Doc No: P0089R1
Date: 2016-02-12
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 (Revision 2)

Abstract

Performance requirements drive many of our most difficult design choices. In memory management, such choices can have surprising and far-reaching effects. Although performance of global memory allocators has improved markedly in recent years, use of local memory allocators can provide significant (sometimes even dramatic) benefits in commonly encountered circumstances we have tried to identify here.

To make reasoned choices on the use of local memory allocators, we need to understand where and how their use may affect runtime performance. We have identified several measures of how systems can stress a global allocator, and may benefit by applying a well-chosen local allocator in its place. If we are to choose wisely where and how to apply a local allocator, we need objective measurements. We have identified several usage patterns which we have encoded into benchmarks to identify precisely where local allocators do (and where they do not) provide substantial benefits. This paper presents our results with limited analysis to help support informed discussion.

Possibly the most significant result is that, where use of a local allocator does yield dramatic improvements, the number of operations are about the same: The slower benchmark run times for the global allocator are dominated by stalls waiting on cache interactions with main memory (due to a severe lack of *physical* and *temporal* locality); the ability to use a local allocator empowers us to act to avoid such stalls.

Implementations of standard allocators (and others) are freely available today – accompanied by copious 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 in a fork of that repository, <https://github.com/bloomberg/bde-allocator-benchmarks>. In light of the data compiled here, there can be no remaining doubt about the industrial importance of providing program control over the allocators used for C++ containers.

Contents

0 Changes from P0089R0

The first version of this paper appeared as N4468. In both that version and P0089R0 the tables presented in Section 7 (Benchmark I) were laid out incorrectly: the columns for the monotonic allocator (AS3-AS6) contained the data for the multipool allocator, and similarly the columns for the multipool allocator (AS7-AS10) contained the data for the monotonic allocator. The data in those tables has been re-arranged to be correct in this version, and the accompanying text revised accordingly.

In addition, P0089R0 omitted the allocator categorization diagram that appears at the start of Section 2.

This version of the paper also corrects typographical errors, and improves the wording of some difficult phrases.

Finally, this version adds a reference to a paper (P0213R0) being prepared concurrently with this paper by Graham Bleaney, which attempts to independently recreate the data presented in P0089R0. Graham's work on P0213R0 led to the discovery of the swapped column data in Benchmark I.

1 Introduction

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

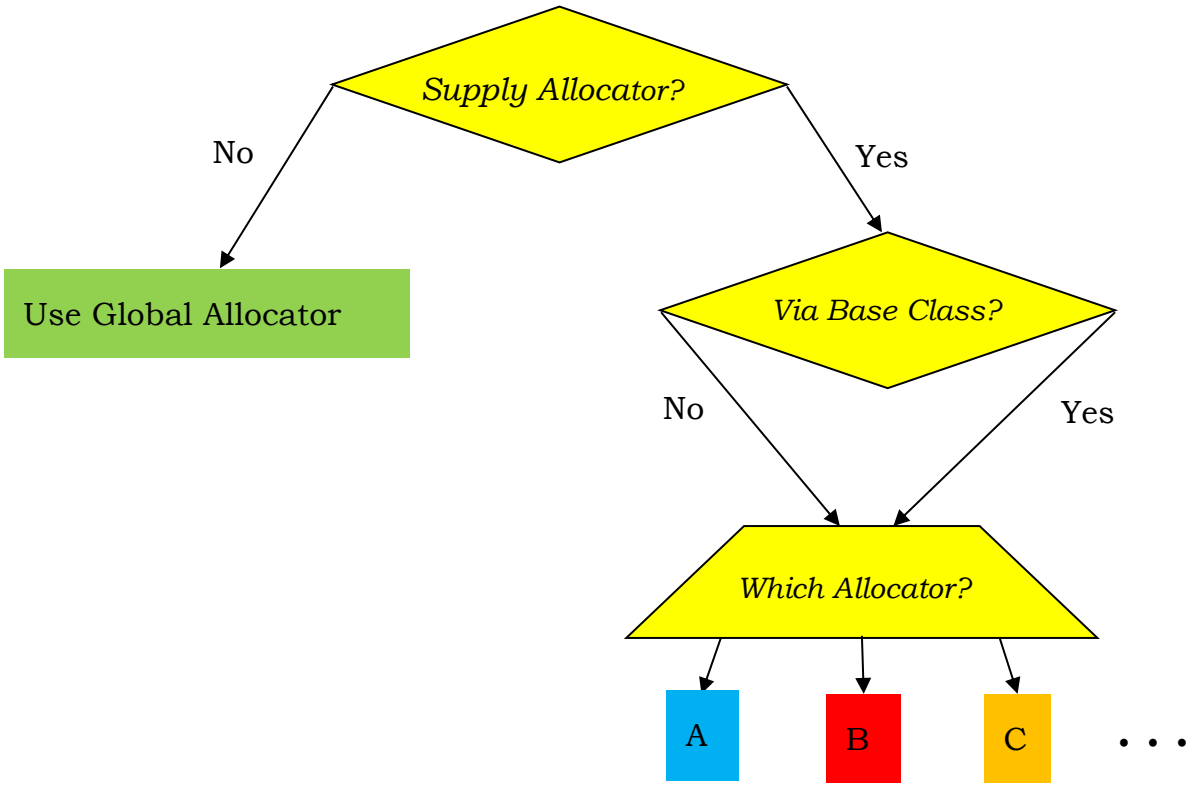
Should we instrument the standard library for such fine-tuning? The arguments against are typically that fine-grained memory management requires more up-front design effort, complicates interfaces, and may actually degrade performance where no local allocator, or a poorly chosen one, is supplied. These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

Nevertheless, a library instrumented to exploit local allocators enables benefits other than just enhanced runtime performance: Allocators can aid testing, debugging, and measurement. Not all memory is alike – some is faster for certain processors, some is shared, some may be write-protected, and we will need allocators to exploit such heterogeneous memory 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 supplying a local 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.



