22
2^{12}	1.03	1.05	0.19	0.19	0.22	0.18	0.33	0.36	0.32	0.32	0.20	0.21	0.20	0.19
2^{13}	1.02	1.05	0.19	0.13	0.22	0.19	0.35	0.35	0.34	0.33	0.20	0.21	0.22	0.19
2^{14}	1.05	1.10	0.19	0.17	0.19	0.16	0.38	0.36	0.38	0.37	0.17	0.19	0.20	0.19
2^{15}	1.13	1.18	0.22	0.19	0.19	0.16	0.50	0.45	0.47	0.45	0.21	0.21	0.17	0.18

2^{16}	1.29	1.32	0.22	0.19	0.20	0.17	0.54	0.47	0.52	0.50	0.22	0.21	0.22	0.21

Table 25: DS5, vector<vector<int>>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	72.60	72.70	9.06	9.06	9.36	8.98	41.70	40.00	41.20	39.20	11.20	10.30	11.20	10.30
2^{7}	74.90	76.00	8.92	8.98	9.29	8.89	46.50	44.80	46.00	43.00	11.40	11.00	12.70	10.30
2^{8}	85.50	85.20	17.10	17.40	17.30	16.90	62.90	58.40	61.30	58.40	22.80	22.50	23.30	22.00
2^{9}	96.40	96.30	18.40	18.70	19.00	18.40	66.20	59.00	64.70	59.30	24.20	22.70	24.50	22.30
2^{10}	102.00	102.00	18.70	18.60	19.10	18.60	67.00	59.60	65.90	59.00	24.80	22.50	24.80	22.50
2^{11}	102.00	101.00	18.40	18.70	19.20	18.20	62.40	55.00	61.30	54.20	24.80	22.60	25.10	22.30
2^{12}	104.00	103.00	18.50	18.70	19.40	18.30	61.60	54.20	60.50	53.40	24.90	22.70	25.10	22.30
2^{13}	103.00	104.00	18.80	18.40	19.00	18.60	61.80	53.40	59.90	53.50	25.30	22.60	25.10	22.60
2^{14}	97.10	96.30	19.20	19.60	20.10	19.20	60.60	53.70	60.20	52.90	29.00	26.70	29.20	26.30
2^{15}	88.10	88.70	23.40	23.20	23.70	23.40	62.60	54.40	60.90	53.90	33.40	30.60	33.20	30.70
2^{16}	76.70	76.70	25.00	25.30	25.80	25.00	63.40	54.80	62.90	54.30	35.00	32.80	35.50	32.40

Table 26: DS6, vector<vector<string>>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	28.80	28.70	2.97	2.69	3.43	2.98	4.89	4.37	5.33	4.73	3.21	2.65	3.64	3.05
2^{7}	28.30	28.50	2.97	2.66	3.36	2.95	4.99	4.44	5.43	4.91	3.20	2.62	3.61	2.97
2^{8}	28.20	28.10	2.94	2.62	3.33	2.92	5.02	4.53	5.53	4.97	3.23	2.60	3.60	3.01
2^{9}	31.80	31.70	2.92	2.61	3.33	2.93	5.08	4.54	5.52	4.92	3.16	2.58	3.58	2.96
2^{10}	46.60	47.20	2.92	2.61	3.33	2.89	5.07	4.49	5.48	4.93	3.15	2.58	3.57	2.98
2^{11}	54.30	54.10	2.92	2.61	3.33	2.89	5.63	4.75	5.88	5.37	3.16	2.60	3.61	2.98
2^{12}	54.70	54.80	2.96	2.66	3.34	2.91	6.90	5.79	7.28	6.23	4.15	3.05	4.58	3.40
2^{13}	55.10	56.00	3.51	2.95	3.77	3.21	7.01	6.03	7.47	6.35	4.27	3.08	4.65	3.48
2^{14}	51.00	50.90	3.53	2.99	3.81	3.25	7.08	6.00	7.47	6.46	4.29	3.14	4.71	3.47
2^{15}	44.80	45.40	3.58	3.01	3.83	3.26	7.07	6.04	7.55	6.52	4.35	3.14	4.75	3.53
2^{16}	38.20	38.20	3.58	3.06	3.86	3.30	7.14	6.11	7.58	6.47	4.37	3.18	4.80	3.54

Table 27: DS7, vector<unordered_set<int>>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14

2^{6}	114.00	116.00	26.00	23.80	26.30	24.00	56.20	54.70	56.90	54.60	27.50	25.80	27.90	26.00
2^{7}	123.00	130.00	26.50	24.40	25.70	23.50	62.70	60.10	62.70	60.50	27.50	26.30	28.20	26.10
2^{8}	162.00	171.00	31.70	27.30	32.20	27.80	78.00	74.20	79.20	73.90	35.00	32.00	35.50	32.50
2^{9}	175.00	181.00	36.80	28.00	38.10	28.00	81.70	74.10	81.20	74.90	36.30	32.10	37.20	32.10
2^{10}	176.00	183.00	40.00	28.90	37.40	28.20	82.10	74.50	82.10	74.70	36.90	32.00	37.40	32.20
2^{11}	176.00	183.00	39.30	28.00	37.30	28.00	81.40	74.40	82.00	74.30	36.90	32.10	37.80	32.10
2^{12}	179.00	185.00	39.40	28.00	37.10	28.00	81.80	74.10	81.60	74.40	37.00	32.00	37.80	32.20
2^{13}	173.00	178.00	39.60	27.90	36.90	28.20	81.80	73.60	81.50	74.30	37.20	32.00	37.80	32.40
2^{14}	157.00	160.00	41.00	29.90	38.80	29.90	81.50	74.10	82.20	74.00	44.00	39.30	45.10	39.20
2^{15}	122.00	131.00	47.60	35.80	44.80	36.20	85.20	75.50	83.70	76.10	50.50	45.20	51.00	45.50
2^{16}	95.40	106.00	51.40	40.50	48.10	38.90	84.80	76.20	88.70	75.90	53.10	48.50	54.80	48.20

Table 28: DS8, vector<unordered_set<string>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	0.97	0.94	0.23	0.19	0.24	0.21	0.26	0.27	0.30	0.26	0.25	0.26	0.25	0.24
2^{7}	1.40	1.43	0.22	0.21	0.22	0.19	0.24	0.26	0.25	0.27	0.24	0.26	0.24	0.24
2^{8}	1.35	1.39	0.25	0.22	0.24	0.23	0.30	0.35	0.34	0.33	0.24	0.23	0.25	0.24
2^{9}	1.29	1.32	0.22	0.18	0.22	0.17	0.37	0.38	0.37	0.36	0.23	0.22	0.19	0.22
2^{10}	1.32	1.38	0.24	0.22	0.22	0.19	0.41	0.39	0.42	0.39	0.23	0.24	0.23	0.22
2^{11}	1.34	1.36	0.23	0.21	0.22	0.17	0.44	0.42	0.43	0.41	0.23	0.23	0.25	0.22
2^{12}	1.34	1.41	0.22	0.20	0.22	0.16	0.46	0.42	0.45	0.43	0.23	0.17	0.27	0.22
2^{13}	1.46	1.54	0.22	0.18	0.22	0.16	0.48	0.49	0.49	0.48	0.23	0.21	0.25	0.21
2^{14}	1.53	1.61	0.22	0.17	0.22	0.18	0.43	0.42	0.45	0.41	0.24	0.22	0.24	0.22
2^{15}	1.61	1.76	0.25	0.21	0.24	0.19	0.50	0.49	0.50	0.49	0.24	0.18	0.23	0.21
2^{16}	1.79	1.92	0.28	0.25	0.29	0.24	0.55	0.51	0.56	0.55	0.30	0.23	0.32	0.24

Table 29: DS9, unordered_set<vector<int>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	73.00	73.20	9.41	9.39	9.34	8.97	41.70	39.70	41.10	39.30	11.20	10.40	11.20	10.30
2^{7}	74.70	75.30	9.32	9.34	9.24	8.87	46.20	43.70	45.30	44.20	12.70	10.60	11.40	10.80
2^{8}	83.10	85.40	18.00	17.30	16.90	17.20	62.20	58.90	61.90	57.60	23.20	22.30	23.10	22.40
2^{9}	91.40	94.90	19.00	19.00	18.80	18.60	65.00	59.90	64.40	58.90	24.30	22.60	24.10	22.60
2^{10}	98.20	101.00	19.20	18.90	19.10	18.60	66.50	59.70	65.40	59.10	24.80	22.60	24.60	22.70
2^{11}	99.50	101.00	19.00	19.10	19.30	18.40	66.90	59.50	66.10	58.70	24.90	22.70	25.10	22.50
2^{12}	102.00	105.00	19.40	19.00	19.20	18.80	67.00	58.90	65.80	59.40	25.30	22.60	25.10	22.70
2^{13}	103.00	104.00	19.00	19.20	19.40	18.40	66.70	59.20	66.20	58.20	25.30	22.90	25.50	22.60
2^{14}	95.80	97.20	19.80	20.00	20.30	19.30	62.80	55.60	61.90	54.30	29.20	26.80	29.60	26.50
2^{15}	87.10	89.80	24.00	23.70	24.00	23.50	64.30	55.00	61.90	54.90	33.60	30.80	33.50	31.00

