Doc No: N4468
Date: 2015-04-11
Authors: John Lakos (jlakos@bloomberg.net)
Jeffrey Mendelsohn (jmendelsohn4@bloomberg.net)
Alisdair Meredith (ameredith1@bloomberg.net)
Nathan Myers (nmyers12@bloomberg.net)

On Quantifying Memory-Allocation Strategies

Abstract

Performance requirements drive many of our difficult design choices. Memory management is an area where such choices can have surprising and far-reaching effects. Although performance of global allocators has improved markedly in recent years, use of local memory allocators can still provide substantial runtime (and other) benefits. The key to the effective use of memory allocators is knowing if and when to use which allocator and why.

To be able to make reasoned recommendations regarding the use of local memory allocators, we must first understand where and how they can affect runtime performance. We have identified several ways to characterize how systems can challenge a global allocator, and how they may benefit by applying a well-chosen local allocator. In order to develop optimal criteria for how to choose where and how to apply a local allocator, we need to obtain objective measurements. We have identified several usage patterns, which we have encoded into benchmarks to identify precisely where local allocators do (and do not) provide substantial benefits. This paper presents our preliminary "raw" initial quantitative results (with relatively little analysis) in the hope of stimulating informed discussion.

Implementations of standard allocators (and others) are freely available today – along with usage examples – in Bloomberg's open-source distribution of the BDE library at <https://github.com/bloomberg/bde>. Benchmark code and results including those discussed in this paper can be found at <https://github.com/bloomberg/bde/benchmarks/allocators>.

Contents

On Quantifying Memory-Allocation Strategies 1
1 Introduction 2
2 Use an Allocator? Which One? 2
3 Available Concrete Allocators: Monotonic and Multipool 3
4 Our Tool Chest of Allocation Strategies 4
5 Characterizing Memory-Allocator Usage Scenarios 6
5.1 Density of allocation operations 7

5.2 Variation in allocated memory sizes 7
5.3 Locality facilitating memory access/manipulation 7
5.4 Utilization of allocated memory 8
5.5 Contention due to concurrent memory allocations 8
5.6 DVLUC the Duck! 8
6 Designing Useful Benchmarks 9
7 Benchmark I: Creating/Destroying Isolated Basic Data Structures. 10
8 Benchmark II: Variation in Locality (long running) 17
9 Benchmark III: Variation in Utilization 26
10 Benchmark IV: Variation in Contention 30
11 Conclusion 33
12 References 34

1 Introduction

Serious engineers appreciate C++ for enabling them to write code at a low level when needed. Resource management is an important aspect of low-level control – particularly memory management.

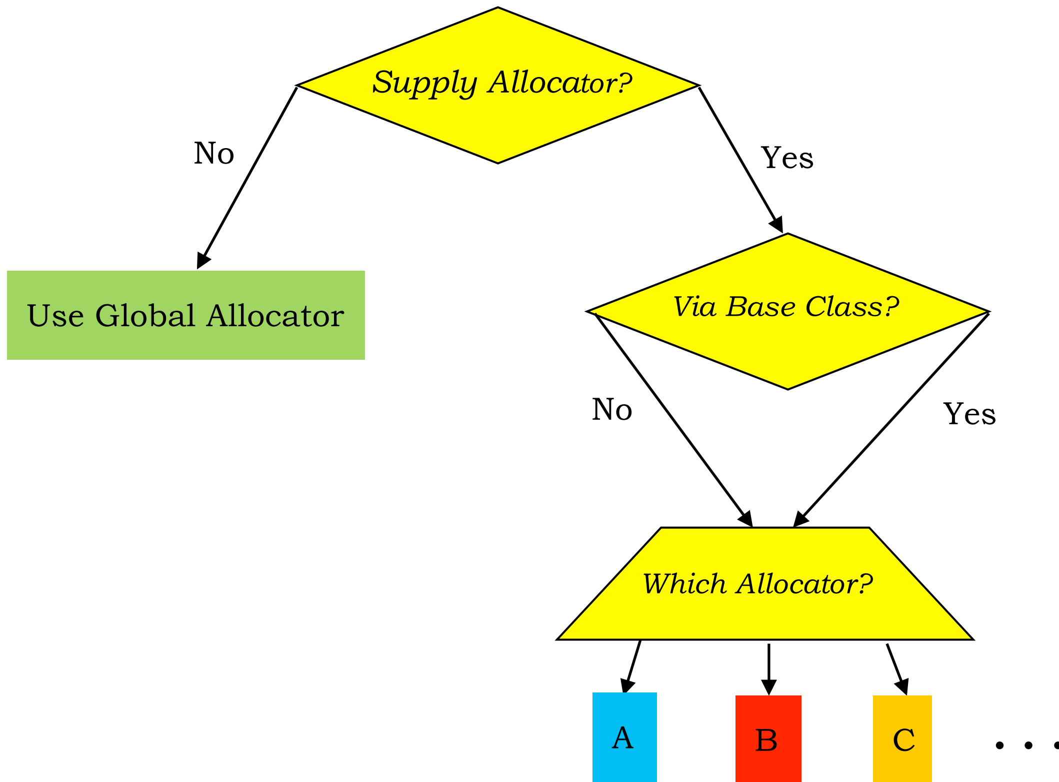
Should we instrument the standard library for such fine-tuning? The arguments against are typically that fine-grained memory management requires more up-front design effort, complicates interfaces, and may actually degrade performance where no allocator (or an ill-chosen one) is supplied. These are valid concerns that can be addressed only by having well-supported facts; by employing careful measurement, we can identify precisely how much performance benefit is available.

A library instrumented to exploit local allocators enables other benefits: allocators can aid testing, debugging, and measurement. Not all memory is alike – some is faster for certain processors, some is shared, some may be protected, and we need allocators to use those effectively.

2 Use an Allocator? Which One?

Before exploring allocator performance metrics, we should identify what we hope to learn. We need help deciding, first, whether injecting a local allocator will help or hurt performance. If an allocator won't help, we should use the system-wide (default) global allocator.

If an allocator would be helpful, we would then need to determine whether one should be "baked in" as a type parameter at compile time (e.g., with the intent of squeezing out the last bit of runtime performance) or passed as an abstract base class (thereby enabling enhanced interoperability for non-template types). Either



way, we then need to choose the allocator or allocators to use. The rest of this paper addresses quantitatively the runtime consequences of these choices.

3 Available Concrete Allocators: Monotonic and Multipool

In this paper, we have selected two allocators from the Fundamentals TS, which it refers to as "monotonic" and "multipool".

A monotonic allocator supplies memory from a contiguous block sequentially until the block is exhausted, after which it dynamically allocates a new block of geometrically increasing size, typically from the global allocator. Returning memory to a monotonic allocator is a no-op; the returned memory remains unused until the monotonic-allocator object is destroyed.

A multipool allocator is quite different: It consists of an array of pools, one for each geometrically increasing request-size range, up to some specified maximum. Each time memory is requested, the memory is provided from the appropriate pool. If the pool is empty, a geometrically increasing block is requested, up to some implementation-defined threshold, from the backing allocator, typically the global allocator. Blocks that exceed the maximum pool size pass through to the backing allocator directly. The combination of a multipool allocator backed by a monotonic allocator forms the third allocator candidate (C) that we consider in this paper.

Both monotonic and multipool allocators are *managed allocators*. A managed allocator is an allocator that, in addition to having `allocate` and `deallocate` methods, also has a `release` method, used to summarily return all 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 common managed-allocator instance, and managing no non-memory resource themselves, we can avoid running the objects' destructors. Instead, we can "wink them out" *en mass* by releasing the memory they occupy, along with all the memory they manage, via the allocator's `release` method.

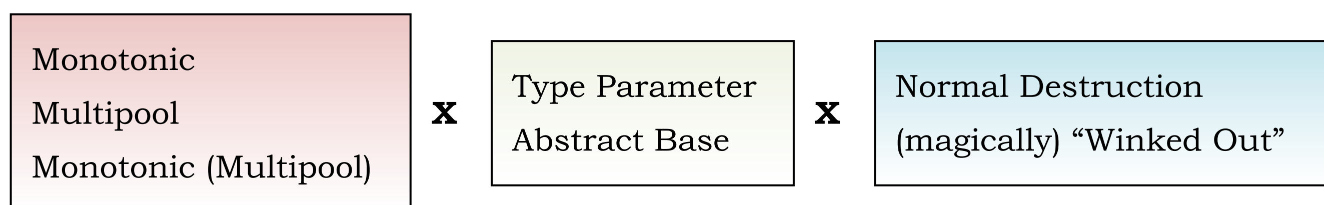
The runtime benefits of bypassing individual destruction of each element in a container can be significant, as deallocating memory is sometimes even more costly than allocating it. Note that this "winking out" technique requires `new`-ing the container object itself into the managed allocator it is to use, so that (1) its destructor is not called, and (2) its footprint is also released when the allocator goes out of scope.

4 Our Tool Chest of Allocation Strategies

Before we start considering interesting benchmarks, we need to consider the available allocation strategies. Each memory-usage pattern will have different properties, and therefore we can reasonably expect different allocation strategies to excel.

In this paper, we will consider up to 14 different allocation strategies for each of the benchmarks we subsequently present. The first of these strategies will be the default global allocator (bound at compile time) which will form the baseline for each successive comparison. (Supplying the default at compile-time produces the same object code, and so we have omitted that as a separate category.) The second case is the `new_delete` allocator supplied via an abstract base class, which (for compilers that do not yet elide runtime dispatch where they clearly could) can be used to compare that additional runtime overhead.

The remaining 12 allocation strategies can best be described by the following cross product:



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 passed via and 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.