It is worth noting that we investigated alternative global allocators beside the native ones on the various platforms, including `tcmalloc` and `jemalloc`, and determined that the native allocators (e.g., the one currently shipped with GCC on Linux) performed as well or better. In short, it isn't about how good the global allocator is, but instead the relative benefits to having local knowledge of the nature of how allocation will occur. In some cases, e.g., Benchmark II, an allocator's runtime performance is entirely irrelevant compared to the physical locality of memory accesses it is able to preserve.

3 Available Concrete Allocators: Monotonic and Multipool

In this paper, we have selected the two kinds of allocators from the current Library Fundamentals TS: "monotonic" and "multipool".

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.

A *multipool allocator* is quite different. Each such allocator object 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, and freed memory is returned to that pool. When the pool has no free memory, the allocator delivers memory from increasingly larger blocks obtained from the backing allocator (possibly the global allocator), up to some (empirically determined) limit. Requests that exceed the maximum pool size pass directly through to the backing allocator. 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 instance, 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.

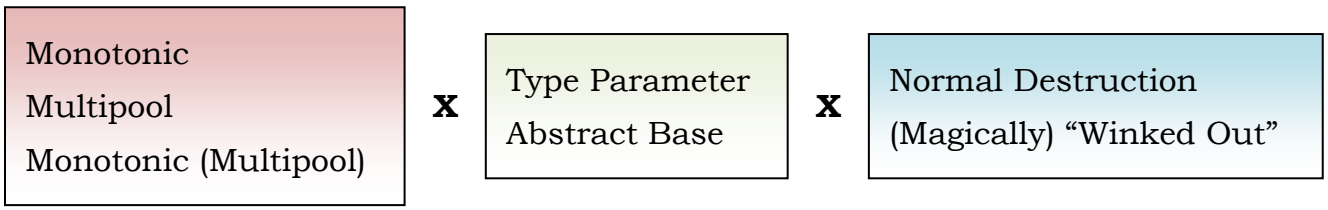
The runtime benefits of bypassing individual destruction of each element in a container can be significant, as de-allocating memory can sometimes be more costly than allocating it. Note that this "winking out" technique requires `new`-ing the container object itself into the managed allocator it is to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

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 consider up to 14 different allocation strategies for each of the benchmarks we will subsequently present. 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. (Supplying the default as a compile-time parameter produces the *same* object code as having it default, 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 the subset of popular 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 (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	(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”

Table 1: Allocation Strategies

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

It is worth noting that a Multipool allocator comes in two flavors: *synchronized* and *unsynchronized* (see the **bdlma** package in <https://github.com/bloomberg/bde>). Throughout Benchmarks I and II, we used the *synchronized* version – even though it was unnecessary to do so; in benchmarks III and IV used the *unsynchronized* version (because we could, as there was just one allocator per thread). The ability to use a

local allocator enables the additional choice of not forcing synchronization to be present where it is not needed. Demonstrating the (perhaps considerable) runtime improvement for avoiding such synchronization where possible will (for now) be left as an exercise for the reader.

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, effective use of local memory allocators is learned only from long 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 an object's memory management.

The first step in characterizing a problem such as this one is to identify its basic size parameters. Problems of vastly different sizes are not usefully comparable. Problem size can 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 obvious, and a single number that combines the two does not seem 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 physical hardware.

Given this overall "size" characterization (**N**, **W**), we now introduce five dimensions that (we assert) 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 subset of program execution. That is, when considering these dimensions, we are looking to improve the performance of a particular subsystem over a finite

duration of execution, rather than that of the program as a whole. Supporting whole-program allocation is the remit of the global allocator.

5.1 *Density of allocation operations (D)*

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 zero would imply that no allocation operations are employed, while a density of one 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 allocations 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 (unlike Bloomberg's **`bsl`** https://github.com/bloomberg/bde/tree/master/groups/bsl) do not do internal pooling are similarly capable of achieving a very 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-related) operations.

5.2 *Variation in allocated memory sizes (V)*

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 (at most) a single memory size is allocated, while a variation of 1 would suggest a much more diverse (e.g., *hyperbolic*) distribution of memory allocation sizes. A low variation value might (in theory) tend to suggest a pool-based allocator, whereas a higher value (again in theory) could perhaps favor a coalescing allocator (but see the actual data in Benchmark I). Keep in mind that requests that are relatively close in size might be treated equivalently.

5.3 *Locality facilitating memory access/manipulation (L)*

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 accessed for the duration

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

The locality, **L**, correlates directly to the instruction count, **I**, but inversely to both the memory footprint size, **M**, and the transition count, **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 (over the duration of interest, 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. *Physical locality* can be independently characterized by holding **T** constant, whereas *temporal locality* would be similarly characterized by holding **M** constant. Note that access locality – both *physical* and *temporal* – will turn out to play a dominant role 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 (U)*

Allocated memory **utilization** is a measure of the relative amount of allocated memory in use at any one time; it is defined as the maximum amount of memory in use by a subsystem, during the durations of interest, divided by the total amount of memory allocated by the subsystem over that period. A utilization of 1 means that, at some point, all of the memory ever allocated by a subsystem (over the duration of interest) is actively in use. A utilization that approaches zero implies a (typically long-running) subsystem in which the same memory is allocated and deallocated repeatedly. Subsystems exhibiting high utilization are often good candidates for monotonic allocators, while a long-running subsystem having low utilization is almost always much more suited to a multipool allocator, or perhaps a multipool allocator backed by a monotonic one (but see the benchmarks below).

5.5 *Contention due to concurrent memory allocations (C)*

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 indicates 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 or deallocate memory on every instruction executed (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 typically slow down subsystems in situations that do not require synchronization, while – compared to the use of local allocators – also degrading performance 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).

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 characterizing memory allocation is a challenge for anyone, including us, so we offer a mnemonic aid by way of a mascot: The mascot is a duck, and his name is **DVLUC**.

6 Designing Useful Benchmarks

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

As it turns out, a single problem that encompasses all five dimensions is not at all easy to invent, 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 as well as enabling each of the fourteen proposed allocation strategies (where appropriate) to have 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 versus the number of subsystems visited (*temporal locality*) while holding other benchmark parameters constant.