2^{16}	77.10	78.20	25.60	25.70	26.00	25.10	63.90	55.50	63.30	54.50	35.30	33.00	35.70	32.60

Table 30: DS10, unordered_set<vector<string>>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	28.70	29.10	3.06	2.75	3.55	3.14	4.96	4.40	5.41	4.84	3.24	2.73	3.73	3.15
2^{7}	29.10	29.00	3.02	2.71	3.47	3.06	5.03	4.52	5.49	4.89	3.23	2.66	3.68	3.08
2^{8}	28.80	29.10	3.00	2.68	3.45	3.04	5.18	4.55	5.57	4.98	3.24	2.66	3.65	3.06
2^{9}	31.80	32.30	2.99	2.64	3.43	2.98	5.12	4.54	5.55	4.95	3.22	2.60	3.65	2.99
2^{10}	46.50	47.10	2.95	2.65	3.40	2.99	5.13	4.57	5.62	4.96	3.21	2.58	3.62	2.97
2^{11}	53.30	53.50	2.94	2.64	3.43	2.96	5.58	4.84	5.75	5.39	3.20	2.63	3.67	3.01
2^{12}	54.60	55.00	3.02	2.66	3.43	2.98	6.47	5.94	6.99	6.28	3.83	3.00	4.21	3.38
2^{13}	56.50	56.50	3.38	2.98	3.72	3.26	7.04	6.04	7.48	6.45	4.15	3.03	4.58	3.39
2^{14}	52.10	52.20	3.50	2.99	3.88	3.25	7.35	6.07	7.83	6.59	4.33	3.05	4.76	3.38
2^{15}	45.70	46.20	3.62	2.99	3.95	3.27	7.70	6.39	8.11	6.83	4.43	3.06	4.81	3.44
2^{16}	39.30	39.30	3.72	3.05	4.03	3.31	7.57	6.30	8.09	6.61	4.52	3.10	4.92	3.45

Table 31: DS11, unordered_set<unordered_set<int>>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	121.00	125.00	25.90	23.70	26.10	23.90	56.30	54.50	56.70	54.70	27.40	25.80	27.80	26.00
2^{7}	141.00	145.00	26.40	24.30	25.60	23.40	62.10	59.60	62.50	60.00	27.90	25.80	28.30	25.80
2^{8}	165.00	173.00	31.50	27.30	32.20	27.70	77.40	73.70	77.80	74.20	34.80	31.90	35.60	32.20
2^{9}	171.00	178.00	35.90	27.60	34.40	27.80	80.00	73.70	79.70	74.60	35.70	32.00	36.50	31.90
2^{10}	177.00	182.00	38.70	28.60	35.60	27.90	81.10	74.30	81.30	74.30	36.70	31.80	37.10	32.00
2^{11}	177.00	183.00	38.20	27.60	36.20	27.70	81.30	74.30	82.20	74.10	37.00	32.00	37.80	31.90
2^{12}	179.00	186.00	39.10	27.70	36.50	28.00	81.60	73.50	81.50	74.10	37.30	31.80	37.90	32.10
2^{13}	165.00	169.00	39.00	27.80	36.70	27.80	81.30	73.90	82.80	73.50	37.30	32.10	38.30	32.10
2^{14}	153.00	156.00	40.90	29.60	38.70	29.60	81.50	74.10	82.40	73.70	44.40	39.20	45.40	39.10
2^{15}	122.00	131.00	47.60	35.70	44.80	36.10	85.70	75.20	83.90	75.40	51.00	45.10	51.40	45.50
2^{16}	100.00	111.00	51.40	40.40	48.00	38.80	85.10	75.50	86.20	75.60	53.60	48.40	54.60	48.20

Table 32: DS12, unordered_set<unordered_set<string>>

Appendix 2: Elided results from Benchmark I

This section contains the full results of Benchmark I, generated for this paper. These results were elided from the body of the paper for space constraints, and are included here for completeness.

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	5.91	6.67	3.05	2.71	3.12	2.83	5.16	4.77	5.23	4.70	4.17	3.82	4.24	3.83
2^{7}	7.55	8.36	3.05	2.64	3.05	2.76	4.65	4.12	4.41	4.05	3.87	3.51	3.82	3.49
2^{8}	7.63	8.45	2.93	2.60	3.02	2.74	4.06	3.70	4.00	3.64	3.70	3.32	3.64	3.36
2^{9}	7.65	8.53	2.94	2.58	3.00	2.71	3.85	3.48	3.78	3.43	3.62	3.25	3.60	3.20
2^{10}	7.66	8.47	2.90	2.58	2.99	2.71	3.79	3.39	3.76	3.33	3.62	3.21	3.59	3.16
2^{11}	7.54	8.44	2.93	2.57	2.99	2.70	3.79	3.35	3.74	3.27	3.64	3.18	3.59	3.13
2^{12}	7.55	8.45	2.95	2.57	3.00	2.70	5.26	4.74	5.21	4.63	3.66	3.19	3.62	3.13
2^{13}	8.56	9.41	2.97	2.57	3.01	2.71	6.55	5.48	6.47	5.38	3.72	3.20	3.66	3.15
2^{14}	8.85	9.67	3.01	2.57	3.05	2.71	6.90	5.94	6.85	5.89	3.82	3.21	3.77	3.19
2^{15}	9.00	9.83	2.99	2.57	3.03	2.71	5.98	4.67	5.92	4.61	3.83	3.21	3.78	3.19
2^{16}	8.97	9.88	2.99	2.57	3.03	2.71	5.64	4.90	5.57	4.82	3.84	3.21	3.79	3.18

Table 33: DS3, unordered_set<int>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	19.87	20.48	9.53	9.05	9.52	9.07	14.30	13.59	14.55	13.69	12.44	11.78	12.46	11.69
2^{7}	19.08	19.91	9.48	8.90	9.47	8.97	21.83	21.21	21.67	21.18	11.93	11.23	11.92	11.11
2^{8}	21.37	22.46	12.54	12.02	12.50	12.02	36.06	35.17	36.56	35.38	15.41	14.74	15.42	14.69
2^{9}	22.39	23.33	13.65	12.78	13.61	12.78	42.36	40.83	42.22	41.16	16.06	14.94	16.06	14.97
2^{10}	22.83	23.77	14.04	12.72	14.04	12.72	45.20	43.53	44.73	43.27	16.14	14.76	16.09	14.70
2^{11}	23.43	24.40	14.28	12.71	14.28	12.71	47.10	45.04	46.81	45.07	16.13	14.62	16.07	14.63
2^{12}	23.89	24.86	14.47	12.75	14.49	12.76	48.13	46.32	48.08	46.39	16.20	14.64	16.17	14.63
2^{13}	43.03	44.81	14.55	12.75	14.54	12.77	48.64	46.61	48.46	46.65	16.26	14.61	16.25	14.62
2^{14}	43.48	45.12	14.55	12.74	14.55	12.76	49.20	47.00	49.01	47.01	16.27	14.61	16.27	14.61
2^{15}	46.82	49.27	15.58	14.19	15.31	13.98	58.15	55.79	58.29	56.33	23.52	22.80	23.89	23.37
2^{16}	62.16	65.78	26.41	24.36	26.17	24.44	66.54	62.56	66.43	63.12	33.62	31.90	33.89	32.28

Table 34: DS4, unordered_set<string>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^6	0.34	0.35	0.32	0.31	0.29	0.29	0.33	0.32	0.33	0.33	0.32	0.32	0.32	0.32
2^7	0.34	0.34	0.31	0.31	0.30	0.30	0.32	0.32	0.32	0.32	0.32	0.31	0.32	0.32
2^8	0.44	0.45	0.31	0.31	0.30	0.30	0.37	0.36	0.37	0.37	0.32	0.31	0.32	0.31
2^9	0.52	0.53	0.31	0.31	0.30	0.30	0.44	0.43	0.44	0.44	0.32	0.32	0.32	0.32
2^{10}	0.56	0.57	0.31	0.31	0.30	0.30	0.48	0.47	0.48	0.47	0.32	0.32	0.32	0.31
2^{11}	0.58	0.59	0.31	0.31	0.30	0.30	0.50	0.49	0.50	0.49	0.32	0.32	0.32	0.32
2^{12}	0.59	0.50	0.31	0.31	0.30	0.30	0.51	0.50	0.51	0.50	0.32	0.32	0.32	0.32
2^{13}	0.48	0.50	0.31	0.31	0.30	0.30	0.48	0.47	0.48	0.47	0.32	0.32	0.32	0.32
2^{14}	0.48	0.49	0.31	0.31	0.30	0.30	0.47	0.46	0.47	0.46	0.32	0.32	0.32	0.32
2^{15}	0.50	0.51	0.31	0.31	0.30	0.30	0.50	0.49	0.52	0.49	0.33	0.32	0.32	0.32
2^{16}	0.56	0.59	0.33	0.33	0.32	0.32	0.57	0.55	0.57	0.55	0.35	0.34	0.34	0.34