AS1 Default Global Allocator (bound at compile time)

AS2 New/Delete Allocator (bound at runtime)

AS3 Monotonic, Type Parameter, Normal Destruction

AS4 Monotonic, Type Parameter, (magically) “Winked Out”

AS5 Monotonic, Abstract Base, Normal Destruction

AS6 Monotonic, Abstract Base, (magically) “Winked Out”

AS7 Multipool, Type Parameter, Normal Destruction

AS8 Multipool, Type Parameter, (magically) “Winked Out”

AS9 Multipool, Abstract Base, Normal Destruction

AS10 Multipool, Abstract Base, (magically) “Winked Out”

AS11 Monotonic(Multipool), Type Parameter, Normal Destruction

AS12 Monotonic(Multipool), Type Parameter, (magically) “Winked Out”

AS13 Monotonic(Multipool), Abstract Base, Normal Destruction

AS14 Monotonic(Multipool), Abstract Base, (magically) “Winked Out”

In each and every case, exactly one of these fourteen allocation strategies will be the best answer from a purely runtime-performance perspective. It is what it is.

Note that, for the purposes of this paper, we have deliberately left the definitions of the `allocate` and `deallocate` methods of all local allocators “out of line” so as to ensure that the added runtime cost of invoking a (virtual) function is observable; subsequently, `inline`-ing all such functions produced a hefty speedup in practice – e.g., ~33% for Benchmark III (see section 9).

5 Characterizing Memory-Allocator Usage Scenarios

Knowing when to supply an allocator and which one to use is neither obvious nor is it typically taught in school at any level. Rather, if and how to use memory allocators effectively is something that is learned only from repeated real-world experience. In this paper, however, we attempt to begin to elucidate some of the important considerations that experts consider when evaluating whether or not to take local control over memory management.

The first step in characterizing a problem is to normalize it to basic size parameters. Problems of vastly different sizes are not comparable, so we want to try to avoid that. After some consideration, we decided that problem size could be roughly characterized in terms of two parameters:

N the number of **instructions** executed

W the number of active **threads**

The relationship between the number of instructions executed and the number of active threads is not clear, and the value of trying to come up with a single number that combines the two does not seem to be useful. Clearly the number of available processors, the size of L1 cache, and a host of other machine-specific physical parameters will affect the detailed analysis. For the scope of this paper, however, we will limit ourselves to characterizing the logical program independently of any physical hardware on which it might be run.

Given this overall "size" characterization (**N**, **W**), we now introduce five dimensions that span the space of memory-allocator usage:

D Density of allocation operations

V Variation in allocated memory sizes

L Locality facilitating memory access/manipulation

U Utilization of allocated memory

C Contention due to concurrent memory allocations

Each of these dimensions resides on a scale from 0 to 1, where 0 indicates the low-end of the scale, and 1 the high end. Note that none of these scales is (necessarily) linear. It is also important to realize that each of these dimensions applies not to the overall program, but instead to just an individual targeted subsystem over some relevant duration of program execution. That is, when considering these dimensions, we are looking to improve the performance of a particular individual subsystem over a finite duration of execution, rather than that of the program as a whole.

5.1 *Density of allocation operations*

The allocation **density** is a measure of the relative number of allocation instructions (`allocate` and `deallocate`) to the total number of instructions executed. A density of 0 would imply that no allocation operations are employed, while a density of 1 would indicate that every operation involves either allocation or deallocation. As an example, a `std::vector<int>` is incapable of achieving a meaningfully high allocation density as the number of allocation operations are at most logarithmic in the number of mutating operations, and we sometimes even do a `reserve` on vectors, thereby reducing the number of allocators for this data structure to just 1 (e.g., Benchmark I, see section 7). By contrast, a vector of (long) strings could be used in a way that admits a relatively high allocation density, as each mutating operation would involve allocation or deallocation of the string-element's memory. Node-based containers (that don't do internal pooling) are similarly capable of achieving a high allocation density. Even with a potentially high density for mutating operations, the overall density will depend on the proportion of mutating to non-mutating (i.e., accessing or other non-allocation/deallocation-realted) operations.

5.2 *Variation in allocated memory sizes*

The **variation** in allocated memory sizes attempts to roughly measure the extent to which allocated memory requests vary over the region and duration of interest. A variation of 0 would mean that only a single memory size is allocated, while a variation of 1 would suggest a much more uniform (or perhaps hyperbolic) distribution of memory sizes. A relatively low value might tend to suggest a pool-based allocator, whereas a higher value might favor a coalescing allocator. Keep in mind that requests that are relatively close in size might be treated equivalently.

5.3 *Locality facilitating memory access/manipulation*

The definition of access **locality** is complex, involving at least three factors:

I the number of **instructions** executed in the subsystem over the duration

M the size of the **memory** footprint of the subsystem

T the number of context **transitions** out of the subsystem during the duration

The locality, **L**, correlates directly to the number of instructions, **I**, but inversely to the memory footprint, **M**, and the number of transitions, **T**. We can therefore argue that access locality, **L**, can be characterized (to a zeroth-order approximation) as:

$$L = \frac{I}{M * T}$$

In other words, the more instructions that flow through our subsystem, the more access locality we have. On the other hand, the bigger our subsystem's footprint or

the more context transitions that occur away from it, the lower the access locality becomes. Note that access locality will turn out to be dominant in some long-running programs – even when the allocation density is negligible (e.g., Benchmark II, see section 8).

5.4 *Utilization of allocated memory*

Allocated memory **utilization** is a measure of the relative amount of allocated memory that remains in use at any one time; it is defined as the maximum amount of memory that is ever in use by a subsystem at one time during the durations of interest divided by the total about of memory allocated by the subsystem over that period. A utilization of 1 means that, at some point, all of the memory every allocated by a subsystem over the duration of interest is actively in use. A utilization that approaches 0 suggests a long-running system in which the same memory is allocated and deallocated repeatedly. Subsystems exhibiting high utilization are typically good candidates for monotonic allocators, while long-running systems having low utilization are more suited for multipool allocators, or (perhaps even better) a multipool allocator backed by a monotonic one.

5.5 *Contention due to concurrent memory allocations*

Allocation **contention** is a measure of the potential bottlenecks that could result from multiple threads attempting to access the same synchronized memory allocator. We define allocation contention as the expected number of concurrent memory allocation operations in any given instant of time, over the duration of interest, divided by the number of active threads, **W**. A contention, **C**, of 0 suggests that **W** is 1 (or the allocation density, **D**, for all but one thread is 0). A contention of 1 would mean that **W** > 1 and each thread is always trying to allocate memory (i.e., **D** per thread is 1). Many modern global memory allocators are "thread aware" and make heroic efforts to mitigate such contention. In doing so, however, they can slow down subsystems in situations that do not require synchronization, while – falling short of expert handling – slow down those in situations that do. Note that, because of the strong correlation between dimensions **C** and **D**, it will turn out to be difficult to observe variations in **C** independently of **D** (e.g., Benchmark IV, see section 10).

5.6 *DVLUC the Duck!*

Remember Rule 6. What is Rule 6, you ask. Rule 6 is, "Don't take yourself so #$%^& seriously!"

DVLUC

D = Density of allocation operations

V = Variation in allocated memory sizes

L = Locality facilitating memory access/manipulation

U = Utilization of allocated memory

C = Contention due to concurrent memory allocations

Remembering these five dimensions of memory-allocation usage is a challenge for anyone, including the folks who identified them, so we decided to create a mnemonic aid by way of a mascot: The mascot is a duck, and his name is **DVLUC**. Deal with it.

6 Designing Useful Benchmarks

After identifying the dimensions of allocation space to explore, we wanted to come up with suitable benchmarks to inform as to how each of these dimensions affected our design decisions. Our first thought was to come up with a single benchmark that spanned all of the dimensions – the idea being to find the centroid, and then vary the arguments along each dimension separately in to discover its effect on the best allocator-strategy choice.

As it turns out, coming up with a single problem that encompasses all five of the dimensions identified above is not at all easy, as some dimensions are strongly correlated with others (e.g., Contention, **C**, and Density, **D**). Instead, we settled on four separate benchmarks, which together seem to cover this five-dimensional space and enable each of the fourteen proposed allocation strategies their fair shot.

Separately, we tried not to assume the answers we expected, and hence strove to cover the entire design space without prejudice. Hence, in our benchmarks we typically explore a wide range of problem sizes using successive powers of two. To better understand secondary effects, we will often choose to trade off comparable parameters, such as the subsystem size versus the number of subsystems (physical locality) or the number of consecutive accesses of a subsystem verses the number of subsystems visited (temporal locality) while holding other benchmark parameters constant.

7 Benchmark I: Creating/Destroying Isolated Basic Data Structures.

In this experiment, we look at the process of creating a variety of isolated complex data structures, using them lightly, and then quickly destroying them. The set of data structures under test encompass many of those we use every day, and were chosen specifically to illustrate thoroughly the first couple of dimensions discussed earlier (section 5). Each standard container under consideration (std::vector and std::unordered_set) will ultimately consist of elements of either int or std::string (where the string length in characters, chosen randomly between 33 and 1000 (uniform distribution), is deliberately outside the range where the short-string optimization pertains).

Twelve representative standard-library data structures were chosen – the second and third sets of four being, respectively, an std::vector and std::unordered_set of elements corresponding to those of the first:

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

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>

The runtime results for executing each of these benchmark tests using each of the 12 data structures above, employing each of the 14 allocation strategies discussed in section 4, for a wide variety of problem sizes (section 6), on an Intel i7-4770 @

3.4GHz with 11GB RAM available, are presented below. Numbers in brackets are run time (in seconds); other numbers (on the same row) are percentages relative to the corresponding run time. For a program that takes 10 seconds using the default operators `new` and `delete` (**AS1**), "50%" indicates a runtime of 5 seconds. Rows numbered 4 to 16 indicate the $\log_2$ of the size of the data structure constructed – e.g., for row 8, the outermost data structure is built up to have 2^8 = 256 elements before being destroyed.