All the results presented here are from runs on a server having dual Intel Xeon E5-2620v2 processors, each having 6 cores (for a total of 12 cores) and 15 MB of L3 cache, running at a fixed clock rate of 2.1 GHz, with 16GB of DDR3-1600 RAM (with

13G available to processes), and otherwise unused. This particular processor has the "Sandy Bridge" architecture, from 2010, re-stepped ("v2") to a smaller die in 2013 and called "Ivy Bridge EP" (http://ark.intel.com/products/75789). Programs were all compiled using gcc-5.1, optimizing "`-O3 -march=native`", and run under Linux 3.18. All experiments used only one core at a time except for Benchmark IV, which measures Contention (**C**) and used more of the available cores.

In addition, we ran the same programs on several other configurations and platforms, including versions built with clang-3.6 on the machine described above, and with gcc and clang on an IBM POWER7 under Linux 3.10, and with MSVC2015R1 on an Intel Haswell desktop machine under Windows 7. Results of these runs can be found on the github site.

Finally, a separate effort has recently been made to recreate our experiments in order to confirm these results (P0213 by Graham Bleaney). We anticipate that paper will appear at approximately the same time as this revision.

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

In this experiment, we look at the process of creating a variety of isolated composite data structures, using them lightly (i.e., writing to each element exactly once using `memset` via a pointer-to-`volatile`), and then quickly destroying them. The set of data structures under test encompasses many of those we use every day, and were chosen specifically to explore the first two dimensions discussed earlier (section 5), namely Density (**D**), and Variation (**V**). Each standard container under consideration (`std::vector` and `std::unordered_set`) will ultimately consist of "leaf" objects of either `int` or `std::string`, where each string's length – chosen randomly over a uniform distribution between 33 and 1000 – is deliberately outside the range where the short-string optimization pertains.

The container implementations are the native ones for the platform, using `scoped_allocator_adaptor` to ensure that the same allocator is used for all parts of the data structure. The monotonic and multipool allocators come from the BSL library.

Twelve representative standard-library data structures were chosen – the fifth through twelfth being, respectively, `std::vector`s and `std::unordered_set`s of elements containing each of the first four data structure types:

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>

Table 2: Data Structures

The runtime results for executing 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 on just one of the several popular platforms we tried (section 6) are presented below.

Unlike our previous paper, however, all tabular numbers for this benchmark are presented (as heat maps) in terms of absolute run times in seconds (rather than percentages relative to the first column). Moreover, the color coding of the maps applies to an entire chart, rather than each individual row – this to help identify patterns – especially in allocation-strategies (columns) – that might otherwise be obfuscated. 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.

This process of creating and destroying each data structure is 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 leaf elements) and the number of iterations of creating and destroying it will be held constant, which we have chosen (arbitrarily) to be 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).

Although this benchmark focuses, primarily, on the dimensions of Density (**D**) and Variation (**V**), discussed in section 5, the relatively short-lived nature of the objects in this benchmark – along with their extremely high Utilization (**U**) – facilitate measuring the benefit of allocations strategies, such as AS4, AS6, AS8, AS10, AS12, and AS14, that "wink-out" object memory. Finally note that, in each of the tables below, Green indicates substantially shorter run times whereas yellow, orange, and especially red indicate longer run times.

7.1 DS1, vector<int>

	← global → virtual		← Monotonic →				← multipool →				← mono + multi →			
				← 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	1.2	1.9	0.3	0.4	0.4	0.4	0.8	1.0	0.9	1.1	0.6	0.7	0.8	0.7
2^7	0.9	1.6	0.3	0.4	0.4	0.4	0.5	0.7	0.6	0.7	0.5	0.5	0.6	0.5
2^8	0.8	1.0	0.2	0.4	0.4	0.3	0.4	0.6	0.5	0.6	0.3	0.5	0.5	0.5
2^9	0.8	1.0	0.2	0.4	0.4	0.4	0.3	0.5	0.5	0.5	0.3	0.4	0.4	0.4
2^{10}	0.7	0.9	0.2	0.3	0.4	0.4	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4
2^{11}	0.7	0.9	0.2	0.3	0.4	0.3	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4
2^{12}	0.7	0.9	0.2	0.3	0.4	0.4	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4
2^{13}	0.8	0.9	0.2	0.3	0.4	0.4	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4
2^{14}	0.8	0.9	0.2	0.3	0.4	0.4	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4
2^{15}	0.8	0.9	0.2	0.3	0.4	0.4	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4
2^{16}	0.8	0.9	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4	0.2	0.4	0.4	0.4

Table 3: DS1, vector<int>

This first data structure (DS1) corresponds to an `std::vector<int>` ranging in size from 2^6 (top row) to 2^{16} (bottom row). Recall that AS1 is the default (global) allocator accessed directly, and that AS2 is the default allocator accessed via pure-virtual functions in an abstract base class. The following three large blocks (four columns each) correspond to the three local allocator mechanisms: *monotonic* (AS3-AS6), *multipool* (AS7-AS10), and *monotonic* backing a *multipool* (AS11-AS14). The first pair of columns within each block (AS3-AS4, AS7-AS8, and AS11-AS12) correspond to direct access where the second pair (AS4-AS5, AS9-AS10, and AS13-AS14) correspond to access via an abstract base class. Finally, the first member of each pair (AS3, AS5, AS7, AS9, AS11, and AS13) corresponds to the normal destruction

process, whereas the second member of each pair (AS4, AS6, AS8, AS10, AS12, and AS14) corresponds to "winking out" the memory, bypassing normal destruction.

Note that each `std::vector` instance used in this benchmark is explicitly pre-sized (using `reserve`) to have exactly the needed capacity. Hence, the measurement data for `vector<int>` (DS1) involves only a single memory allocation/deallocation. Hence, for this first data structure, the allocation Density (**D**) was vanishingly small, and the requested memory-size Variation (**V**) was nil. Although the data for DS1 (above) is largely composed of test-apparatus artifacts, it exhibits recurring patterns across the various allocation strategies (columns) consistent with what is seen below.