Table 35: DS5, vector<vector<int>>

	← global	→	←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual	→			← virtual	→			← virtual	→
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^6	14.87	15.04	8.93	8.60	8.94	8.59	37.65	36.19	37.71	36.09	10.84	10.19	10.84	10.19
2^7	15.02	15.26	8.96	8.59	8.96	8.60	37.91	36.21	37.82	36.32	10.85	10.19	10.85	10.20
2^8	16.51	16.21	9.13	8.69	9.12	8.70	46.52	44.69	46.80	44.80	15.02	14.71	15.00	15.13
2^9	26.21	26.58	17.91	17.82	17.90	17.86	55.60	52.70	55.81	52.49	24.33	24.50	24.42	24.46
2^{10}	30.83	31.19	18.77	18.27	18.92	18.21	56.97	52.43	56.88	52.58	25.65	24.72	25.60	24.76
2^{11}	49.57	49.88	18.79	18.30	18.83	18.25	57.43	52.22	57.44	52.17	26.18	24.75	26.05	25.04
2^{12}	50.00	50.40	18.64	18.16	18.71	18.13	57.51	51.96	57.46	51.90	26.53	24.72	26.53	24.60
2^{13}	50.07	50.61	18.73	18.30	18.78	18.22	57.62	51.93	57.59	51.91	26.78	24.80	26.84	24.87
2^{14}	50.30	50.59	19.23	18.75	19.34	18.73	57.50	51.75	57.45	51.73	28.86	26.84	28.85	26.88
2^{15}	50.46	51.01	21.73	21.13	21.75	21.16	57.58	51.81	57.55	51.88	30.82	28.73	30.82	28.79
2^{16}	50.44	50.67	22.90	22.35	22.92	22.37	57.92	52.12	57.97	52.03	31.84	29.77	31.85	29.79

Table 36: DS6, vector<vector<string>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	7.62	8.36	3.03	2.67	3.11	2.76	5.92	5.17	5.87	5.13	3.88	3.31	3.84	3.27
2^{7}	7.64	8.38	3.10	2.68	3.18	2.76	6.23	5.48	6.16	5.42	3.96	3.30	3.92	3.26
2^{8}	7.65	8.39	3.16	2.68	3.24	2.76	6.30	5.54	6.22	5.47	3.99	3.31	3.94	3.25
2^{9}	7.65	8.40	3.19	2.67	3.27	2.75	6.41	5.66	6.34	5.59	4.00	3.30	3.97	3.26
2^{10}	7.63	8.40	3.19	2.67	3.27	2.76	6.44	5.67	6.38	5.60	4.01	3.30	3.97	3.26
2^{11}	9.06	9.88	3.20	2.67	3.28	2.75	6.54	5.73	6.40	5.74	4.02	3.30	3.98	3.26
2^{12}	9.37	10.54	3.21	2.67	3.29	2.76	7.39	6.58	7.35	6.50	4.22	3.41	4.20	3.36
2^{13}	10.81	11.68	3.56	2.72	3.66	2.82	8.18	6.89	8.07	6.82	4.87	3.50	4.84	3.46
2^{14}	11.25	12.19	3.87	2.73	3.95	2.83	8.63	6.94	8.54	6.84	5.15	3.55	5.13	3.50
2^{15}	11.42	12.39	4.03	2.74	4.12	2.84	8.80	6.93	8.71	6.85	5.35	3.59	5.29	3.54
2^{16}	11.50	12.44	4.12	2.76	4.21	2.87	8.80	6.90	8.73	6.81	5.47	3.65	5.42	3.60

Table 37: DS7, vector<unordered_set<int>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	22.78	23.68	13.18	11.38	13.15	11.36	48.54	46.87	48.80	46.64	15.66	13.99	15.67	13.98
2^{7}	22.99	23.85	13.15	11.34	13.14	11.31	49.17	47.01	49.23	47.17	15.70	13.98	15.71	13.99
2^{8}	24.82	25.77	13.28	11.65	13.27	11.43	60.34	58.27	60.66	58.26	20.69	19.48	21.11	19.86
2^{9}	36.80	39.14	23.19	21.10	23.18	21.01	69.03	65.18	69.12	65.10	31.17	29.14	31.10	29.17
2^{10}	42.60	44.97	26.08	21.33	26.19	21.33	70.60	64.98	70.71	65.05	33.47	29.40	33.42	29.50
2^{11}	64.39	66.34	27.72	21.30	27.72	21.56	71.18	64.60	71.06	64.60	34.38	29.48	34.35	29.52
2^{12}	65.17	67.16	29.06	21.72	28.94	21.38	67.73	60.94	67.83	61.18	35.24	29.65	35.41	30.20
2^{13}	66.32	68.88	29.64	21.44	28.82	21.45	67.85	60.58	67.64	60.84	35.63	29.61	35.41	29.73
2^{14}	65.35	67.37	30.21	22.47	30.60	22.60	67.97	60.60	67.78	60.76	38.58	32.19	38.52	32.60
2^{15}	66.37	67.60	32.72	24.68	32.56	24.58	68.11	60.97	68.18	61.01	40.72	34.57	40.50	34.41
2^{16}	66.25	67.50	33.77	25.84	34.24	25.87	67.86	60.71	68.22	60.87	41.60	35.61	41.88	35.50

Table 38: DS8, vector<unordered_set<string>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	0.38	0.39	0.32	0.31	0.32	0.32	0.35	0.34	0.35	0.34	0.34	0.33	0.34	0.33
2^{7}	0.40	0.40	0.32	0.31	0.32	0.32	0.34	0.33	0.34	0.33	0.33	0.33	0.33	0.33
2^{8}	0.40	0.40	0.32	0.31	0.32	0.32	0.41	0.40	0.41	0.40	0.33	0.33	0.33	0.33
2^{9}	0.40	0.41	0.32	0.31	0.32	0.32	0.48	0.47	0.48	0.47	0.33	0.33	0.33	0.33
2^{10}	0.40	0.41	0.33	0.31	0.33	0.32	0.52	0.50	0.52	0.51	0.34	0.33	0.34	0.33
2^{11}	0.41	0.41	0.33	0.31	0.33	0.32	0.54	0.53	0.54	0.53	0.34	0.33	0.34	0.33
2^{12}	0.41	0.41	0.33	0.31	0.33	0.32	0.55	0.53	0.55	0.54	0.34	0.33	0.34	0.33
2^{13}	0.57	0.58	0.33	0.31	0.34	0.32	0.55	0.54	0.55	0.54	0.34	0.33	0.34	0.33
2^{14}	0.57	0.58	0.33	0.31	0.34	0.32	0.56	0.54	0.56	0.54	0.34	0.33	0.34	0.33
2^{15}	0.59	0.60	0.33	0.31	0.34	0.32	0.54	0.53	0.55	0.53	0.34	0.33	0.34	0.33
2^{16}	0.70	0.72	0.37	0.34	0.37	0.34	0.63	0.60	0.63	0.60	0.37	0.35	0.37	0.35

Table 39: DS9, unordered_set<vector<int>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	15.14	15.40	8.96	8.60	8.95	8.58	37.85	36.20	37.85	36.10	10.85	10.19	10.86	10.19
2^{7}	14.98	15.28	8.96	8.59	8.97	8.60	38.01	36.24	38.15	36.31	10.84	10.19	10.84	10.18
2^{8}	16.07	16.56	9.08	8.64	9.07	8.66	46.27	44.97	47.00	44.66	14.72	14.54	14.71	14.91
2^{9}	27.03	27.65	18.10	17.97	18.24	18.14	56.21	52.79	56.22	52.89	24.96	25.22	24.85	25.11
2^{10}	32.76	32.96	18.90	18.56	19.13	18.76	57.58	52.83	57.80	52.92	26.14	25.13	26.53	25.21
2^{11}	34.48	34.59	18.81	18.08	18.72	18.22	57.67	52.36	57.72	52.50	26.99	25.24	26.74	25.22
2^{12}	36.11	36.29	19.25	18.60	19.22	18.53	57.83	52.32	57.90	52.14	26.97	25.19	27.14	25.24
2^{13}	50.77	51.10	18.82	18.20	18.86	18.38	57.93	52.06	58.12	52.19	27.56	25.61	27.44	25.22
2^{14}	50.62	51.48	19.64	19.06	19.60	19.04	58.03	52.15	57.99	52.09	29.30	27.07	29.19	27.24
2^{15}	51.42	51.47	21.83	21.24	21.93	21.31	57.89	52.04	58.08	52.17	31.20	28.85	31.15	28.91
2^{16}	51.05	51.19	23.19	22.57	23.09	22.40	58.26	52.24	58.17	52.35	32.09	29.92	32.16	29.95