This benchmark focuses, primarily, on the dimensions of density (**D**), variability (**V**), discussed in section 5. The relatively short-lived nature of the objects in this benchmark – along with their extremely high allocation Utilization (**U**) – facilitate measuring the benefit of allocations strategies, such as AS4, AS6, AS8, AS10, AS12, and AS14, that "wink-out" object memory.

Note that each vector in this benchmark is explicitly pre-sized (using `reserve`) to have exactly the needed capacity; hence, measurements for `vector<int>` (DS1), involving only a single memory allocation, are just noise.

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[0.004s]	1	4	5	4	377	9	1	1	1	3	3	3	3
5	[0.004s]	1	4	5	3	350	1	1	1	1	2	2	2	2
6	[0.002s]	1	3	4	3	296	1	1	1	1	2	2	2	2
7	[0.003s]	1	2	2	2	208	1	1	1	1	1	1	1	1
8	[0.002s]	1	1	2	1	162	1	1	1	1	1	1	1	1
9	[0.002s]	1	1	1	1	130	1	1	1	1	1	1	1	1
10	[0.002s]	1	1	1	1	117	1	1	1	1	1	1	1	1
11	[0.002s]	1	1	1	1	105	1	1	1	1	1	1	1	1
12	[0.002s]	1	1	1	1	117	1	1	1	1	1	1	1	1
13	[0.002s]	1	1	1	1	97	1	1	1	1	1	1	1	1
14	[0.002s]	1	1	1	1	107	1	1	1	1	1	1	1	1
15	[0.002s]	1	1	1	1	128	1	1	1	1	1	1	1	1
16	[0.002s]	1	1	1	1	106	1	1	1	1	1	1	1	1
DS1	**vector<int>**													

For DS2, `vector<string>`, we insert 2^n strings of randomly distributed size (in the range [33..1000] bytes), then destroy the vector and repeat for a total of 2^{27-n} times.

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[24.6s]	101	109	111	108	111	68	69	70	69	90	85	88	85
5	[23.4s]	95	103	102	103	103	72	71	73	72	89	86	88	85
6	[22s]	101	102	99	102	101	78	77	79	78	91	88	91	89
7	[21.9s]	102	109	106	109	108	78	78	79	78	91	88	89	88
8	[22.5s]	101	169	164	169	166	78	78	80	78	89	87	89	86
9	[22.8s]	102	193	189	192	188	77	77	79	78	89	86	88	85
10	[23.2s]	101	202	196	202	196	75	76	78	76	87	84	87	84
11	[29.6s]	134	165	164	167	163	60	61	61	60	69	66	69	65
12	[40.5s]	102	128	121	127	122	44	44	45	44	51	48	50	48
13	[45.1s]	98	110	102	108	102	39	39	41	40	49	44	49	45
14	[53s]	89	97	87	96	87	34	35	36	35	48	39	47	39
15	[52s]	101	111	95	110	97	36	37	38	37	56	43	55	44
16	[55.4s]	100	105	89	104	90	36	36	38	36	57	41	56	41
DS2	**vector<string>**													

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[5.17s]	108	152	150	157	152	44	46	58	52	102	93	104	96
5	[4.9s]	108	117	113	121	115	43	45	58	52	84	78	84	81
6	[4.79s]	108	93	89	95	90	44	46	57	51	75	69	77	72
7	[6.52s]	105	57	56	64	56	31	32	41	36	51	47	52	47
8	[6.59s]	103	52	50	53	50	31	32	41	36	49	44	49	45
9	[6.41s]	103	50	48	51	48	32	33	41	37	49	44	48	44
10	[6.33s]	106	51	46	52	47	31	33	42	37	50	44	49	44
11	[6.33s]	104	51	46	51	47	32	33	41	37	49	43	48	44
12	[6.49s]	102	77	71	76	72	31	32	41	36	49	43	48	43
13	[7.28s]	104	91	79	90	78	27	29	36	33	45	38	43	38
14	[7.48s]	104	94	86	94	85	27	28	34	31	44	37	42	37
15	[7.66s]	104	68	54	68	55	26	27	35	31	43	36	42	36
16	[7.98s]	104	65	57	65	57	25	27	34	30	43	35	42	36
DS3	**unordered_set<int>**													

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[55.5s]	99	92	91	92	91	66	67	68	66	79	76	78	76
5	[53.1s]	100	87	87	87	86	70	70	71	70	79	78	78	77
6	[52.4s]	99	83	83	83	81	72	71	72	72	78	77	78	77
7	[48.8s]	99	99	98	102	98	78	77	78	78	84	82	83	82
8	[50.2s]	98	127	125	126	124	78	77	78	78	84	82	83	81
9	[49.9s]	99	138	136	140	137	78	77	79	78	84	82	84	82
10	[51.4s]	99	141	141	141	138	77	76	80	76	83	80	82	80
11	[53.1s]	99	141	141	143	139	75	75	79	75	82	79	81	79
12	[55.8s]	98	142	138	139	136	73	73	76	73	80	77	79	76
13	[83.9s]	93	101	95	98	94	51	50	53	50	58	54	57	53
14	[85.6s]	103	109	99	107	97	54	56	59	55	72	62	70	62
15	[104s]	110	95	83	102	87	51	50	59	51	68	58	70	59
16	[126s]	104	93	80	93	79	49	49	61	50	70	56	72	56

DS4 unordered_set<string>

For the remaining tests, applied to nested data structures, the inner structure was chosen (arbitrarily) to get 2^7 =128 elements; the outer container gets 2^n elements, and will be constructed and destroyed 2^{20} times, for a total of 2^{27} insertions, as above. In this way, we make the total number of insert operations across data structures of different physical sizes comparable (section 6).

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[0.06s]	174	106	112	171	164	49	49	92	85	103	87	128	119
5	[0.05s]	218	99	97	152	151	54	54	98	94	82	77	138	125
6	[0.05s]	214	91	88	157	157	54	57	118	105	79	107	144	128
7	[0.05s]	223	80	99	387	383	69	58	127	100	82	75	130	134
8	[0.05s]	471	196	237	677	611	61	70	154	113	74	70	141	133
9	[0.13s]	224	152	143	354	332	25	26	51	49	31	29	66	59
10	[0.13s]	237	179	164	360	347	26	26	51	49	32	30	62	58
11	[0.16s]	210	102	105	246	210	22	21	41	41	28	24	54	48
12	[0.17s]	224	113	99	243	261	26	20	41	51	26	24	61	55
13	[0.17s]	271	109	99	270	243	22	21	46	60	31	25	81	78
14	[0.20s]	236	116	93	254	233	24	23	84	62	51	33	126	117
15	[0.22s]	247	115	98	262	255	38	31	101	83	56	42	158	122
16	[0.26s]	213	99	78	240	228	37	32	83	78	73	47	137	118

DS5 vector<vector<int>>

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[11.1s]	335	261	252	498	500	61	60	119	116	75	68	149	140
5	[10.7s]	354	254	240	556	539	57	57	118	116	71	64	162	145
6	[10.3s]	439	282	265	663	662	58	58	139	133	78	69	265	211
7	[12.1s]	454	270	246	632	615	55	56	173	167	104	77	322	275
8	[17.8s]	351	207	178	519	479	51	52	142	149	99	70	262	219
9	[20.6s]	306	179	152	393	361	49	48	122	119	90	60	212	176
10	[25.1s]	286	146	121	321	295	39	39	101	99	74	47	177	147
11	[33.2s]	217	112	91	242	223	29	30	76	74	58	36	136	111
12	[33.2s]	217	112	91	241	222	30	30	77	75	59	36	135	111
13	[33.1s]	217	110	89	240	220	30	30	78	75	59	37	149	125
14	[33s]	215	112	90	239	221	31	31	106	103	67	44	192	166
15	[32.8s]	214	111	91	239	219	43	43	123	122	87	64	211	194
16	[33.5s]	N/A	110	89	N/A	N/A	53	53	N/A	N/A	97	74	N/A	N/A

DS6 **vector<vector<string>>**

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[6.96s]	228	74	71	174	184	69	68	135	130	72	67	145	140
5	[7.95s]	194	72	68	156	158	58	58	116	109	60	55	123	116
6	[7.86s]	196	81	77	168	169	58	58	116	109	60	56	125	119
7	[7.9s]	192	82	78	171	169	57	57	115	109	59	55	125	119
8	[7.97s]	195	85	80	184	180	57	57	116	109	61	56	128	120
9	[7.81s]	198	85	81	184	179	58	58	117	108	61	55	130	120
10	[9.15s]	219	77	71	173	166	48	49	101	94	54	47	124	106
11	[9.48s]	216	78	71	171	162	48	48	104	96	57	46	126	105
12	[9.95s]	208	78	69	165	157	47	46	100	92	57	46	123	102
13	[9.78s]	207	78	69	162	154	47	47	101	94	58	46	125	103
14	[9.78s]	207	79	69	163	155	48	48	100	94	59	47	125	103
15	[10.1s]	207	78	69	162	153	48	47	101	94	60	47	126	103
16	[9.86s]	209	79	71	166	157	50	49	105	97	63	49	132	109