The first observation is that direct access is superior to access via a base class for global and local allocators in essentially all cases. For the global allocator (AS1-AS2), this overhead ranged from ~20%-25% for larger vector sizes, but jumped sharply to ~60%-70% for the two smallest ones shown (64 and 128 elements). The clear winning strategy for each of the three local allocators was direct access *without* "winking out" memory (AS3, AS7, and AS11, respectively). Any attempt to deviate from typical usage dramatically reduced runtime performance (~50%-80%). (A plausible conjecture here would be that the optimizer is tuned for the typical case.)

When always "winking out" memory, accessing the allocator directly versus via a pure abstract base class generally made no statistically significant difference. Finally note that, except for the two smallest vectors (corresponding to the rows labeled 2^6 and 2^7), all of the local allocation strategies (AS3-AS14) were – at least – close to *twice as fast* as directly accessing the default allocator (AS1). It will turn out that this surprising observation can be repeated in each of the eleven remaining experiments in this benchmark, again in Benchmark III (except, of course, for the monotonic allocator alone (AS3-AS6)), and yet again in Benchmark IV. Note that Benchmark II deals entirely with locality of *access*, and therefore the runtime performance of the allocation and deallocation operations themselves is not relevant.

7.2 DS2, vector<string>

	← global →		← Monotonic →				← multipool →				← mono + multi →			
	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	68.9	67.3	12.9	12.8	13.3	12.9	18.1	17.8	18.2	17.7	15.5	14.8	15.6	14.8
2^7	68.8	68.2	12.8	12.9	13.2	12.9	20.6	20.2	20.6	20.4	15.1	14.3	15	14.4
2^8	70.8	68.9	13.2	12.8	13.6	12.9	30.8	30.4	30.7	30.3	15.3	14.6	15.4	14.7
2^9	73.1	71.2	13.5	13.5	13.9	13.5	38.2	37.6	38	37.3	15.9	15.1	15.9	15.1
2^{10}	75.4	74.3	13.6	13.5	14	13.7	41.1	40.3	41.6	40.9	16	15.1	15.9	15
2^{11}	76.9	74.5	13.6	13.5	14.1	13.6	43.9	43.2	43.7	42.6	16	15	16	15.1
2^{12}	76.1	74.8	13.7	13.5	14	13.6	41.2	38.8	40.6	39.4	15.9	14.9	15.8	15
2^{13}	76.1	74.8	13.6	13.6	14	13.6	41.4	39.2	41.3	39.9	15.9	15	15.8	14.9
2^{14}	78.3	76.5	13.6	13.6	14	13.6	45.8	42.3	44.8	44	16.1	15.2	16.2	15.4
2^{15}	90.4	91	20.2	20.1	20.5	20.1	62.2	58.7	62.2	58.2	26	25	26	24.9
2^{16}	103	103	21.5	21.3	21.8	21.3	66.5	59.2	65.1	59.9	27	25.3	27.1	25.2

Table 4: DS2, vector<string>

For DS2, `vector<string>`, we insert 2^n strings, where n again ranges from 2^6 (top row) to 2^{16} (bottom row). Each string is of randomly chosen, uniformly distributed length (in the range [33..1000] bytes), its data is accessed (written via `memset`), and then the entire vector is destroyed, all of which is repeated for a total of 2^{27-n} iterations. Because each mutating operation in this benchmark involves an allocation or deallocation (and all other operations are few), the Density (**D**) is extremely high, and the Variation (**V**), due to the randomly chosen string lengths (greater than 32) is also quite high.

Looking at the data for DS2 (above), we quickly observe that the choice of the underlying allocator mechanism used dominates. First we see that run time of using the default allocator (AS1-AS2), which is roughly the same *irrespective of how it is accessed*, is dramatically more (~75%-575%) than that of any of the local-allocator-based strategies (AS3-AS14). Next we observe that using just a monotonic allocator (AS3-AS6) works best with respect to the run time of the global allocator (~20%, or 5x), followed by a combination of monotonic and multipool allocators (~25%, or 4x), with a multipool allocator alone bringing up the rear (~60%, or 1.7x.), yet all are still significantly and consistently faster than the global allocator. We can also easily observe that there is an abrupt jump in run time (across the board) when the data structure size rises beyond 2^{14} string elements, yet the relative performance of all of the allocation strategies remains roughly the same. Looking more closely, we can see that the effects of accessing each of the allocators directly, versus via a virtual-function interface, makes little or no difference, although there is some slight recurring bias favoring direct access. Finally we note that “winking out” tends to somewhat reduce run time (~1%-9%) – the most pronounced being when a monotonic

allocator is involved and, secondarily when the allocator is accessed via a virtual function.

7.3 DS3, unordered_set<int>

	← global →		← monotonic →				← multipool →				← mono + multi →			
	virtual			(wink)	← virtual →	(wink)		(wink)	← virtual →	(wink)		(wink)	← virtual →	(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^6	10.2	11	5.08	4.88	5.62	5.34	7.16	7.12	7.5	7.2	6.19	5.73	6.4	5.81
2^7	12.5	13.3	5.04	4.81	5.68	5.24	6.37	6.22	6.71	6.31	5.8	5.46	6.08	5.5
2^8	15.8	16.4	4.99	4.79	5.54	5.22	5.95	5.81	6.21	5.92	5.65	5.32	5.82	5.4
2^9	18.3	19	5.01	4.8	5.53	5.18	5.78	5.56	6.01	5.7	5.56	5.2	5.76	5.21
2^{10}	21.4	22.3	4.99	4.83	5.55	5.2	5.72	5.46	5.95	5.55	5.52	5.27	5.68	5.24
2^{11}	25.5	26.1	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}	27.1	28	5.02	4.81	5.55	5.2	6.42	6.1	6.57	6.25	5.51	5.12	5.68	5.27
2^{13}	27.9	28.8	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}	28.5	29	5.03	4.8	5.58	5.26	7.03	6.59	7.18	6.68	5.64	5.19	5.8	5.34
2^{15}	28.3	29.2	5.03	4.78	5.56	5.28	7.11	6.65	7.2	6.83	5.68	5.17	5.78	5.24
2^{16}	31.6	31.8	5.02	4.76	5.6	5.22	6.79	6.37	6.93	6.46	5.68	5.17	5.79	5.24

Table 5: DS3, unordered_set<int>

For DS3, `unordered_map<int>`, we repeated the initial experiment, DS1, on elements of type `int`, but this time substituting `unordered_map` for `vector` as the container type. Although the appended data does not itself involve memory allocation, creating each container node to hold it (absent **bsl**-style internal pooling, which was the case on this platform) does; hence, Density (**D**) for this data set is high, while Variation (**V**) is nil.