Table 40: DS10, unordered_set<vector<string>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual			← virtual →				← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	7.48	8.43	3.11	2.72	3.16	2.78	6.06	5.27	5.97	5.20	3.94	3.30	3.89	3.24
2^{7}	7.59	8.49	3.26	2.72	3.32	2.78	6.25	5.57	6.17	5.42	4.00	3.29	3.94	3.23
2^{8}	7.57	8.50	3.27	2.72	3.33	2.78	6.37	5.61	6.31	5.54	4.00	3.29	3.94	3.23
2^{9}	7.60	8.50	3.27	2.72	3.33	2.78	6.44	5.64	6.39	5.57	4.01	3.29	3.96	3.23
2^{10}	7.62	8.51	3.27	2.72	3.33	2.78	6.47	5.68	6.38	5.62	4.01	3.29	3.96	3.23
2^{11}	7.62	8.54	3.28	2.72	3.34	2.79	6.52	5.75	6.44	5.65	4.02	3.29	3.96	3.23
2^{12}	7.79	8.78	3.28	2.72	3.34	2.79	7.68	6.70	7.61	6.60	4.25	3.40	4.25	3.34
2^{13}	10.87	11.97	3.79	2.77	3.89	2.84	8.66	7.04	8.62	6.98	5.14	3.51	5.04	3.44
2^{14}	11.30	12.41	4.27	2.82	4.35	2.90	8.96	7.10	8.86	7.01	5.33	3.56	5.27	3.49
2^{15}	11.38	12.48	4.37	2.85	4.45	2.92	9.04	7.06	8.99	6.99	5.48	3.61	5.41	3.55
2^{16}	11.44	12.52	4.43	2.89	4.51	2.97	9.04	7.05	8.97	6.99	5.56	3.68	5.49	3.62

Table 41: DS11, unordered_set<unordered_set<int>>

	← global →		←	monotonic		→	←	multipool		→	←	multi + mono		→
		virtual				← virtual →			← virtual →				← virtual →	
				(wink)		(wink)		(wink)		(wink)		(wink)		(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	22.6	23.6	13.1	11.3	13.2	11.4	48.9	46.7	48.8	46.8	15.8	14.1	15.8	14.1
2^{7}	23.1	23.8	13.2	11.3	13.2	11.4	49.2	47.2	49.2	47.4	15.8	14.0	15.8	14.1
2^{8}	24.8	26.3	13.3	11.3	13.2	11.7	60.1	58.1	60.2	58.7	20.4	19.6	21.2	19.5
2^{9}	38.0	39.7	24.1	21.1	23.6	21.4	70.0	65.2	70.0	65.2	32.3	29.9	32.4	29.3
2^{10}	44.0	45.9	27.7	21.8	27.8	21.7	71.3	65.2	71.7	65.4	34.3	29.9	34.6	30.0
2^{11}	46.5	49.1	29.7	22.1	29.4	22.0	71.8	64.9	71.9	64.9	36.1	30.2	36.0	30.3
2^{12}	46.9	48.5	28.9	21.7	29.3	21.6	71.7	64.5	71.9	64.6	36.6	30.3	36.4	30.1
2^{13}	66.2	68.3	29.6	21.6	29.9	21.9	72.1	65.1	73.1	65.3	36.7	30.3	36.7	30.7
2^{14}	66.9	68.0	30.5	22.6	30.5	22.5	72.5	65.1	72.8	65.1	39.0	32.4	39.6	32.7
2^{15}	67.0	68.5	32.6	24.7	32.8	24.8	68.9	61.6	69.3	61.7	41.2	34.7	41.5	34.7
2^{16}	66.1	67.8	34.0	26.0	34.1	26.0	68.9	61.3	69.0	61.4	42.1	35.7	42.4	35.7

Table 42: DS12, unordered_set<unordered_set<string>>

Appendix 3: Absolute Run Times for Benchmark II

This section contains the absolute run times for Benchmark II, generated for this paper. The analysis of these results is included in the body of the paper in Chapter 10: “Benchmark II: Variation in Locality (Long Running).” These raw numbers are included here for completeness.

List Length (`S`)	Access Factor (`af`)									Number of Lists (`k`)
	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
2^{21}	21.02	21.24	21.29	21.15	21.03	21.28	21.10	21.05	21.14	2^0
2^{20}	21.08	20.58	21.09	21.21	20.95	20.79	21.36	21.08	20.92	2^1
2^{19}	17.74	16.79	17.45	18.25	16.06	18.65	18.00	19.95	21.89	2^2
2^{18}	15.35	15.57	15.42	15.74	15.76	16.28	17.13	18.49	21.95	2^3
2^{17}	15.29	15.38	15.34	15.52	15.63	16.10	16.99	18.39	21.90	2^4
2^{16}	14.99	15.04	15.06	15.18	15.37	15.80	16.70	18.28	21.88	2^5
2^{15}	14.96	15.02	15.05	15.17	15.36	15.81	16.59	18.28	22.06	2^6
2^{14}	14.96	15.00	15.04	15.16	15.35	15.80	16.63	18.23	21.90	2^7
2^{13}	12.62	12.59	12.85	12.95	13.09	13.87	14.94	17.17	21.88	2^8
2^{12}	11.88	11.94	12.06	12.23	12.54	13.18	14.35	16.91	21.71	2^9
2^{11}	11.93	11.98	12.10	12.27	12.61	13.19	14.46	16.82	21.95	2^{10}
2^{10}	9.93	10.00	10.12	10.30	10.71	11.44	12.90	15.94	21.90	2^{11}
2^9	9.89	9.95	10.09	10.27	10.64	11.44	12.86	15.87	21.69	2^{12}
2^8	10.02	10.06	10.18	10.38	10.74	11.50	12.91	15.72	21.93	2^{13}
2^7	10.00	10.05	10.15	10.34	10.67	11.34	12.77	15.35	21.78	2^{14}
2^6	10.22	10.29	10.37	10.56	10.93	11.55	13.00	15.53	22.15	2^{15}
2^5	8.74	8.79	8.88	8.98	9.22	9.73	10.67	13.28	23.93	2^{16}
2^4	7.91	7.95	7.99	8.09	8.28	8.60	9.56	13.14	21.93	2^{17}
2^3	7.10	7.13	7.19	7.32	7.56	8.09	9.25	14.22	22.45	2^{18}
2^2	7.57	7.55	7.63	7.84	8.07	8.20	9.61	15.11	24.50	2^{19}
2^1	8.29	8.35	8.56	8.86	9.32	8.96	10.79	16.40	29.07	2^{20}
2^0	11.65	11.57	11.47	11.72	12.50	12.22	13.77	21.49	40.06	2^{21}

Table 43: Absolute run times for Benchmark II, problem size 2^{21}, without shuffling step, using global allocator

	Access Factor (af)									
List Length (S)	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Number of Lists (k)
2^{21}	21.2	21.2	21.1	21.1	21.3	21.1	21.3	21.2	21.2	2^0
2^{20}	174.8	176.1	175.6	175.1	175.7	175.4	174.8	175.5	175.6	2^1
2^{19}	346.4	348.1	347.6	350.3	351.0	355.0	360.6	366.4	378.3	2^2
2^{18}	165.1	165.6	178.6	182.9	194.6	220.7	263.0	316.4	390.0	2^3
2^{17}	133.6	131.5	137.1	150.3	160.8	189.3	218.4	292.2	396.2	2^4
2^{16}	95.7	99.0	101.4	105.7	117.8	139.0	182.9	260.2	397.7	2^5
2^{15}	98.8	99.9	101.7	104.8	115.9	132.2	171.0	247.1	395.9	2^6
2^{14}	99.5	100.2	101.9	104.6	115.5	136.0	168.8	243.9	396.9	2^7
2^{13}	99.4	100.2	101.5	104.7	116.2	135.0	169.8	244.3	397.0	2^8
2^{12}	77.0	79.2	81.0	84.0	96.9	110.4	155.5	231.4	395.6	2^9
2^{11}	48.5	49.6	58.4	62.7	70.4	89.2	133.1	222.4	398.3	2^{10}
2^{10}	45.0	48.4	50.5	54.2	67.5	87.7	132.9	219.5	400.5	2^{11}
2^9	33.0	38.3	39.7	45.0	56.0	78.2	125.3	214.0	402.1	2^{12}
2^8	20.7	26.0	24.8	31.7	40.9	66.8	116.0	211.5	399.9	2^{13}
2^7	18.6	19.9	21.8	29.8	40.4	62.3	110.7	205.9	392.5	2^{14}
2^6	15.1	17.2	20.5	26.4	38.0	63.0	111.8	203.4	398.8	2^{15}
2^5	12.4	13.9	17.1	24.7	38.8	63.3	114.8	209.2	390.5	2^{16}
2^4	10.7	12.8	16.8	24.3	37.3	65.3	115.6	218.0	393.9	2^{17}
2^3	10.0	12.3	16.2	24.6	40.7	67.8	126.0	233.8	401.2	2^{18}
2^2	10.1	13.2	16.7	26.3	43.2	75.1	137.8	250.7	392.6	2^{19}
2^1	11.6	14.1	19.2	28.7	45.3	78.4	132.8	215.8	310.0	2^{20}
2^0	14.7	17.5	22.7	33.5	52.8	78.8	115.8	175.5	275.2	2^{21}