DS7 **vector<unordered_set<int>>**

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[6.96s]	228	74	71	174	184	69	68	135	130	72	67	145	140
5	[7.95s]	194	72	68	156	158	58	58	116	109	60	55	123	116
6	[7.86s]	196	81	77	168	169	58	58	116	109	60	56	125	119
7	[7.9s]	192	82	78	171	169	57	57	115	109	59	55	125	119
8	[7.97s]	195	85	80	184	180	57	57	116	109	61	56	128	120
9	[7.81s]	198	85	81	184	179	58	58	117	108	61	55	130	120
10	[9.15s]	219	77	71	173	166	48	49	101	94	54	47	124	106
11	[9.48s]	216	78	71	171	162	48	48	104	96	57	46	126	105
12	[9.95s]	208	78	69	165	157	47	46	100	92	57	46	123	102
13	[9.78s]	207	78	69	162	154	47	47	101	94	58	46	125	103
14	[9.78s]	207	79	69	163	155	48	48	100	94	59	47	125	103
15	[10.1s]	207	78	69	162	153	48	47	101	94	60	47	126	103
16	[9.86s]	209	79	71	166	157	50	49	105	97	63	49	132	109

DS8 **vector<unordered_set<int>>**

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[6.96s]	228	74	71	174	184	69	68	135	130	72	67	145	140
5	[7.95s]	194	72	68	156	158	58	58	116	109	60	55	123	116
6	[7.86s]	196	81	77	168	169	58	58	116	109	60	56	125	119
7	[7.9s]	192	82	78	171	169	57	57	115	109	59	55	125	119
8	[7.97s]	195	85	80	184	180	57	57	116	109	61	56	128	120
9	[7.81s]	198	85	81	184	179	58	58	117	108	61	55	130	120
10	[9.15s]	219	77	71	173	166	48	49	101	94	54	47	124	106
11	[9.48s]	216	78	71	171	162	48	48	104	96	57	46	126	105
12	[9.95s]	208	78	69	165	157	47	46	100	92	57	46	123	102
13	[9.78s]	207	78	69	162	154	47	47	101	94	58	46	125	103
14	[9.78s]	207	79	69	163	155	48	48	100	94	59	47	125	103
15	[10.1s]	207	78	69	162	153	48	47	101	94	60	47	126	103
16	[9.86s]	209	79	71	166	157	50	49	105	97	63	49	132	109

DS9 **vector<unordered_set<int>>**

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[6.96s]	228	74	71	174	184	69	68	135	130	72	67	145	140
5	[7.95s]	194	72	68	156	158	58	58	116	109	60	55	123	116
6	[7.86s]	196	81	77	168	169	58	58	116	109	60	56	125	119
7	[7.9s]	192	82	78	171	169	57	57	115	109	59	55	125	119
8	[7.97s]	195	85	80	184	180	57	57	116	109	61	56	128	120
9	[7.81s]	198	85	81	184	179	58	58	117	108	61	55	130	120
10	[9.15s]	219	77	71	173	166	48	49	101	94	54	47	124	106
11	[9.48s]	216	78	71	171	162	48	48	104	96	57	46	126	105
12	[9.95s]	208	78	69	165	157	47	46	100	92	57	46	123	102
13	[9.78s]	207	78	69	162	154	47	47	101	94	58	46	125	103
14	[9.78s]	207	79	69	163	155	48	48	100	94	59	47	125	103
15	[10.1s]	207	78	69	162	153	48	47	101	94	60	47	126	103
16	[9.86s]	209	79	71	166	157	50	49	105	97	63	49	132	109

DS10 vector<unordered_set<int>>

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[7.64s]	101	59	51	58	51	42	42	42	38	47	40	47	41
5	[7.77s]	101	65	58	64	58	39	39	39	36	43	37	43	37
6	[8.08s]	103	65	59	66	60	37	37	38	33	40	35	42	35
7	[7.28s]	117	72	64	73	66	43	42	42	38	45	38	47	39
8	[7.24s]	104	76	69	79	70	43	42	44	38	46	38	47	39
9	[7.23s]	102	84	72	86	74	43	42	43	38	46	37	47	38
10	[9.22s]	84	71	60	73	60	34	35	35	30	40	30	41	32
11	[10s]	103	72	57	72	57	35	35	39	31	46	30	46	31
12	[10.2s]	106	72	58	75	59	36	36	44	33	49	31	50	32
13	[10.2s]	104	72	57	73	57	36	36	43	33	49	31	50	32
14	[10.5s]	103	69	55	71	55	37	37	46	33	51	33	51	33
15	[12.7s]	116	68	53	68	53	45	43	51	41	54	34	54	35
16	[11.6s]	116	68	54	68	55	46	44	53	42	57	37	58	37

DS11 unordered_set<unordered_set<int>>

	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
4	[43s]	105	141	138	148	143	50	49	54	50	58	55	59	55
5	[42.1s]	104	140	134	138	143	47	46	51	46	55	51	57	53
6	[44.9s]	105	138	132	143	139	43	43	49	43	54	48	58	51
7	[55.3s]	107	116	104	114	102	39	39	48	40	57	46	57	47
8	[68.6s]	107	102	88	107	90	36	36	51	37	54	40	53	41
9	[68.2s]	102	106	92	107	93	35	35	51	35	52	38	54	40
10	[70.5s]	102	103	89	102	88	33	32	50	33	52	37	51	36
11	[73.3s]	101	98	84	98	85	34	35	50	34	50	35	50	36
12	[79.6s]	102	92	79	92	79	34	33	48	34	47	33	47	33
13	[80.8s]	105	83	71	83	71	35	35	49	35	47	33	47	33
14	[85.1s]	94	79	67	79	67	38	38	51	38	56	42	56	43
15	[91.6s]	93	76	65	76	65	43	42	56	43	63	50	63	50
16	[93.1s]	90	73	62	72	62	45	44	57	44	65	52	65	52

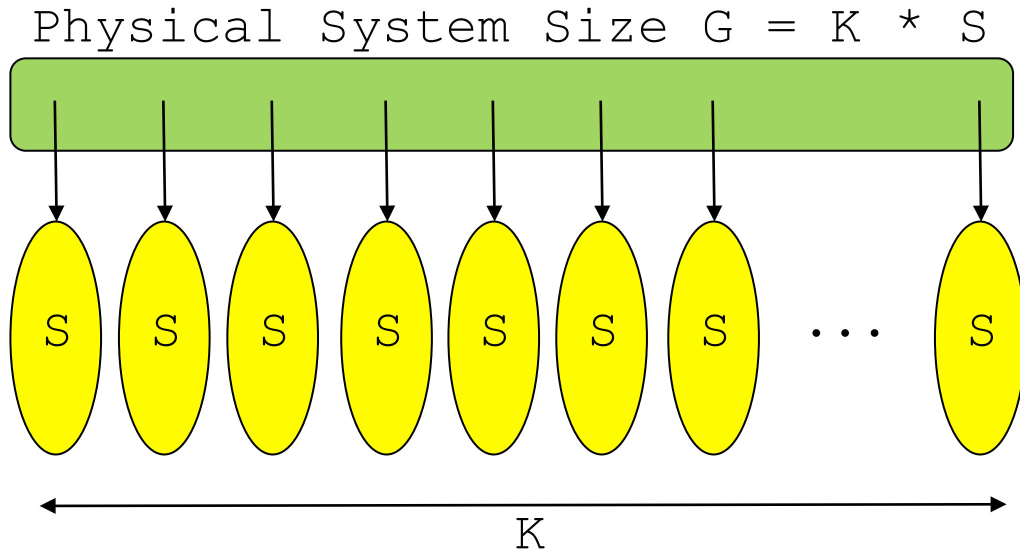
DS12 unordered_set<unordered_set<string>>

Most of the results above are not particularly surprising (to us), but with one exception: AS2 is substantially worse than AS1, but only for DS5-8 – i.e., for all complex data structures nested within a properly reserved `std::vector`. Further investigation is clearly warranted.

Keep in mind that the member functions of all of our allocators (except for the global default, AS1) are implemented "out of line" and that we know empirically (from Benchmark III, section 10) that there can be substantial benefits to making all such methods inline.

8 Benchmark II: Variation in Locality (long running)

One of the most valuable aspects of allocators is not that they speed up short-running programs, but that they stop long–running ones from slowing down over time. All global allocators eventually exhibit fragmentation: Memory that, at one time, dispensed contiguously, no longer does so, and runtime performance can start to degrade.