Our first observation with respect to the DS3 data (above) is that run time using the global allocator (AS1-AS2) is always the largest, and grows substantially with (physical) data-structure size, while such growth doesn't appear for any of the (local) allocation strategies (AS3-AS14). For this data structure, there is indication that access via a virtual function call (AS2, AS5-AS6, AS9-AS10, AS13-AS14) is typically somewhat slower (~1%-10%) than direct access (AS1, AS3-AS4, AS7-AS8, and AS11-AS12); however, the DS3 data shows consistently that the "winking-out" feature (AS4, AS6, AS8, AS10, AS12, AS14) is a clear win (5%-10%) everywhere that it can be done. Finally, we note that monotonic (alone) AS3-AS6 is the best allocator choice, with direct access and "winking out" (AS4) being the overall best allocation strategy: We observe a runtime improvement (over the global allocator) approaching 700% for larger data structures (e.g., 2^{16} nodes).

7.4 DS4, unordered_set<string>

	← global →		← monotonic →				← multipool →				← mono + multi →			
	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	120	52.2	51.9	52.4	51.2	58.4	57.6	59.7	58.9	55.1	54.1	56.9	55.3
2^7	103	122	52.5	52.1	52.9	51.8	63.3	61.9	64.4	63.8	55.3	54	56.8	55.7
2^8	109	128	53.6	53	53.7	52.6	76.3	74.7	77.4	75.9	56.5	54.9	57.9	56.7
2^9	113	134	54.5	53.4	54.9	53	83.1	81.7	82.8	81.4	57.3	56.7	58	56.4
2^{10}	119	143	56.6	54.9	56.9	54.6	87.6	85.9	88.1	86.5	58.8	56.9	59.2	57.3
2^{11}	122	144	57	55.3	57.7	54.9	90.7	89.2	90.7	88.4	59.4	57.6	60	57.8
2^{12}	122	146	57.9	55.9	58.4	55.7	93.2	90.7	93.2	90.7	60.5	58.3	60.7	58.4
2^{13}	124	148	58.2	56.3	58.5	55.9	95.1	91.5	94.3	92	60.5	58.2	60.7	58.7
2^{14}	139	166	59.1	57.3	59.6	56.8	98.5	94.1	97.8	95.8	61.8	59.6	62.2	60
2^{15}	176	211	66	62.7	66.2	62.4	121	115	122	115	76.5	73.3	76.8	74
2^{16}	196	232	78.5	72	79.1	71	137	127	136	127	87.1	82.4	87.8	82.9

Table 6: DS4, unordered_set<string>

Next, we again used an `unordered_set` as our container, but this time, like DS2, used, as elements, strings of uniformly distributed random length (again in the range of 33 to 1000 to thwart the short-string optimization). This time we have a high Density (**D**) with moderately high (unimodal) Variation (**V**).

The DS4 results largely mirror those of DS3, but with some notable differences. The run time for the global allocator (AS1-AS2) is again substantially larger than that of any local allocator, and grows aggressively with increasing data structure size. On the other hand, that same relative growth is this time reflected in each of the other (local) allocation strategies (AS3-AS12). There is some tendency for access via a virtual-function interface to be slower than direct access, but much less so: ~1% for all local allocators compared to ~20% for the global one. For this data structure, we again see that the monotonic allocator (AS3-AS6) is clearly optimal, and that "winking out" is a consistent win (~1%-10%) across all (local) allocators, the relative runtime benefit of which tends to grow quickly with increasing data structure size. Finally, using a monotonic allocator (alone) and employing "winking out" (AS4 and AS6) were fastest at roughly 2x better than the default global allocator (AS1 and AS2). Note that access via a virtual function (AS6) consistently won out (~2%-4%) over direct access (AS4).

For the remaining eight benchmark scenarios (DS5 – DS8 and DS9 – DS12), each of the (composite) elements correspond, respectively, to the four preceding configurations (DS1 – DS4), and were chosen (arbitrarily) to have 2^7 =128 *leaf* elements (of type either `int` or `std::string`). Each outer container again has 2^n (composite) elements (each of those having 128 *leaf* elements), and is constructed and destroyed 2^{27-7-n} times, for a total of 2^{27} *leaf*-element insertions, as was the case for DS1-DS4. In this way, we keep the total number of operations involving *leaf*

objects across all 12 distinct data structures (DS1 – DS12) in this benchmark comparable (section 6).

7.5 DS5, vector<vector<int>>

	← global →		← monotonic →				← multipool →				← mono + multi →			
	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 7: DS5, vector<vector<int>>

This first composite data structure, `vector<vector<int>>` (DS5) has a low allocation Density (**D**) and a nil requested memory-size Variation (**V**).

The data for DS5 suggest that (1) every local allocator strategy considered is far, far better (~300%-700%) than the global one (AS1-AS2), (2) any runtime differences between virtual-function interface versus direct access are not statistically significant, (3) "winking out" this data structure is typically a relative win (~10%-30%), especially for the most runtime-performant allocator in these tests, namely monotonic (AS3-AS6). In passing, we also observe an across-the-board "platform boundary" in the form of an "elbow" to increasing run time as the size of the outer vector exceeds 2^{14} composite elements (last two rows). Note that this increase is *per leaf element inserted* as precisely the same number of leaf elements are inserted for each row of each table corresponding to each of the twelve experiments in this benchmark.