Table 44: Absolute run times for Benchmark II, problem size 2^{21}, with shuffling step, using global allocator

	Access Factor (af)									
List Length (S)	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	Number of Lists (k)
2^{21}	32.3	32.2	32.5	32.4	32.3	32.3	32.2	32.3	32.2	2^0
2^{20}	32.2	32.4	32.3	32.3	32.3	32.2	32.3	32.4	32.1	2^1
2^{19}	31.2	30.4	29.6	29.4	30.4	29.7	30.3	31.9	32.4	2^2
2^{18}	22.8	22.4	22.4	23.3	23.6	24.0	25.0	27.8	32.5	2^3
2^{17}	21.7	21.9	21.9	22.0	22.4	23.0	24.4	27.1	32.3	2^4
2^{16}	21.7	21.7	21.8	22.0	22.3	23.0	24.3	27.0	32.5	2^5
2^{15}	21.3	21.3	21.4	21.6	21.9	22.6	24.0	26.8	32.2	2^6
2^{14}	21.3	21.3	21.4	21.6	21.9	22.6	24.0	26.9	32.2	2^7
2^{13}	21.3	21.3	21.4	21.5	21.9	22.6	23.9	26.7	32.6	2^8
2^{12}	14.8	14.9	15.1	15.3	15.9	17.0	19.3	23.9	32.7	2^9
2^{11}	14.8	15.0	15.2	15.5	16.1	17.2	19.6	24.0	33.1	2^{10}
2^{10}	15.0	15.2	15.3	15.7	16.2	17.5	19.8	24.4	34.1	2^{11}
2^9	10.0	10.1	10.4	10.8	11.5	13.2	16.2	22.9	35.5	2^{12}
2^8	10.4	10.5	10.7	11.2	12.1	13.8	17.2	24.2	38.8	2^{13}
2^7	10.1	10.3	10.6	11.1	12.2	14.4	18.8	27.6	45.3	2^{14}
2^6	10.5	10.7	11.2	11.8	13.4	16.2	22.2	34.0	58.0	2^{15}
2^5	8.9	9.2	9.7	10.6	12.9	16.7	25.5	41.3	80.5	2^{16}
2^4	8.3	8.7	9.4	10.7	13.9	19.4	31.5	59.0	105.4	2^{17}
2^3	7.4	8.0	9.1	10.9	15.3	22.7	37.6	70.4	120.5	2^{18}
2^2	7.8	8.5	9.7	12.0	16.4	27.8	53.8	81.6	184.9	2^{19}
2^1	9.0	10.2	12.2	15.6	23.4	39.4	66.5	107.7	297.5	2^{20}
2^0	13.3	14.9	18.4	24.5	36.2	54.9	80.7	129.6	345.7	2^{21}

Table 45: Absolute run times for Benchmark II, problem size 2^{21}, without shuffling step, using multipool allocator

List Length (`S`)	Access Factor (`af`)									Number of Lists (`k`)
	2^8	2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	
2^{21}	32.2	32.5	32.4	32.3	32.3	32.4	32.2	32.1	32.5	2^0
2^{20}	51.2	51.3	50.7	50.8	51.2	51.0	51.3	51.0	51.5	2^1
2^{19}	47.3	49.0	51.1	50.2	49.0	51.2	53.5	53.6	57.7	2^2
2^{18}	35.9	35.5	35.6	37.0	38.0	39.0	43.2	49.7	60.5	2^3
2^{17}	32.2	32.4	32.6	33.1	34.0	35.9	39.7	47.4	62.1	2^4
2^{16}	32.2	32.4	32.6	33.0	34.0	35.9	39.9	48.1	63.9	2^5
2^{15}	31.8	31.9	32.2	32.7	33.8	35.8	40.0	48.7	65.8	2^6
2^{14}	31.8	31.9	32.1	32.7	33.8	36.0	40.5	49.5	67.5	2^7
2^{13}	31.8	32.0	32.3	32.9	34.1	36.6	41.5	51.4	71.1	2^8
2^{12}	24.5	24.7	25.1	26.0	27.6	31.0	37.9	51.3	77.7	2^9
2^{11}	24.2	24.5	25.2	26.3	28.4	32.4	40.8	57.1	90.6	2^{10}
2^{10}	23.7	24.0	24.7	26.0	28.4	33.1	43.1	61.6	99.6	2^{11}
2^9	10.7	11.1	11.9	13.5	16.8	23.1	35.9	61.6	111.9	2^{12}
2^8	10.6	11.1	12.0	13.9	17.3	24.6	39.3	67.8	124.5	2^{13}
2^7	10.6	11.1	12.2	14.2	18.5	26.6	42.9	76.2	141.2	2^{14}
2^6	10.9	11.6	12.7	15.3	19.8	29.4	48.5	86.7	162.0	2^{15}
2^5	10.5	11.3	12.7	15.7	21.6	32.9	55.8	100.9	187.3	2^{16}
2^4	9.5	10.3	12.1	15.2	22.2	36.4	61.8	112.5	211.4	2^{17}
2^3	8.8	10.0	12.3	16.6	25.6	44.3	80.6	149.6	274.7	2^{18}
2^2	9.4	11.1	14.4	21.5	34.5	61.5	112.6	211.9	353.5	2^{19}
2^1	11.3	13.8	18.5	28.1	46.5	79.8	138.1	231.0	364.9	2^{20}
2^0	15.5	19.1	26.3	41.0	69.0	101.4	140.1	213.2	367.6	2^{21}

Table 46: Absolute run times for Benchmark II, problem size 2^{21}, with shuffling step, using multipool allocator

Appendix 4: Absolute run times for Benchmark III

This section contains the absolute run times for Benchmark III, generated for this paper. The analysis of these results is included in the body of the paper in Chapter 11 “Benchmark III: Variation in Utilization.” These raw numbers are included here for completeness.

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{30}	2^{15}	2^{10}	0.052	0.055	0.226	0.226	0.023	0.024	0.024	0.024
2^{30}	2^{16}	2^{10}	0.052	0.055	0.226	0.226	0.024	0.024	0.024	0.024
2^{30}	2^{17}	2^{10}	0.052	0.054	0.226	0.226	0.024	0.024	0.024	0.024
2^{30}	2^{18}	2^{10}	0.052	0.054	0.226	0.226	0.024	0.024	0.024	0.024
2^{30}	2^{19}	2^{10}	0.052	0.054	0.226	0.226	0.024	0.024	0.024	0.024
2^{30}	2^{20}	2^{10}	0.052	0.054	0.226	0.226	0.024	0.024	0.024	0.025
2^{30}	2^{20}	2^{11}	0.026	0.027	0.217	0.218	0.012	0.012	0.012	0.013
2^{30}	2^{20}	2^{12}	0.013	0.013	0.213	0.213	0.007	0.007	0.006	0.006
2^{30}	2^{20}	2^{13}	0.006	0.007	0.211	0.211	0.008	0.008	0.213	0.213
2^{30}	2^{20}	2^{14}	0.003	0.004	0.209	0.209	0.004	0.004	0.210	0.210
2^{30}	2^{20}	2^{15}	0.002	0.002	0.208	0.208	0.002	0.002	0.209	0.209

Table 47: Total Allocated Memory, T = 2^{30}, absolute run times in seconds

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{31}	2^{15}	2^{10}	0.104	0.109	0.453	0.453	0.046	0.048	0.048	0.048
2^{31}	2^{16}	2^{10}	0.104	0.109	0.452	0.453	0.048	0.048	0.047	0.048
2^{31}	2^{17}	2^{10}	0.103	0.108	0.452	0.452	0.047	0.048	0.048	0.048
2^{31}	2^{18}	2^{10}	0.103	0.108	0.452	0.452	0.048	0.048	0.048	0.048
2^{31}	2^{19}	2^{10}	0.103	0.108	0.452	0.452	0.048	0.048	0.048	0.049
2^{31}	2^{20}	2^{10}	0.103	0.108	0.452	0.452	0.048	0.048	0.048	0.049
2^{31}	2^{20}	2^{11}	0.053	0.054	0.435	0.435	0.024	0.025	0.024	0.025
2^{31}	2^{20}	2^{12}	0.025	0.026	0.426	0.427	0.014	0.014	0.012	0.012
2^{31}	2^{20}	2^{13}	0.013	0.013	0.421	0.421	0.015	0.015	0.425	0.425
2^{31}	2^{20}	2^{14}	0.007	0.007	0.418	0.419	0.008	0.008	0.420	0.420
2^{31}	2^{20}	2^{15}	0.003	0.004	0.417	0.417	0.004	0.004	0.418	0.418