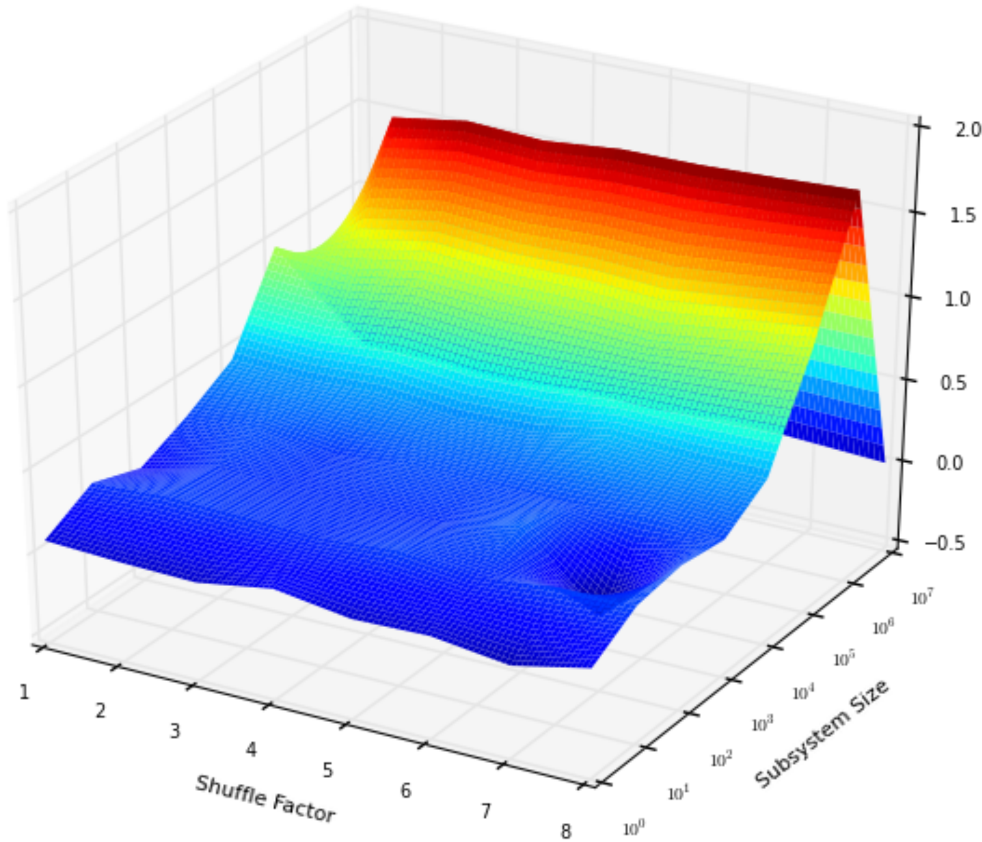


In order to demonstrate this common phenomenon without involving any local arena allocators, we created a simple program that acts like a long-running time-multiplexed system. This system will consist of a `std::vector<Subsystem*>`, where each subsystem is modeled as an `std::list<int>`. The global physical system size, G, will be defined as the total number of entries in the combined lists. The size of each subsystem, S, will be the initial number of links per list in each subsystem. The number of subsystems, K, will therefore be the (integral) ratio G/S. At the start of the program, each subsystem is `new`-ed, in-turn, which, when constructed populates itself with the specified S links. The system is now in its initial state.

This experiment is geared towards identifying opportunities for the use of allocators (specifically a multipool allocator) before actually plugging one in. To that end, we want to contrast the runtime performance of subsystems where memory has been allocated contiguously and where it has been "shuffled" over time to be less so. We therefore define a parameter, SF, which represents the *shuffle factor.* Specifying a shuffle factor of 0 leaves the system in its initial sate. A shuffle factor of 1 means that each linked list is visited and popped (from the front), immediately after which a new value is pushed onto (the back of) some randomly chosen list in the system until each element in each list has been popped exactly once. A shuffle factor of 2 means that the process is repeated the same number of times, although there is no longer any assurance that all of the lists still have the same length (as they did initially). The larger the shuffle factor, the more discontiguous and random the memory within each subsystem becomes.

In order to determine the extent to which local memory allocators might be useful (prior to actually installing them), we wanted to measure the effect on memory access times within each subsystem as we vary the amount of shuffling. To do that, we will want to iterate through the linked list in each subsystem some number of times – accessing each integer datum in turn – before moving to the next subsystem. An *access factor* (AF) of 2 denotes two complete passes through a subsystem's linked list

before moving to the next one in the vector of subsystems. While we are at it, we will also want to vary the number subsystems, K, and, inversely, subsystem-size, S, so as to keep the overall problem-size, G, constant.



	1	2	3	4	5	6	7	8
10^0	0.077	0.057	0.047	0.123	0.057	0.070	0.011	0.113
10^1	0.237	0.268	0.257	0.236	0.275	0.225	0.276	0.283
10^2	0.141	0.253	0.240	0.243	0.243	0.336	-0.091	0.323
10^3	0.276	0.307	0.251	0.298	0.293	0.285	0.314	0.282
10^4	0.444	0.429	0.446	0.401	0.418	0.502	0.468	0.458
10^5	0.964	0.930	1.009	0.974	0.967	0.932	0.993	1.048
10^6	0.478	1.674	1.741	1.717	1.759	1.760	1.777	1.800
10^7	0.012	0.008	-0.012	0.024	0.012	0.046	0.021	-0.003

The data and graph above illustrates the additional access runtimes (after shuffle times are subtracted) scaled to a run without shuffling for comparable systems in

which the shuffle factor (columns) ranges from 1 to 8 and the number of subsystem sizes (rows) range from 10^0 to 10^7. The physical size of each system is the same at 10^7 (links), and the access factor (AF) is maintained at 10 (i.e., each link of a subsystem is accessed sequentially10 times before moving to the next subsystem). This data was obtained using a Lenovo W520 laptop having four CPUs (eight threads) and 32 Gigabytes of RAM.

As we can see, increasing the shuffle factor at lower values of SF substantially affects the runtime cost of accessing the data. As the shuffle factor continues to increase, however, its effect on access runtime quickly reaches a horizontal asymptote, after which no additional performance desegregation is observed. The adverse effect of shuffling on memory access times appears to be relatively more pronounced for fewer larger subsystems (e.g., S = 10^6) than for many smaller ones (e.g., S = 10^3).

Given a sufficient amount of memory shuffle (say, SF = 5), we'd like to determine more precisely under what specific circumstances a lack of physical locality within subsystems most adversely affects the relative runtime of accessing memory (and therefore fairly begs for a local allocator). So far, we can fully characterize our system with just four parameters (G, S, AF, and SF). Recall from section 5, however, that locality is defined in terms of three factors: number of instructions (I), size of memory involved (M), and number of transitions away from the subsystem (T).

In order to model the difference between higher temporal locality (where I/T is relatively large) and lower temporal locality (where I/T is relatively small), we need to introduce a fifth parameter called the repeat factor, RF, that governs the number of times to traverse the vector of subsystems (each time performing the local accesses as governed by AF). By keeping the product of the local accesses (AF) and the subsystem iterations (RF) constant, we can observe the relative effects of high versus low temporal locality for the same number of total accesses.

If we are to make a fair comparison regarding the relative runtime cost of shuffled memory, we'll need to do the same amount of work shuffling memory either way. We will therefore hijack the sign of the shuffle factor to imply whether or not the shuffle occurs before (+) or after (-) the indicated data access pattern. For convenience, we will also assume that a negative global physical size (G) implies a (positive) binary exponent for both that value and the subsequent subsystem size (S). Using this notation, we can concisely characterize arbitrary runs of the program:

- `20 18 64 -3 4`: The global physical size (G) is 2^{20}. The initial size of each of the (four) subsystems (S) is 2^{18}. On each of the 4 iterations through the subsystems (RF), each element of each subsystem will be accessed 64 times (AF). After accessing the data, the entire contents of each subsystem will be shuffled 3 times (SF).
- `20 18 64 +3 4`: Same as above, except that the shuffling of data occurs *before* accessing the data.
- `20 18 4 +3 64`: Same as above, except that each subsystem's linked list is iterated over only four times before moving to the next subsystem, thereby

reducing temporal locality while keeping the overall number of memory accesses the same.

- `21 18 4 +3 64`: Same as above, except the overall physical size of the problem has doubled.
- `21 19 4 +3 64`: Same as above, except the size of each individual subsystem has doubled.
- `20 19 4 +5 64`: Same as above, except the number of times each subsystem is shuffled has increased by two.

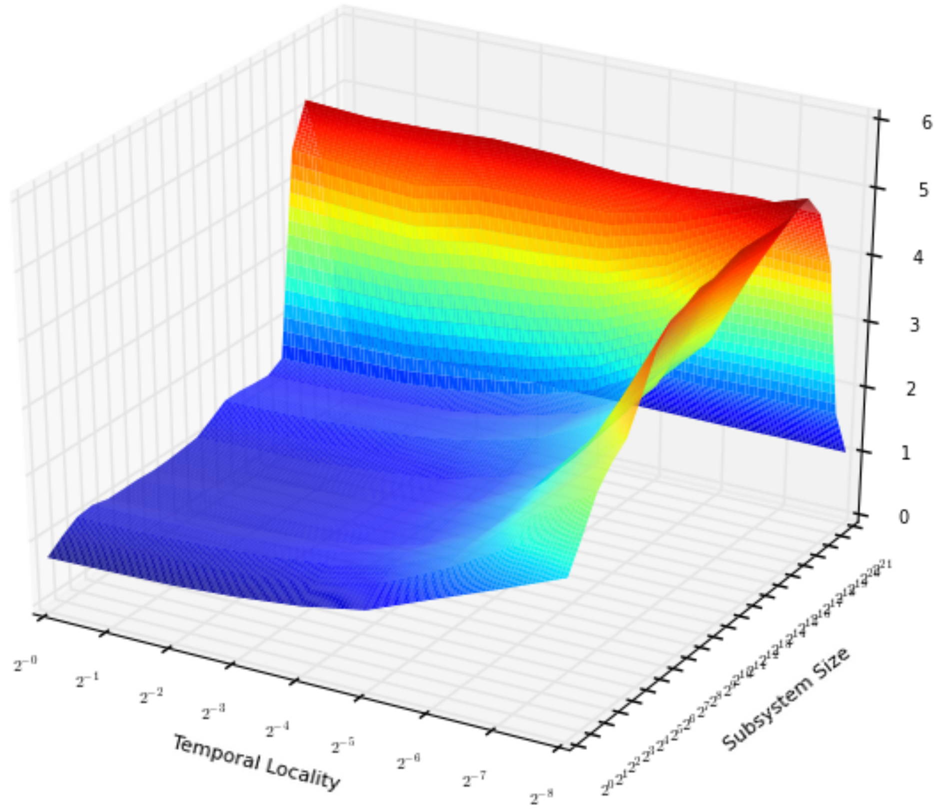
In order to explore the entire space, we assumed a shuffle factor (SF) of 5 and examined an increasingly large sequence of physical design spaces, contrasting both physical and temporal locality. Physical locality was determined by the ratio of subsystem size to overall system size, while temporal locality was defined by the ratio of the number of instructions executed within the subsystem to the number of transitions away from the subsystem over the duration of interest.