7.6 DS6, vector<vector<string>>

	← global →		←	monotonic		→	←	multipool		→	←	mono + multi		→
	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	72.6	72.7	9.06	9.06	9.36	8.98	41.7	40	41.2	39.2	11.2	10.3	11.2	10.3
2^7	74.9	76	8.92	8.98	9.29	8.89	46.5	44.8	46	43	11.4	11	12.7	10.3
2^8	85.5	85.2	17.1	17.4	17.3	16.9	62.9	58.4	61.3	58.4	22.8	22.5	23.3	22
2^9	96.4	96.3	18.4	18.7	19	18.4	66.2	59	64.7	59.3	24.2	22.7	24.5	22.3
2^{10}	102	102	18.7	18.6	19.1	18.6	67	59.6	65.9	59	24.8	22.5	24.8	22.5
2^{11}	102	101	18.4	18.7	19.2	18.2	62.4	55	61.3	54.2	24.8	22.6	25.1	22.3
2^{12}	104	103	18.5	18.7	19.4	18.3	61.6	54.2	60.5	53.4	24.9	22.7	25.1	22.3
2^{13}	103	104	18.8	18.4	19	18.6	61.8	53.4	59.9	53.5	25.3	22.6	25.1	22.6
2^{14}	97.1	96.3	19.2	19.6	20.1	19.2	60.6	53.7	60.2	52.9	29	26.7	29.2	26.3
2^{15}	88.1	88.7	23.4	23.2	23.7	23.4	62.6	54.4	60.9	53.9	33.4	30.6	33.2	30.7
2^{16}	76.7	76.7	25	25.3	25.8	25	63.4	54.8	62.9	54.3	35	32.8	35.5	32.4

Table 8: DS6, vector<vector<string>>

Next we consider `vector<vector<string>>` (DS6), which has both a high allocation Density (**D**) and a high Variation (**V**).

The data for DS6 (above) suggests that the default global allocator (AS1-AS2) is the least performant choice, and that direct versus virtual-function access makes no significant difference. The monotonic allocator (AS3-AS6) again proves to be the best allocator choice, but "winking out" doesn't seem to have much of a (consistent) effect for this allocator. Yet "winking out" clearly does exhibit a significant improvement (~5%-15%) when the monotonic allocator is used to back a multipool allocator (AS11-AS14), and especially when used alone (AS3-AS6). We also note that the global allocator (unlike all local allocators) exibited a *reduction* in run time as the outer data-structure size increased beyond 2^{13} composite (`vector<string>`) elements.

Note that there appears to be an across-the-board "platform boundary" when the number of (composite) elements increases from 2^7 to 2^8 where all allocation times – especially the local ones, and particularly those involving a multipool – jump abruptly (~12%-100%). A second "platform boundary" occurs for just the *global* allocator (AS1-AS2) when the number of (composite) elements increases from 2^9 to 2^{10}, where the (per-element) runtime cost plateaus (see rows 2^9-2^{10}). Yet a third "platform boundary" occurs for the 12 *local* allocator strategies (AS3-AS14) when the number of composite elements increases from 2^{13} and 2^{14}, where the (per-element) cost begins to accelerate, and – at the same time – the (per-element) global allocator run times also begin to decrease sharply (see rows 2^{13}-2^{16}).

7.7 DS7, vector<unordered_set<int>>

	← global → virtual		← monotonic → ← virtual →				← multipool → ← virtual →				← mono + multi → ← 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	28.8	28.7	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	28.3	28.5	2.97	2.66	3.36	2.95	4.99	4.44	5.43	4.91	3.2	2.62	3.61	2.97
2^8	28.2	28.1	2.94	2.62	3.33	2.92	5.02	4.53	5.53	4.97	3.23	2.6	3.6	3.01
2^9	31.8	31.7	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}	46.6	47.2	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}	54.3	54.1	2.92	2.61	3.33	2.89	5.63	4.75	5.88	5.37	3.16	2.6	3.61	2.98
2^{12}	54.7	54.8	2.96	2.66	3.34	2.91	6.9	5.79	7.28	6.23	4.15	3.05	4.58	3.4
2^{13}	55.1	56	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	50.9	3.53	2.99	3.81	3.25	7.08	6	7.47	6.46	4.29	3.14	4.71	3.47
2^{15}	44.8	45.4	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}	38.2	38.2	3.58	3.06	3.86	3.3	7.14	6.11	7.58	6.47	4.37	3.18	4.8	3.54

Table 9: DS7, vector<unordered_set<int>>

Then we have `vector<unordered_set<int>>` (DS7), which has a fairly high allocation Density (**D**) and nil Variation (**V**).

The data for DS7 (above) shows that the default global allocator (AS1-AS2) is again, this time by far, the least performant choice, and that direct versus virtual-function access makes no significant difference for the global allocator, but does have a noticable effect for all local allocators (~5%-15%). The best allocator choice in this scenario is again the monotonic allocator (AS3-AS6) but this time *by a factor of almost 20x over the default*. The second most striking observation in this data is the across-the-board improvement (~5%-35%) (for local allocators) of "winking out" the data structure, especially for larger physical sizes, with the largest *percentage* benefit – by far – coming from the composite allocator (AS11-AS14). Notice that, just like DS6, the global allocator's (per-leaf element) run times peak and then recede, whereas the local allocator times tend to grow monotonically and, except between 2^{11} to 2^{13}, very slowly.

7.8 DS8, vector<unordered_set<string>>

	← global →		← monotonic →				← multipool →				← mono + multi →			
	virtual			(wink)	← virtual →	(wink)		(wink)	← virtual →	(wink)		(wink)	← virtual →	(wink)
data size	AS1	AS2	AS3	AS4	AS5	AS6	AS7	AS8	AS9	AS10	AS11	AS12	AS13	AS14
2^{6}	114	116	26	23.8	26.3	24	56.2	54.7	56.9	54.6	27.5	25.8	27.9	26
2^{7}	123	130	26.5	24.4	25.7	23.5	62.7	60.1	62.7	60.5	27.5	26.3	28.2	26.1
2^{8}	162	171	31.7	27.3	32.2	27.8	78	74.2	79.2	73.9	35	32	35.5	32.5
2^{9}	175	181	36.8	28	38.1	28	81.7	74.1	81.2	74.9	36.3	32.1	37.2	32.1
2^{10}	176	183	40	28.9	37.4	28.2	82.1	74.5	82.1	74.7	36.9	32	37.4	32.2
2^{11}	176	183	39.3	28	37.3	28	81.4	74.4	82	74.3	36.9	32.1	37.8	32.1
2^{12}	179	185	39.4	28	37.1	28	81.8	74.1	81.6	74.4	37	32	37.8	32.2
2^{13}	173	178	39.6	27.9	36.9	28.2	81.8	73.6	81.5	74.3	37.2	32	37.8	32.4
2^{14}	157	160	41	29.9	38.8	29.9	81.5	74.1	82.2	74	44	39.3	45.1	39.2
2^{15}	122	131	47.6	35.8	44.8	36.2	85.2	75.5	83.7	76.1	50.5	45.2	51	45.5
2^{16}	95.4	106	51.4	40.5	48.1	38.9	84.8	76.2	88.7	75.9	53.1	48.5	54.8	48.2