Table 48: Total Allocated Memory, `T` = 2^{31}, absolute run times in seconds

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{32}	2^{15}	2^{10}	0.208	0.218	0.905	0.906	0.092	0.096	0.095	0.096
2^{32}	2^{16}	2^{10}	0.208	0.217	0.905	0.905	0.095	0.096	0.095	0.096
2^{32}	2^{17}	2^{10}	0.207	0.216	0.904	0.905	0.095	0.097	0.095	0.096
2^{32}	2^{18}	2^{10}	0.206	0.216	0.904	0.904	0.095	0.096	0.095	0.096
2^{32}	2^{19}	2^{10}	0.206	0.216	0.905	0.905	0.096	0.096	0.096	0.097
2^{32}	2^{20}	2^{10}	0.206	0.216	0.904	0.905	0.096	0.097	0.096	0.097
2^{32}	2^{20}	2^{11}	0.105	0.109	0.870	0.870	0.049	0.049	0.048	0.049
2^{32}	2^{20}	2^{12}	0.050	0.053	0.853	0.853	0.026	0.026	0.024	0.025
2^{32}	2^{20}	2^{13}	0.025	0.026	0.842	0.842	0.030	0.030	0.850	0.850
2^{32}	2^{20}	2^{14}	0.013	0.014	0.837	0.837	0.016	0.016	0.840	0.840
2^{32}	2^{20}	2^{15}	0.007	0.007	0.835	0.834	0.008	0.008	0.836	0.836

Table 49: Total Allocated Memory, `T` = 2^{32}, absolute run times in seconds

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{33}	2^{15}	2^{10}	0.417	0.436	1.813	1.812	0.185	0.192	0.190	0.193
2^{33}	2^{16}	2^{10}	0.415	0.435	1.810	1.811	0.190	0.190	0.196	0.192
2^{33}	2^{17}	2^{10}	0.412	0.432	1.809	1.810	0.190	0.190	0.190	0.192
2^{33}	2^{18}	2^{10}	0.411	0.431	1.810	1.810	0.190	0.191	0.191	0.192
2^{33}	2^{19}	2^{10}	0.411	0.431	1.808	1.810	0.191	0.192	0.191	0.196
2^{33}	2^{20}	2^{10}	0.412	0.432	1.809	1.811	0.191	0.194	0.192	0.195
2^{33}	2^{20}	2^{11}	0.209	0.217	1.740	1.741	0.096	0.098	0.096	0.098
2^{33}	2^{20}	2^{12}	0.101	0.105	1.707	1.706	0.056	0.053	0.048	0.049
2^{33}	2^{20}	2^{13}	0.050	0.053	1.685	1.684	0.059	0.060	1.700	1.699
2^{33}	2^{20}	2^{14}	0.027	0.028	1.674	1.674	0.031	0.031	1.680	1.680
2^{33}	2^{20}	2^{15}	0.014	0.014	1.669	1.670	0.016	0.016	1.673	1.671

Table 50: Total Allocated Memory, `T` = 2^{33}, absolute run times in seconds

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{34}	2^{15}	2^{10}	0.833	0.873	3.622	3.623	0.370	0.384	0.381	0.386
2^{34}	2^{16}	2^{10}	0.831	0.870	3.619	3.623	0.381	0.381	0.379	0.384
2^{34}	2^{17}	2^{10}	0.824	0.863	3.618	3.622	0.379	0.382	0.380	0.383
2^{34}	2^{18}	2^{10}	0.822	0.864	3.618	3.618	0.379	0.383	0.379	0.383
2^{34}	2^{19}	2^{10}	0.822	0.862	3.615	3.618	0.384	0.385	0.382	0.387
2^{34}	2^{20}	2^{10}	0.824	0.863	3.616	3.618	0.382	0.386	0.383	0.388
2^{34}	2^{20}	2^{11}	0.419	0.434	3.481	3.482	0.192	0.195	0.192	0.196
2^{34}	2^{20}	2^{12}	0.201	0.210	3.412	3.412	0.105	0.106	0.095	0.097
2^{34}	2^{20}	2^{13}	0.100	0.105	3.368	3.368	0.118	0.120	3.396	3.397
2^{34}	2^{20}	2^{14}	0.053	0.056	3.346	3.347	0.062	0.063	3.362	3.363
2^{34}	2^{20}	2^{15}	0.027	0.028	3.336	3.336	0.032	0.032	3.344	3.345

Table 51: Total Allocated Memory, `T` = 2^{34}, absolute run times in seconds

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{35}	2^{15}	2^{10}	1.670	1.748	7.240	7.245	0.752	0.771	0.769	0.776
2^{35}	2^{16}	2^{10}	1.662	1.740	7.242	7.244	0.761	0.762	0.759	0.769
2^{35}	2^{17}	2^{10}	1.647	1.727	7.233	7.242	0.758	0.763	0.764	0.766
2^{35}	2^{18}	2^{10}	1.645	1.728	7.232	7.245	0.758	0.765	0.761	0.771
2^{35}	2^{19}	2^{10}	1.645	1.725	7.236	7.240	0.767	0.775	0.769	0.773
2^{35}	2^{20}	2^{10}	1.647	1.728	7.236	7.241	0.768	0.774	0.775	0.775
2^{35}	2^{20}	2^{11}	0.837	0.867	6.958	6.963	0.384	0.389	0.385	0.391
2^{35}	2^{20}	2^{12}	0.401	0.421	6.827	6.830	0.208	0.213	0.193	0.195
2^{35}	2^{20}	2^{13}	0.200	0.211	6.741	6.738	0.236	0.240	6.799	6.801
2^{35}	2^{20}	2^{14}	0.106	0.111	6.695	6.697	0.124	0.126	6.718	6.714
2^{35}	2^{20}	2^{15}	0.054	0.056	6.672	6.673	0.062	0.063	6.684	6.682

Table 52: Total Allocated Memory, T = 2^{35}, absolute run times in seconds

Appendix 5: Elided results for Benchmark III

This section contains the full analyzed results for Benchmark III, generated for this paper. Some of these results are presented in the body of the paper in Chapter 11 “Benchmark III: Variation in Utilization.” These numbers were elided for space reasons, and are included here for completeness.

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{31}	2^{15}	2^{10}	0.1042	105	434	435	44	46	46	46
2^{31}	2^{16}	2^{10}	0.1039	105	435	436	46	46	46	46
2^{31}	2^{17}	2^{10}	0.1030	105	439	439	46	46	46	47
2^{31}	2^{18}	2^{10}	0.1029	105	439	440	46	47	46	47
2^{31}	2^{19}	2^{10}	0.1029	105	439	439	46	47	47	47
2^{31}	2^{20}	2^{10}	0.1032	105	438	438	47	47	47	47
2^{31}	2^{20}	2^{11}	0.0528	103	824	824	46	47	46	47
2^{31}	2^{20}	2^{12}	0.0253	104	1682	1684	54	56	48	49
2^{31}	2^{20}	2^{13}	0.0128	104	3299	3300	117	118	3331	3330
2^{31}	2^{20}	2^{14}	0.0067	105	6201	6210	117	118	6229	6224
2^{31}	2^{20}	2^{15}	0.0034	103	12121	12123	115	116	12147	12148

Table 53: Total Allocated Memory, T = 2^{31}

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{32}	2^{15}	2^{10}	0.2083	105	435	435	44	46	46	46
2^{32}	2^{16}	2^{10}	0.2078	105	435	436	46	46	46	46
2^{32}	2^{17}	2^{10}	0.2070	105	437	437	46	47	46	46
2^{32}	2^{18}	2^{10}	0.2060	105	439	439	46	47	46	47
2^{32}	2^{19}	2^{10}	0.2057	105	440	440	47	47	47	47
2^{32}	2^{20}	2^{10}	0.2063	105	438	439	46	47	47	47
2^{32}	2^{20}	2^{11}	0.1050	103	829	829	46	47	46	47
2^{32}	2^{20}	2^{12}	0.0504	104	1692	1693	52	52	48	49
2^{32}	2^{20}	2^{13}	0.0253	104	3332	3333	118	119	3362	3363
2^{32}	2^{20}	2^{14}	0.0134	104	6249	6248	117	118	6272	6274
2^{32}	2^{20}	2^{15}	0.0068	103	12285	12269	115	117	12295	12292

Table 54: Total Allocated Memory, T = 2^{32}

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{33}	2^{15}	2^{10}	0.4166	105	435	435	44	46	46	46
2^{33}	2^{16}	2^{10}	0.4155	105	436	436	46	46	47	46
2^{33}	2^{17}	2^{10}	0.4117	105	439	440	46	46	46	47
2^{33}	2^{18}	2^{10}	0.4113	105	440	440	46	46	46	47
2^{33}	2^{19}	2^{10}	0.4112	105	440	440	46	47	47	48
2^{33}	2^{20}	2^{10}	0.4122	105	439	439	46	47	47	47
2^{33}	2^{20}	2^{11}	0.2093	104	831	832	46	47	46	47
2^{33}	2^{20}	2^{12}	0.1005	104	1698	1697	56	53	47	49
2^{33}	2^{20}	2^{13}	0.0502	105	3353	3353	118	119	3383	3383
2^{33}	2^{20}	2^{14}	0.0267	105	6276	6276	117	118	6299	6299
2^{33}	2^{20}	2^{15}	0.0136	103	12302	12311	115	117	12331	12319