When the size of a problem is small, all of it fits in cache, and there is no need for a memory allocator. Once the problem size exceeds that which can be fully accommodated by the computer's cache memory, local memory allocators become relevant. For physical sizes below 2^{18}, there was no observable benefit for using allocators on the laptop.

The results of two specific runs, the first of size 2^{21} and the second of size 2^{25} follow. Each of these runs clearly show that, when the temporal locality is high, the greatest opportunity for effective use of allocators occurs when subsystem size is relatively large, and quickly tapers off with reduced subsystem size. On the other hand, when temporal locality is low, the opportunity for significant performance improvement using local allocators spans a much wider range of subsystem sizes.

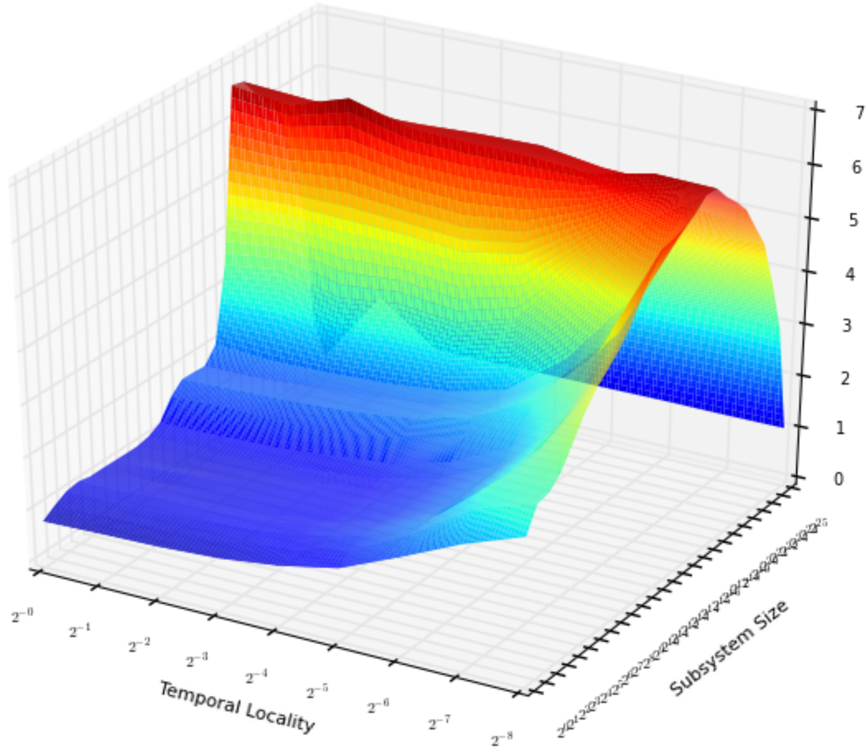
The two pictures below are reminiscent of the process of inflating a hot-air balloon: The low-locality (near) end is fully inflated, while the high-locality (far) end is only partially so. The greater the area under the curve, the more opportunity there is for a local allocator to be useful at improving runtime performance.

Problem Size = 2^{21}



	2^-0	2^-1	2^-2	2^-3	2^-4	2^-5	2^-6	2^-7	2^-8
2^0	0.726	0.748	0.775	0.841	0.942	1.126	1.529	1.921	2.332
2^1	0.900	0.924	0.955	1.013	1.124	1.366	1.833	2.400	2.779
2^2	1.042	1.045	1.087	1.146	1.265	1.498	1.920	2.670	3.275
2^3	1.072	1.078	1.101	1.162	1.286	1.533	1.959	2.792	3.550
2^4	1.011	1.032	1.062	1.126	1.238	1.477	1.959	2.812	3.738
2^5	1.012	1.030	1.093	1.151	1.266	1.503	1.989	2.925	4.197
2^6	1.015	1.046	1.076	1.145	1.277	1.562	2.080	3.074	4.724
2^7	0.998	1.017	1.067	1.135	1.272	1.582	2.104	3.177	5.008
2^8	1.010	1.029	1.048	1.129	1.272	1.550	2.081	3.173	5.070
2^9	1.037	1.054	1.085	1.159	1.303	1.568	2.112	3.206	5.183
2^10	1.069	1.099	1.117	1.205	1.327	1.599	2.118	3.240	5.182
2^11	1.208	1.210	1.251	1.324	1.486	1.729	2.234	3.266	5.259
2^12	1.356	1.366	1.414	1.471	1.620	1.882	2.367	3.417	5.321
2^13	1.360	1.381	1.430	1.472	1.609	1.861	2.397	3.412	5.308
2^14	1.333	1.345	1.374	1.461	1.625	1.844	2.348	3.380	5.293
2^15	1.319	1.329	1.367	1.416	1.560	1.827	2.365	3.368	5.282
2^16	1.448	1.463	1.484	1.581	1.740	2.028	2.571	3.689	5.299
2^17	4.535	4.452	4.386	4.566	4.571	4.574	4.777	5.081	5.386
2^18	5.158	5.091	5.107	5.151	5.111	5.028	5.023	5.078	5.028
2^19	4.219	4.145	4.162	4.242	4.253	4.200	4.160	4.223	4.167
2^20	1.749	1.740	1.744	1.745	1.746	1.763	1.732	1.753	1.757
2^21	0.998	0.984	0.992	1.010	1.002	1.002	0.998	1.006	1.006

Problem Size = 2^{25}



	2^-0	2^-1	2^-2	2^-3	2^-4	2^-5	2^-6	2^-7	2^-8
2^0	0.760	0.795	0.824	0.888	1.030	1.265	1.759	2.285	2.700
2^1	0.912	0.953	0.971	1.050	1.191	1.445	2.025	2.773	3.153
2^2	1.035	1.038	1.086	1.155	1.293	1.569	2.075	2.947	3.200
2^3	1.072	1.090	1.111	1.179	1.307	1.564	2.068	2.992	3.400
2^4	1.023	1.070	1.071	1.131	1.254	1.529	2.017	2.964	3.763
2^5	1.026	1.013	1.078	1.156	1.272	1.513	2.006	2.964	4.157
2^6	1.020	1.081	1.071	1.150	1.273	1.516	2.033	3.015	4.456
2^7	1.014	0.993	1.063	1.152	1.249	1.531	2.065	3.032	4.654
2^8	1.014	1.018	1.066	1.124	1.261	1.541	2.083	3.127	4.870
2^9	1.064	1.084	1.116	1.186	1.317	1.594	2.133	3.235	5.141
2^10	1.187	1.201	1.239	1.340	1.458	1.756	2.320	3.441	5.509
2^11	1.539	1.595	1.557	1.631	1.784	2.065	2.613	3.776	5.879
2^12	1.743	1.782	1.810	1.891	2.022	2.323	2.912	4.015	6.199
2^13	1.736	1.756	1.803	1.886	2.012	2.320	2.935	4.106	6.232
2^14	1.719	1.733	1.806	1.888	2.005	2.314	2.980	4.128	6.303
2^15	2.003	1.931	1.950	2.007	2.119	2.391	3.251	4.602	6.424
2^16	3.414	3.577	3.275	3.411	3.294	3.630	4.839	5.891	6.622
2^17	6.685	6.542	6.882	6.572	6.528	6.508	6.417	6.647	6.608
2^18	6.628	6.550	6.496	6.507	6.554	6.571	6.398	6.339	6.326
2^19	6.301	6.187	6.131	6.174	6.178	6.216	5.798	6.110	6.105
2^20	6.066	6.018	5.804	5.823	5.915	5.930	5.816	5.818	5.836
2^21	5.644	5.613	5.471	5.528	5.533	5.530	5.446	5.498	5.378
2^22	4.941	5.052	4.975	4.953	4.975	4.976	5.054	4.874	4.932
2^23	4.163	4.210	4.160	4.167	4.185	4.168	4.142	4.170	4.076
2^24	1.163	3.122	3.068	3.112	3.122	3.112	3.127	3.146	3.095
2^25	0.391	1.800	0.996	0.999	1.002	1.000	1.003	0.993	0.995

Theory is all well and good, but practice makes perfect. We now provide actual data obtained from using a small selection of allocator strategies (AS1, AS7, AS9, and AS13) on the same dedicated machine used in Benchmark I.