Table 10: DS8, vector<unordered_set<string>>

Now we consider the final data structure in this second set of four employing `std::vector` as the outermost container, `vector<unordered_set<string>>` (DS8), which has a high allocation Density (**D**) and a moderately high (unimodal) memory-size Variation (**V**).

The above data for DS8 again shows that the global allocator (AS1-AS2) is the least performant, and that the monotonic allocator by itself (AS3-AS6) is the best choice. Access via a virtual-function-based interface (when compared to direct access) seems to have a consistant overhead for the global allocator (~10%), but not nearly so for the local allocators, especially the monotonic allocator (AS3-AS6), for which run time using a pure abstract base class for data structures having 2^{10} or more (composite) elements was consistently better (~5-7%). "Winking out" is again a relative win (~5%-25%) across all local allocators. Note that the *global*-allocator times (AS1-AS2), much like DS6 and DS7, peak and then recede with data structure size, where as all of the *local*-allocator times (AS3-AS14) above 2^{7} elements are largely monotonically non-decreasing.

We pause here briefly to mention that the detailed raw data presented throughout this paper reflects execution on just a single platform. In preparations for the first revision of this paper (P0089R0), however, we ran these benchmarks using multiple compilers on multiple machine types. An interesting result, the details of which can be viewed online, is that, for the Clang compiler (only), the runtime overhead of accessing via an abstract base class on the hardware platforms we tested was two to three times that of using an allocator directly, but only for the previous four (out of twelve) data structures (DS5-DS8), which have an `std::vector` at the top-level.

We now turn to consider the third and final set of four data structures, each having instead an std::`unordered_set` as the outer-most container.

7.9 DS9, unordered_set<vector<int>>

	← global → virtual		← monotonic → ← virtual → (wink)				← multipool → ← virtual → (wink)				← mono + multi → ← virtual → (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 11: DS9, unordered_set<vector<int>>

The first data structure in our final group of four, `unordered_set<vector<int>>`, has a high allocation Density (**D**), and a nil Variation (**V**).

The data for DS9 (above) again suggest that the global allocator is clearly the least effective choice (~300%-~900%), and that the relative overhead of access via a virtual-function interface (compared to direct access) is quite small (~1%-5%) for the global allocator, and *non-existent* for all local allocators. For this data structure, the best allocator choice again appears to be monotonic (AS3-AS6), however the composite allocator – i.e. a multipool backed by a monotonic allocator (AS11-AS14) is a very close second. Note that the substantial (per-leaf-element) increase in run time (with respect to increasing data-structure size) for the global allocator (AS1-AS2) is not reflected in local allocators employing a monotonic allocator (AS3-6, AS11-AS14). For larger data-structure sizes, there was also a consistent benefit to "winking out" local memory (~2%-30%), especially where a monotonic allocator was involved.

7.10 DS10, unordered_set<vector<string>>

	← global →		←	monotonic	→		←	multipool	→		←	mono + multi	→	
	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	73.2	9.41	9.39	9.34	8.97	41.7	39.7	41.1	39.3	11.2	10.4	11.2	10.3
2^7	74.7	75.3	9.32	9.34	9.24	8.87	46.2	43.7	45.3	44.2	12.7	10.6	11.4	10.8
2^8	83.1	85.4	18	17.3	16.9	17.2	62.2	58.9	61.9	57.6	23.2	22.3	23.1	22.4
2^9	91.4	94.9	19	19	18.8	18.6	65	59.9	64.4	58.9	24.3	22.6	24.1	22.6
2^{10}	98.2	101	19.2	18.9	19.1	18.6	66.5	59.7	65.4	59.1	24.8	22.6	24.6	22.7
2^{11}	99.5	101	19	19.1	19.3	18.4	66.9	59.5	66.1	58.7	24.9	22.7	25.1	22.5
2^{12}	102	105	19.4	19	19.2	18.8	67	58.9	65.8	59.4	25.3	22.6	25.1	22.7
2^{13}	103	104	19	19.2	19.4	18.4	66.7	59.2	66.2	58.2	25.3	22.9	25.5	22.6
2^{14}	95.8	97.2	19.8	20	20.3	19.3	62.8	55.6	61.9	54.3	29.2	26.8	29.6	26.5
2^{15}	87.1	89.8	24	23.7	24	23.5	64.3	55	61.9	54.9	33.6	30.8	33.5	31
2^{16}	77.1	78.2	25.6	25.7	26	25.1	63.9	55.5	63.3	54.5	35.3	33	35.7	32.6

Table 12: DS10, unordered_set<vector<string>>

Next we consider `unordered_set<vector<string>>` (DS10), which has a high allocation Density (**D**) and a moderately high (unimodal) memory-size Variation (**V**).

The results for DS10, `unordered_set<vector<string>>` (above) are, unsurprisingly, not dissimilar for those of DS8, `vector<unordered_set<string>>`. The global allocator is yet again the least efficient choice, and the best choice yet again appears to be the monotonic allocator alone (~300%-600%), with the overhead of non-direct access minimal: ~1%-3% for the global allocator (AS1-AS2), and non-existent for all local allocators (AS3-AS14). The technique of "winking out" the data structure is not as consistant a win for the monotonic allocator alone (AS3-AS6) as it was in DS8, but continues to be so for the other two (less runtime performant) local allocator mechanisms (AS7-AS14).