Table 55: Total Allocated Memory, T = 2^{33}

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{34}	2^{15}	2^{10}	0.8330	105	435	435	44	46	46	46
2^{34}	2^{16}	2^{10}	0.8308	105	436	436	46	46	46	46
2^{34}	2^{17}	2^{10}	0.8236	105	439	440	46	46	46	47
2^{34}	2^{18}	2^{10}	0.8223	105	440	440	46	47	46	47
2^{34}	2^{19}	2^{10}	0.8221	105	440	440	47	47	46	47
2^{34}	2^{20}	2^{10}	0.8238	105	439	439	46	47	47	47
2^{34}	2^{20}	2^{11}	0.4185	104	832	832	46	47	46	47
2^{34}	2^{20}	2^{12}	0.2009	104	1698	1698	52	53	47	48
2^{34}	2^{20}	2^{13}	0.1003	105	3358	3358	118	119	3386	3387
2^{34}	2^{20}	2^{14}	0.0532	105	6287	6288	117	118	6317	6317
2^{34}	2^{20}	2^{15}	0.0271	103	12324	12325	116	118	12354	12357

Table 56: Total Allocated Memory, T = 2^{34}

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{35}	2^{15}	2^{10}	1.6700	105	434	434	45	46	46	46
2^{35}	2^{16}	2^{10}	1.6615	105	436	436	46	46	46	46
2^{35}	2^{17}	2^{10}	1.6471	105	439	440	46	46	46	47
2^{35}	2^{18}	2^{10}	1.6453	105	440	440	46	47	46	47
2^{35}	2^{19}	2^{10}	1.6448	105	440	440	47	47	47	47
2^{35}	2^{20}	2^{10}	1.6474	105	439	440	47	47	47	47
2^{35}	2^{20}	2^{11}	0.8373	104	831	832	46	47	46	47
2^{35}	2^{20}	2^{12}	0.4013	105	1701	1702	52	53	48	49
2^{35}	2^{20}	2^{13}	0.2005	105	3363	3361	118	120	3392	3392
2^{35}	2^{20}	2^{14}	0.1064	105	6293	6295	117	118	6314	6311
2^{35}	2^{20}	2^{15}	0.0541	104	12344	12346	115	117	12366	12362

Table 57: Total Allocated, T = 2^{35}

Appendix 6: Absolute Run Times for Benchmark IV

This section contains the absolute run times for Benchmark IV, generated for this paper. The analysis of these results is included in the body of the paper in Chapter12: “Benchmark IV: Variation in Contention.” These raw numbers are included here for completeness.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{15}$	2^6	1	0.103	0.105	0.026	0.025	0.049	0.049	0.049	0.049
$100*2^{15}$	2^6	2	0.103	0.105	0.051	0.026	0.049	0.049	0.049	0.049
$100*2^{15}$	2^6	3	0.103	0.105	0.026	0.026	0.049	0.049	0.049	0.049
$100*2^{15}$	2^6	4	0.103	0.105	0.026	0.026	0.078	0.049	0.049	0.049
$100*2^{15}$	2^6	5	0.181	0.190	0.050	0.052	0.080	0.081	0.079	0.082
$100*2^{15}$	2^6	6	0.181	0.190	0.051	0.051	0.079	0.079	0.082	0.080
$100*2^{15}$	2^6	7	0.186	0.191	0.059	0.055	0.081	0.082	0.079	0.081
$100*2^{15}$	2^6	8	0.192	0.192	0.058	0.055	0.082	0.096	0.089	0.084

Table 58: Absolute run times in seconds for number of iterations N = $100*2^{15}$, size of allocation S = 2^6, number of threads W varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{15}$	2^7	1	0.208	0.210	0.027	0.026	0.049	0.049	0.049	0.049
$100*2^{15}$	2^7	2	0.208	0.210	0.027	0.026	0.049	0.049	0.049	0.049
$100*2^{15}$	2^7	3	0.209	0.210	0.027	0.026	0.049	0.049	0.049	0.049
$100*2^{15}$	2^7	4	0.209	0.210	0.028	0.051	0.080	0.049	0.049	0.049
$100*2^{15}$	2^7	5	0.461	0.379	0.060	0.054	0.079	0.080	0.080	0.078
$100*2^{15}$	2^7	6	0.467	0.477	0.059	0.058	0.085	0.082	0.081	0.079
$100*2^{15}$	2^7	7	0.470	0.480	0.077	0.076	0.083	0.083	0.080	0.083
$100*2^{15}$	2^7	8	0.477	0.480	0.106	0.092	0.082	0.080	0.083	0.082

Table 59: Absolute run times in seconds for number of iterations N = $100*2^{15}$, size of allocation S = 2^7, number of threads W varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{15}$	2^8	1	0.209	0.210	0.030	0.029	0.049	0.049	0.049	0.049
$100*2^{15}$	2^8	2	0.209	0.210	0.030	0.029	0.049	0.049	0.049	0.049
$100*2^{15}$	2^8	3	0.213	0.217	0.064	0.045	0.053	0.053	0.081	0.049
$100*2^{15}$	2^8	4	0.209	0.210	0.101	0.102	0.049	0.049	0.080	0.080
$100*2^{15}$	2^8	5	0.461	0.476	0.129	0.147	0.080	0.080	0.083	0.080
$100*2^{15}$	2^8	6	0.461	0.476	0.166	0.166	0.080	0.080	0.079	0.080
$100*2^{15}$	2^8	7	0.461	0.478	0.182	0.182	0.080	0.079	0.086	0.085
$100*2^{15}$	2^8	8	0.461	0.477	0.199	0.198	0.097	0.082	0.080	0.080

Table 60: Absolute run times in seconds for number of iterations N = $100*2^{15}$, size of allocation S = 2^8, number of threads W varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{16}$	2^8	1	0.417	0.419	0.059	0.056	0.097	0.098	0.098	0.097
$100*2^{16}$	2^8	2	0.417	0.419	0.181	0.180	0.097	0.098	0.102	0.097
$100*2^{16}$	2^8	3	0.921	0.420	0.202	0.201	0.157	0.157	0.100	0.105
$100*2^{16}$	2^8	4	0.921	0.957	0.299	0.216	0.097	0.098	0.098	0.157
$100*2^{16}$	2^8	5	0.920	0.752	0.312	0.317	0.162	0.157	0.157	0.157
$100*2^{16}$	2^8	6	0.921	0.953	0.336	0.338	0.168	0.161	0.161	0.157
$100*2^{16}$	2^8	7	0.921	0.962	0.366	0.372	0.162	0.172	0.172	0.164
$100*2^{16}$	2^8	8	0.928	0.961	0.397	0.402	0.162	0.162	0.159	0.158

Table 61: Absolute run times in seconds for number of iterations N = $100*2^{16}$, size of allocation S = 2^8, number of threads W varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{17}$	2^8	1	0.834	0.838	0.338	0.342	0.194	0.196	0.196	0.194
$100*2^{17}$	2^8	2	1.843	0.838	0.378	0.375	0.194	0.196	0.196	0.194
$100*2^{17}$	2^8	3	0.835	0.838	0.405	0.403	0.194	0.196	0.196	0.194
$100*2^{17}$	2^8	4	0.836	0.839	0.434	0.435	0.194	0.196	0.196	0.194
$100*2^{17}$	2^8	5	1.843	1.902	0.631	0.635	0.316	0.316	0.320	0.318
$100*2^{17}$	2^8	6	1.471	1.907	0.672	0.635	0.317	0.321	0.318	0.323
$100*2^{17}$	2^8	7	1.843	1.901	0.720	0.718	0.318	0.321	0.324	0.321
$100*2^{17}$	2^8	8	1.841	1.905	0.791	0.790	0.320	0.323	0.325	0.318

Table 62: Absolute run times in seconds for number of iterations `N` = $100*2^{17}$, size of allocation `S` = 2^8, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{18}$	2^8	1	1.666	1.675	0.694	0.685	0.388	0.391	0.391	0.388
$100*2^{18}$	2^8	2	1.668	1.676	0.742	0.750	0.388	0.391	0.391	0.387
$100*2^{18}$	2^8	3	1.669	1.676	0.801	0.799	0.388	0.391	0.391	0.388
$100*2^{18}$	2^8	4	1.670	1.678	1.177	0.856	0.388	0.391	0.391	0.388
$100*2^{18}$	2^8	5	3.684	2.796	1.252	1.250	0.630	0.625	0.547	0.625
$100*2^{18}$	2^8	6	3.682	3.593	1.270	1.349	0.631	0.628	0.629	0.625
$100*2^{18}$	2^8	7	3.692	3.805	1.439	1.436	0.632	0.629	0.640	0.633
$100*2^{18}$	2^8	8	3.686	3.817	1.575	1.576	0.630	0.632	0.632	0.632

Table 63: Absolute run times in seconds for number of iterations `N` = $100*2^{18}$, size of allocation `S` = 2^8, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{19}$	2^8	1	3.333	3.351	1.380	1.377	0.775	0.781	0.781	0.774
$100*2^{19}$	2^8	2	3.335	3.351	1.498	1.482	0.775	0.782	0.782	0.775
$100*2^{19}$	2^8	3	3.337	3.840	1.599	1.596	0.775	0.782	0.782	0.775
$100*2^{19}$	2^8	4	3.337	3.353	1.700	1.704	0.775	0.782	0.782	0.775
$100*2^{19}$	2^8	5	6.276	5.159	2.439	2.497	1.262	1.254	1.253	1.254
$100*2^{19}$	2^8	6	5.584	7.500	2.658	2.617	1.258	1.144	1.253	1.254
$100*2^{19}$	2^8	7	6.595	7.613	2.852	2.861	1.260	1.258	1.264	1.250
$100*2^{19}$	2^8	8	7.374	7.618	3.151	3.150	1.261	1.265	1.258	1.262

Table 64: Absolute run times in seconds for number of iterations `N` = $100*2^{19}$, size of allocation `S` = 2^8, number of threads `W` varied.