Benchmark Arguments	`new_delete` type parameter (**AS1**)	multipool type parameter (**AS7**)	multipool abstract base (**AS9**)	monotonic (multipool) abstract base (**AS13**)
-21 4 256 5 1	6.382s	7.99s (125%)	8.56s (134%)	9.91s (155%)
	6.299s	7.89s (125%)	8.64s (137%)	9.82s (156%)
	6.408s	7.93s (124%)	8.22s (128%)	9.81s (153%)
-21 4 256 5 0	4.312s	6.05s (140%)	6.50s (151%)	8.01s (186%)
	4.296s	6.13s (143%)	6.44s (150%)	7.91s (184%)
	4.306s	6.18s (143%)	6.38s (148%)	7.88s (183%)
-21 4 256 -5 1	6.063s	7.62s (126%)	8.57s (141%)	9.62s (159%)
	5.885s	7.93s (135%)	8.26s (140%)	9.55s (162%)
	6.041s	7.96s (132%)	8.11s (134%)	9.42s (156%)
-21 4 256 -5 0	4.316s	5.95s (138%)	6.24s (144%)	7.98s (185%)
	4.357s	5.96s (137%)	6.46s (148%)	7.99s (183%)
	4.355s	5.97s (137%)	6.38s (147%)	7.92s (182%)
-21 4 1 5 1	4.555s	6.07s (133%)	6.59s (145%)	7.98s (175%)
	4.502s	6.00s (133%)	6.83s (152%)	8.03s (178%)
	4.580s	6.09s (133%)	6.61s (144%)	7.97s (174%)
-21 4 1 5 0	4.251s	6.08s (143%)	6.23s (147%)	8.02s (189%)
	4.246s	5.92s (139%)	6.66s (157%)	7.98s (188%)
	4.395s	5.89s (134%)	6.35s (144%)	7.84s (178%)
-21 4 1 -5 1	4.398s	6.07s (138%)	6.47s (147%)	8.06s (183%)
	4.305s	6.23s (145%)	6.51s (151%)	8.04s (187%)
	4.285s	5.94s (139%)	6.35s (148%)	8.04s (188%)
-21 4 1 -5 0	4.247s	5.96s (140%)	6.37s (150%)	7.83s (184%)
	4.306s	5.97s (139%)	6.66s (155%)	7.94s (184%)
	4.347s	5.90s (136%)	6.45s (148%)	7.86s (181%)

Benchmark Arguments	`new_delete` type parameter (**AS1**)	multipool type parameter (**AS7**)	multipool abstract base (**AS9**)	monotonic (multipool) abstract base (**AS13**)
-21 17 256 5 1	39.577s	4.89s (12%)	5.03s (13%)	5.00s (13%)
	37.827s	5.00s (13%)	5.09s (13%)	4.97s (13%)
	40.415s	4.95s (12%)	5.04s (12%)	4.95s (12%)
-21 17 256 5 0	0.884s	0.49s (55%)	0.57s (64%)	0.57s (64%)
	0.908s	0.48s (53%)	0.57s (63%)	0.56s (62%)
	0.855s	0.52s (61%)	0.56s (66%)	0.57s (67%)
-21 17 256 -5 1	2.875s	3.06s (106%)	3.14s (109%)	2.99s (104%)
	2.934s	3.06s (104%)	3.10s (106%)	3.07s (105%)
	2.847s	2.98s (105%)	3.07s (108%)	2.98s (105%)
-21 17 256 -5 0	0.918s	0.51s (56%)	0.57s (62%)	0.62s (68%)
	0.954s	0.50s (52%)	0.56s (59%)	0.56s (59%)
	0.932s	0.51s (55%)	0.57s (62%)	0.57s (61%)
-21 17 1 5 256	62.526s	8.81s (14%)	8.61s (14%)	8.75s (14%)
	62.194s	8.43s (14%)	8.43s (14%)	8.30s (13%)
	63.505s	8.76s (14%)	8.74s (14%)	8.28s (13%)
-21 17 1 5 0	0.880s	0.49s (55%)	0.57s (64%)	0.56s (64%)
	0.880s	0.50s (57%)	0.59s (67%)	0.56s (64%)
	0.898s	0.48s (53%)	0.57s (64%)	0.59s (66%)
-21 17 1 -5 256	3.864s	4.61s (119%)	4.73s (122%)	4.75s (123%)
	3.875s	4.75s (123%)	4.80s (124%)	4.68s (121%)
	3.959s	4.55s (115%)	4.66s (118%)	4.61s (117%)
-21 17 1 -5 0	0.900s	0.48s (53%)	0.57s (63%)	0.57s (63%)
	0.917s	0.51s (56%)	0.54s (58%)	0.59s (64%)
	0.865s	0.48s (55%)	0.60s (70%)	0.55s (64%)

Each of the parameter combinations shown above (first column) was tried three times (to show variation) using four representative allocation strategies (columns 2-5). (Note that a monotonic allocator alone is not applicable in this usage scenario due to its long running, low-utilization nature). We could have subtracted out the (extraneous) fixed cost of shuffling, but such was not necessary to illustrate the quite dramatic benefits of employing local memory allocators to alleviate the runtime cost of memory fragmentation over long-running programs. We have, however, included a

zero-iteration run (a 0 in the last column) along with each test scenario for those who might seek to refine the performance advantages of local memory allocators with respect to access locality (**L**) in such circumstances.

9 Benchmark III: Variation in Utilization

To demonstrate the effect of Utilization, memory was allocated in chunks (of size S) until a first threshold was reached; the amount of active memory (A) to use. Then, a chunk was deallocated and another chunk allocated until the desired total amount of allocated memory (T) was reached. After every allocation, the value at the first byte of the allocation was incremented. The data collected depicts a large variation in A / T; the definition of Utilization. Since virtually no other work is done, the Density of this benchmark's allocations is extremely high.

The three size parameters T, A, and S are measured in bytes. The results of the experiment are normalized to the result for AS1. Specifically, the results under AS1 are times in seconds and the values under the other allocators are a percentage of the AS1 value; lower implies a shorter run time. The measurements were obtained on a system with six Intel X5670 @ 2.93 GHz and 96 GB of memory installed. While the system was not dedicated to this task, it was used during off-hours.

Total Allocated Memory (T) = 2^{30}

T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{30}	2^{15}	2^{10}	0.066s	115	546	551	56	62	52	60
2^{30}	2^{16}	2^{10}	0.065s	112	488	496	51	60	52	60
2^{30}	2^{17}	2^{10}	0.064s	113	492	500	53	61	54	63
2^{30}	2^{18}	2^{10}	0.065s	107	479	491	52	60	53	60
2^{30}	2^{19}	2^{10}	0.066s	108	550	554	53	59	52	59
2^{30}	2^{20}	2^{10}	0.065s	109	563	576	54	62	54	62
2^{30}	2^{20}	2^{11}	0.033s	108	1079	1086	54	61	55	62
2^{30}	2^{20}	2^{12}	0.017s	109	1994	1840	55	64	57	65
2^{30}	2^{20}	2^{13}	0.008s	108	1790	1802	127	130	1817	1843
2^{30}	2^{20}	2^{14}	0.004s	111	1739	1751	123	128	1764	1781
2^{30}	2^{20}	2^{15}	0.002s	107	1687	1703	118	121	1712	1717

Total Allocated Memory (T) = 2^{31}

T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{31}	2^{15}	2^{10}	0.130s	115	644	558	55	60	54	62
2^{31}	2^{16}	2^{10}	0.132s	110	565	551	53	59	53	60
2^{31}	2^{17}	2^{10}	0.128s	110	587	637	52	61	52	60
2^{31}	2^{18}	2^{10}	0.128s	113	712	706	53	61	54	61
2^{31}	2^{19}	2^{10}	0.129s	110	698	698	54	62	54	61
2^{31}	2^{20}	2^{10}	0.132s	108	680	682	63	61	54	61
2^{31}	2^{20}	2^{11}	0.067s	108	1039	1040	53	60	53	60
2^{31}	2^{20}	2^{12}	0.034s	107	2208	2015	53	62	52	61
2^{31}	2^{20}	2^{13}	0.017s	106	1710	1713	119	124	1716	1751
2^{31}	2^{20}	2^{14}	0.008s	110	1728	1739	123	126	1762	1772
2^{31}	2^{20}	2^{15}	0.004s	107	1753	1761	122	127	1772	1794

Total Allocated Memory (T) = 2^{32}

T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{32}	2^{15}	2^{10}	0.261s	109	fail	fail	53	59	52	60
2^{32}	2^{16}	2^{10}	0.259s	115	fail	fail	52	60	53	61
2^{32}	2^{17}	2^{10}	0.270s	111	fail	fail	50	57	50	58
2^{32}	2^{18}	2^{10}	0.258s	109	fail	fail	53	60	53	60
2^{32}	2^{19}	2^{10}	0.258s	109	fail	fail	54	61	54	61
2^{32}	2^{20}	2^{10}	0.257s	109	fail	fail	54	62	54	62
2^{32}	2^{20}	2^{11}	0.133s	107	fail	fail	54	61	54	61
2^{32}	2^{20}	2^{12}	0.067s	108	fail	fail	53	62	55	63

2^{32}	2^{20}	2^{13}	0.033s	108	fail	fail	122	129	fail	fail
2^{32}	2^{20}	2^{14}	0.017s	111	fail	fail	124	127	fail	fail
2^{32}	2^{20}	2^{15}	0.008s	107	fail	fail	122	127	fail	fail

Total Allocated Memory (T) = 2^{33}

T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{33}	2^{15}	2^{10}	0.517s	108	fail	fail	54	60	54	61
2^{33}	2^{16}	2^{10}	0.513s	110	fail	fail	54	61	53	61
2^{33}	2^{17}	2^{10}	0.512s	111	fail	fail	53	61	53	61
2^{33}	2^{18}	2^{10}	0.523s	110	fail	fail	52	60	52	60
2^{33}	2^{19}	2^{10}	0.532s	109	fail	fail	52	60	52	60
2^{33}	2^{20}	2^{10}	0.518s	108	fail	fail	54	61	53	61
2^{33}	2^{20}	2^{11}	0.263s	108	fail	fail	54	61	56	63
2^{33}	2^{20}	2^{12}	0.135s	107	fail	fail	53	62	53	61
2^{33}	2^{20}	2^{13}	0.068s	107	fail	fail	120	126	fail	fail
2^{33}	2^{20}	2^{14}	0.034s	108	fail	fail	122	125	fail	fail
2^{33}	2^{20}	2^{15}	0.017s	113	fail	fail	126	133	fail	fail

Total Allocated Memory (T) = 2^{34}

T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{34}	2^{15}	2^{10}	1.035s	108	fail	fail	55	61	55	62
2^{34}	2^{16}	2^{10}	1.024s	111	fail	fail	53	60	53	61
2^{34}	2^{17}	2^{10}	1.034s	111	fail	fail	53	61	53	61
2^{34}	2^{18}	2^{10}	1.027s	112	fail	fail	53	61	54	61

2^{34}	2^{19}	2^{10}	1.048s	110	fail	fail	53	61	53	60
2^{34}	2^{20}	2^{10}	1.073s	107	fail	fail	52	59	52	59
2^{34}	2^{20}	2^{11}	0.523s	109	fail	fail	55	61	56	63
2^{34}	2^{20}	2^{12}	0.273s	108	fail	fail	52	61	53	60
2^{34}	2^{20}	2^{13}	0.132s	111	fail	fail	125	130	fail	fail
2^{34}	2^{20}	2^{14}	0.066s	109	fail	fail	124	131	fail	fail
2^{34}	2^{20}	2^{15}	0.033s	110	fail	fail	125	130	fail	fail

Total Allocated Memory (T) = 2^{35}

T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{35}	2^{15}	2^{10}	2.098s	110	fail	fail	53	59	53	60
2^{35}	2^{16}	2^{10}	2.022s	111	fail	fail	54	63	55	62
2^{35}	2^{17}	2^{10}	2.064s	110	fail	fail	53	60	53	60
2^{35}	2^{18}	2^{10}	2.055s	109	fail	fail	54	61	54	61
2^{35}	2^{19}	2^{10}	2.148s	108	fail	fail	52	59	55	62
2^{35}	2^{20}	2^{10}	2.083s	115	fail	fail	54	61	53	61
2^{35}	2^{20}	2^{11}	1.065s	107	fail	fail	54	61	55	62
2^{35}	2^{20}	2^{12}	0.549s	104	fail	fail	52	61	54	62
2^{35}	2^{20}	2^{13}	0.263s	108	fail	fail	124	128	fail	fail
2^{35}	2^{20}	2^{14}	0.133s	111	fail	fail	123	127	fail	fail
2^{35}	2^{20}	2^{15}	0.068s	109	fail	fail	122	127	fail	fail

The most striking result is that some of the tests failed to run to completion; the system's memory was exhausted. Clearly, when we choose an allocator, the need for re-use of deallocated memory is a critical factor.

The results for the largest three S values in all the tables expose the effect of an implementation detail of the used multipool. Allocations larger than a certain size (2^{12} bytes as per code inspection) will be passed directly to the underlying allocator. As such, for S > 2^{12}, there is noticeable performance degradation for the multipool allocators and the creation of failure scenarios for AS11 and AS13.

10 Benchmark IV: Variation in Contention

In this experiment, a set of threads was created and used to repeatedly allocate and deallocate a chunk of memory. To emphasize the cost of Contention, every function called by a thread had an instance of an allocator. For the default global allocator, AS1, and the new/delete allocator, AS2, all of the threads will contended for the same allocator. For the other allocators, each thread had access to its own private allocator; hence, there is no contention except for when these allocators must make a request to their backing allocators. After every allocation the value at the first byte of the memory was incremented. Note that the Allocation Density of this experiment is extremely high.

The size parameter (S) is measured in bytes. The other parameters for this experiment are the number of iterations (N) and the number of threads (W). The results of the experiment are normalized to the result for AS1. Specifically, the results under AS1 are times in seconds and the values under the other allocators are a percentage of the AS1 value; lower implies a shorter run time. The measurements were obtained on a system with six Intel X5670 @ 2.93 GHz and 96 GB of memory installed. While the system was not dedicated to this task, it was used during off-hours.

Number of Iterations (N) = 2^{15}, Size of Allocation (S) = 2^6

N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{15}	2^6	1	0.017s	106	51	53	43	44	43	43
2^{15}	2^6	2	0.023s	104	65	68	53	46	44	44
2^{15}	2^6	3	0.025s	102	76	75	56	58	53	53
2^{15}	2^6	4	0.026s	104	86	87	56	61	55	62
2^{15}	2^6	5	0.028s	106	90	85	56	69	59	66
2^{15}	2^6	6	0.033s	94	84	87	58	62	58	61
2^{15}	2^6	7	0.031s	108	87	87	61	65	65	69
2^{15}	2^6	8	0.036s	103	101	99	57	62	57	60

Number of Iterations (N) = 2^{15}, Size of Allocation (S) = 2^7

N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{15}	2^7	1	0.022s	107	72	75	33	33	33	33
2^{15}	2^7	2	0.036s	96	65	63	33	33	33	29
2^{15}	2^7	3	0.035s	101	83	80	39	35	35	41
2^{15}	2^7	4	0.037s	96	91	96	43	44	48	42
2^{15}	2^7	5	0.040s	107	120	119	45	47	46	47
2^{15}	2^7	6	0.042s	98	104	109	49	47	48	48
2^{15}	2^7	7	0.045s	98	112	112	44	46	45	45
2^{15}	2^7	8	0.051s	101	121	121	41	40	43	42

Number of Iterations (N) = 2^{15}, Size of Allocation (S) = 2^8

N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{15}	2^8	1	0.025s	100	95	95	30	30	30	30
2^{15}	2^8	2	0.035s	99	121	122	34	34	33	37
2^{15}	2^8	3	0.036s	102	148	149	45	45	40	44
2^{15}	2^8	4	0.038s	98	160	162	44	47	43	44
2^{15}	2^8	5	0.043s	97	165	166	45	44	44	40
2^{15}	2^8	6	0.041s	103	204	202	49	53	48	49
2^{15}	2^8	7	0.042s	99	224	221	47	48	48	51
2^{15}	2^8	8	0.051s	100	210	211	46	45	45	47

Number of Iterations (N) = 2^{16}, Size of Allocation (S) = 2^8

N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13

2^{16}	2^{8}	1	0.050s	89	125	124	29	29	29	30
2^{16}	2^{8}	2	0.056s	102	152	154	40	41	40	42
2^{16}	2^{8}	3	0.056s	101	186	173	44	48	40	42
2^{16}	2^{8}	4	0.059s	105	231	228	46	45	42	47
2^{16}	2^{8}	5	0.069s	102	206	207	40	42	42	40
2^{16}	2^{8}	6	0.071s	92	231	228	43	42	39	44
2^{16}	2^{8}	7	0.077s	97	237	238	43	37	40	40
2^{16}	2^{8}	8	0.081s	99	280	286	39	40	38	43

Number of Iterations (N) = 2^{17}, Size of Allocation (S) = 2^{8}

N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{17}	2^{8}	1	0.086s	102	136	136	34	34	34	34
2^{17}	2^{8}	2	0.099s	101	178	178	38	44	38	43
2^{17}	2^{8}	3	0.101s	106	188	185	38	38	37	41
2^{17}	2^{8}	4	0.104s	103	213	249	37	40	38	44
2^{17}	2^{8}	5	0.114s	101	245	251	38	39	39	42
2^{17}	2^{8}	6	0.114s	95	274	271	41	43	39	42
2^{17}	2^{8}	7	0.143s	98	246	255	33	34	34	34
2^{17}	2^{8}	8	0.138s	95	302	315	37	36	37	42

Number of Iterations (N) = 2^{18}, Size of Allocation (S) = 2^{8}

N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{18}	2^{8}	1	0.171s	102	158	159	34	34	34	34
2^{18}	2^{8}	2	0.193s	97	164	155	35	36	35	36

2^{18}	2^{8}	3	0.198s	95	191	223	34	35	34	34
2^{18}	2^{8}	4	0.205s	93	265	227	35	35	35	35
2^{18}	2^{8}	5	0.209s	108	264	278	35	36	36	40
2^{18}	2^{8}	6	0.201s	104	300	300	39	38	36	44
2^{18}	2^{8}	7	0.239s	105	289	301	37	38	37	39
2^{18}	2^{8}	8	0.250s	102	328	332	34	36	34	41

Number of Iterations (N) = 2^{19}, Size of Allocation (S) = 2^{8}

N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
2^{19}	2^{8}	1	0.344s	100	156	158	34	34	34	34
2^{19}	2^{8}	2	0.375s	97	198	198	33	34	34	34
2^{19}	2^{8}	3	0.362s	101	245	229	35	36	35	36
2^{19}	2^{8}	4	0.373s	102	257	258	35	35	35	35
2^{19}	2^{8}	5	0.380s	101	269	265	35	35	35	37
2^{19}	2^{8}	6	0.382s	102	337	344	36	37	39	38
2^{19}	2^{8}	7	0.443s	95	356	326	36	38	36	41
2^{19}	2^{8}	8	0.478s	95	353	335	32	33	34	37

Since modern default global allocators were designed with threading as a concern, the results are not jaw-dropping. The benchmarks demonstrate, again, the relative efficiency of the allocators; the default global allocator must pay a premium to handle multiple threads concurrently. Interestingly, the monotonic allocators performed more and more poorly as the total amount of memory allocated memory increased (likely due to a dearth of physical locality within the sequential buffer itself).

11 Conclusion

Object-level control over memory allocation is intrinsic to C++, and must always be so if this language hopes to retain its supremacy as the high-level “systems” language it has always aspired to be. Supporting object-specific memory allocation is admittedly an added burden – exacerbated by an initially poor model – which is finally being

addressed by *N3916: Polymorphic Memory Resources*. Any future incarnation of STL should incorporate the lessons elucidated here.

12 References

[1] The Bloomberg BDE Library open source distribution, https://github.com/bloomberg/bde

[2] John Lakos, *Large Scale C++ Software Design*, Addison-Wesley, 1996.

[3] Pablo Halpern, *N3916: Polymorphic Memory Resources*.