Appendix 7: Absolute Run Times for Benchmark IV, with static buffer removed

This section contains the absolute run times for Benchmark IV, after re-running the benchmark with the static buffer removed. The analysis of these results is included in the body of the paper in Chapter 12: "Benchmark IV: Variation in Contention." These raw numbers are included here for completeness.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{15}$	2^6	1	0.103	0.105	0.076	0.076	0.049	0.049	0.049	0.049
$100*2^{15}$	2^6	2	0.104	0.105	0.079	0.078	0.049	0.049	0.049	0.049
$100*2^{15}$	2^6	3	0.103	0.105	0.080	0.080	0.049	0.049	0.049	0.049
$100*2^{15}$	2^6	4	0.104	0.105	0.081	0.081	0.049	0.049	0.049	0.049
$100*2^{15}$	2^6	5	0.180	0.190	0.121	0.123	0.078	0.079	0.080	0.072
$100*2^{15}$	2^6	6	0.192	0.191	0.123	0.125	0.079	0.080	0.081	0.080
$100*2^{15}$	2^6	7	0.182	0.191	0.127	0.134	0.082	0.082	0.080	0.082
$100*2^{15}$	2^6	8	0.188	0.196	0.140	0.136	0.092	0.093	0.083	0.087

Table 65: Absolute run times in seconds for number of iterations `N` = $100*2^{15}$, size of allocation `S` = 2^6, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{15}$	2^7	1	0.208	0.210	0.133	0.132	0.049	0.049	0.049	0.049
$100*2^{15}$	2^7	2	0.209	0.210	0.142	0.141	0.079	0.053	0.049	0.049
$100*2^{15}$	2^7	3	0.209	0.210	0.146	0.146	0.049	0.049	0.049	0.049
$100*2^{15}$	2^7	4	0.209	0.210	0.150	0.150	0.049	0.049	0.049	0.081
$100*2^{15}$	2^7	5	0.461	0.476	0.215	0.213	0.086	0.080	0.079	0.083
$100*2^{15}$	2^7	6	0.464	0.476	0.223	0.227	0.081	0.079	0.082	0.085
$100*2^{15}$	2^7	7	0.467	0.486	0.243	0.241	0.080	0.083	0.091	0.096
$100*2^{15}$	2^7	8	0.464	0.479	0.256	0.251	0.081	0.081	0.083	0.080

Table 66: Absolute run times in seconds for number of iterations `N` = $100*2^{15}$, size of allocation `S` = 2^7, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{15}$	2^8	1	0.208	0.210	0.203	0.201	0.049	0.049	0.049	0.049
$100*2^{15}$	2^8	2	0.209	0.210	0.222	0.224	0.049	0.049	0.079	0.049
$100*2^{15}$	2^8	3	0.209	0.211	0.245	0.241	0.049	0.049	0.049	0.049
$100*2^{15}$	2^8	4	0.210	0.210	0.268	0.263	0.049	0.082	0.049	0.049
$100*2^{15}$	2^8	5	0.461	0.476	0.400	0.379	0.085	0.082	0.079	0.079
$100*2^{15}$	2^8	6	0.462	0.477	0.421	0.415	0.080	0.081	0.083	0.083
$100*2^{15}$	2^8	7	0.462	0.476	0.462	0.463	0.080	0.081	0.083	0.084
$100*2^{15}$	2^8	8	0.463	0.478	0.491	0.491	0.079	0.083	0.093	0.088

Table 67: Absolute run times in seconds for number of iterations `N` = $100*2^{15}$, size of allocation `S` = 2^8, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{16}$	2^8	1	0.417	0.419	0.323	0.320	0.097	0.098	0.097	0.097
$100*2^{16}$	2^8	2	0.418	0.422	0.423	0.425	0.097	0.098	0.097	0.097
$100*2^{16}$	2^8	3	0.417	0.419	0.509	0.513	0.159	0.098	0.097	0.097
$100*2^{16}$	2^8	4	0.418	0.420	0.571	0.557	0.097	0.098	0.097	0.097
$100*2^{16}$	2^8	5	0.922	0.637	0.801	0.830	0.158	0.157	0.158	0.158
$100*2^{16}$	2^8	6	0.921	0.953	0.855	0.856	0.159	0.165	0.159	0.171
$100*2^{16}$	2^8	7	0.921	0.949	0.897	0.892	0.160	0.158	0.159	0.158
$100*2^{16}$	2^8	8	0.921	0.954	0.929	0.919	0.167	0.165	0.162	0.168

Table 68: Absolute run times in seconds for number of iterations `N` = $100*2^{16}$, size of allocation `S` = 2^8, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{17}$	2^8	1	0.834	0.838	0.793	0.793	0.194	0.196	0.194	0.194
$100*2^{17}$	2^8	2	0.833	0.838	0.986	0.945	0.194	0.196	0.194	0.194
$100*2^{17}$	2^8	3	0.835	1.908	0.999	0.993	0.194	0.196	0.194	0.194
$100*2^{17}$	2^8	4	0.836	0.840	1.031	1.042	0.315	0.196	0.194	0.194
$100*2^{17}$	2^8	5	1.842	1.391	1.538	1.515	0.318	0.316	0.317	0.314
$100*2^{17}$	2^8	6	1.840	1.584	1.580	1.559	0.318	0.319	0.317	0.318
$100*2^{17}$	2^8	7	1.846	1.905	1.631	1.618	0.325	0.316	0.320	0.316
$100*2^{17}$	2^8	8	1.842	1.913	1.686	1.673	0.326	0.320	0.319	0.320

Table 69: Absolute run times in seconds for number of iterations `N` = $100*2^{17}$, size of allocation `S` = 2^8, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{18}$	2^8	1	1.668	1.676	1.822	1.821	0.388	0.391	0.388	0.388
$100*2^{18}$	2^8	2	1.669	1.677	1.872	1.870	0.388	0.391	0.388	0.388
$100*2^{18}$	2^8	3	1.670	1.677	1.909	1.905	0.388	0.392	0.388	0.388
$100*2^{18}$	2^8	4	1.669	1.677	2.052	2.877	0.628	0.392	0.388	0.388
$100*2^{18}$	2^8	5	3.685	2.842	2.595	2.779	0.634	0.628	0.627	0.543
$100*2^{18}$	2^8	6	3.682	3.810	2.912	2.887	0.633	0.629	0.630	0.615
$100*2^{18}$	2^8	7	3.682	3.809	3.003	2.966	0.631	0.636	0.640	0.630
$100*2^{18}$	2^8	8	3.701	3.824	3.055	3.022	0.637	0.637	0.636	0.640

Table 70: Absolute run times in seconds for number of iterations `N` = $100*2^{18}$, size of allocation `S` = 2^8, number of threads `W` varied.

			←global→		←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
$100*2^{19}$	2^8	1	3.331	3.350	3.404	3.406	0.775	0.782	0.775	0.775
$100*2^{19}$	2^8	2	3.335	3.354	3.484	3.484	0.775	0.782	0.775	0.775
$100*2^{19}$	2^8	3	3.334	3.355	3.548	3.548	1.256	1.250	0.779	0.775
$100*2^{19}$	2^8	4	3.337	3.355	3.592	3.603	0.775	0.783	0.775	0.775
$100*2^{19}$	2^8	5	5.163	6.050	4.809	4.800	1.259	1.254	1.257	1.256
$100*2^{19}$	2^8	6	5.914	7.623	5.594	5.595	1.231	1.259	1.253	1.257
$100*2^{19}$	2^8	7	7.123	7.430	5.672	5.649	1.253	1.257	1.254	1.261
$100*2^{19}$	2^8	8	7.381	7.630	5.804	5.802	1.261	1.255	1.256	1.269

Table 71: Absolute run times in seconds for number of iterations `N` = $100*2^{19}$, size of allocation `S` = 2^8, number of threads `W` varied.