Interestingly, as with DS8, there appears to be an across-the-board "platform boundary" (i.e., where run times differ sharply) for data structures between 2^7 and 2^8 (composite) elements, and another one, more closely tied to the monotonic allocators (AS3-AS6, AS11-AS14) between 2^{14} and 2^{15} elements. Global allocator times (AS1-AS2) again peak and then recede, whereas all local allocator times (AS3-AS14), for systems above 2^7 composite elements, are again – for the most part – monotonically increasing.

7.11 DS11, unordered_set<unordered_set<int>>

	← global →		← monotonic →				← multipool →				← mono + multi →			
	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	28.7	29.1	3.06	2.75	3.55	3.14	4.96	4.4	5.41	4.84	3.24	2.73	3.73	3.15
2^7	29.1	29	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	28.8	29.1	3	2.68	3.45	3.04	5.18	4.55	5.57	4.98	3.24	2.66	3.65	3.06
2^9	31.8	32.3	2.99	2.64	3.43	2.98	5.12	4.54	5.55	4.95	3.22	2.6	3.65	2.99
2^{10}	46.5	47.1	2.95	2.65	3.4	2.99	5.13	4.57	5.62	4.96	3.21	2.58	3.62	2.97
2^{11}	53.3	53.5	2.94	2.64	3.43	2.96	5.58	4.84	5.75	5.39	3.2	2.63	3.67	3.01
2^{12}	54.6	55	3.02	2.66	3.43	2.98	6.47	5.94	6.99	6.28	3.83	3	4.21	3.38
2^{13}	56.5	56.5	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}	52.1	52.2	3.5	2.99	3.88	3.25	7.35	6.07	7.83	6.59	4.33	3.05	4.76	3.38
2^{15}	45.7	46.2	3.62	2.99	3.95	3.27	7.7	6.39	8.11	6.83	4.43	3.06	4.81	3.44
2^{16}	39.3	39.3	3.72	3.05	4.03	3.31	7.57	6.3	8.09	6.61	4.52	3.1	4.92	3.45

Table 13: DS11, unordered_set<unordered_set<int>>

We next consider `unordered_set<unordered_set<int>>` (DS11), which has a high allocation Density (**D**) and a fairly low (entirely bimodal) memory-size Variation (**V**).

The data for DS11 (above) strongly suggest – even more so than any other data set considered in this benchmark – that the global allocator is *by far* the least effective choice: ~10x-20x slower when compared to the best ones, which in this case is either a monotonic allocator alone (AS3-AS6), or possibly one backing a multipool allocator (AS11-AS14), that "winks out" allocated memory, and provides direct access to the allocator, as opposed to via a base class (i.e., AS4 or AS12). The relative overhead of access via a virtual-function interface (compared to direct access) is negligible (~0%-1%) for the global allocator (AS1-AS2), and somewhat larger (~5%-10%) for all the local allocators (AS3-AS14) – at least on a percentage basis; the maximum absolute runtime overhead, however, remains roughly the same at ~0.5s. The multipool allocator alone (AS7-AS10) was again less effective (~2x) than the other local allocator strategies (AS7-AS14), but still a considerable improvement (~5x-10x) over the global one (AS1-AS2). The relative advantage of "winking out" memory was significant across the board (~10%-45%), especially when a monotonic allocator was involved, and the composite allocator (AS11-AS14) in particular.

We note that the global allocator (AS1-AS2) seemed to hit a "platform boundary" between 2^9 and 2^{11} (composite) elements, where the (per-leaf-element) run time increased dramatically (~50%), before eventually receding (see rows 2^{13}-2^{16}). This anomaly did not appear to be reflected in any of the local allocators (AS3-AS14), although there did appear to a fairly abrupt increase in run time (~10%-%25) for all local allocators, when the data size increased from 2^{11} to 2^{13} (composite) elements.

7.12 DS12, unordered_set<unordered_set<string>>

	← global → virtual		← monotonic → ← virtual →				← multipool → ← virtual →				← mono + multi → ← 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	125	25.9	23.7	26.1	23.9	56.3	54.5	56.7	54.7	27.4	25.8	27.8	26
2^7	141	145	26.4	24.3	25.6	23.4	62.1	59.6	62.5	60	27.9	25.8	28.3	25.8
2^8	165	173	31.5	27.3	32.2	27.7	77.4	73.7	77.8	74.2	34.8	31.9	35.6	32.2
2^9	171	178	35.9	27.6	34.4	27.8	80	73.7	79.7	74.6	35.7	32	36.5	31.9
2^{10}	177	182	38.7	28.6	35.6	27.9	81.1	74.3	81.3	74.3	36.7	31.8	37.1	32
2^{11}	177	183	38.2	27.6	36.2	27.7	81.3	74.3	82.2	74.1	37	32	37.8	31.9
2^{12}	179	186	39.1	27.7	36.5	28	81.6	73.5	81.5	74.1	37.3	31.8	37.9	32.1
2^{13}	165	169	39	27.8	36.7	27.8	81.3	73.9	82.8	73.5	37.3	32.1	38.3	32.1
2^{14}	153	156	40.9	29.6	38.7	29.6	81.5	74.1	82.4	73.7	44.4	39.2	45.4	39.1
2^{15}	122	131	47.6	35.7	44.8	36.1	85.7	75.2	83.9	75.4	51	45.1	51.4	45.5
2^{16}	100	111	51.4	40.4	48	38.8	85.1	75.5	86.2	75.6	53.6	48.4	54.6	48.2

Table 14: DS12, unordered_set<unordered_set<string>>

In this final data structure, we consider `unordered_set<unordered_set<string>>` (DS12), which has a very high allocation Density (**D**) and a moderately low (bimodal) Variation (**V**).

The data for DS12 (above) again suggests that the global allocator (AS1-AS2) is the least runtime performant choice (~300% to 500%) compared with the most performant one, monotonic (AS3-AS6), but not nearly as much so as in the preceding data structure, DS11, where the leaf component was instead of type `int`. There seems to be a "platform boundary" between 2^7 and 2^8 (composite) elements, where run time jumps abruptly (~15%-25%) for all allocators. The overhead of accessing through an abstract base class (versus directly) for the global allocator (AS1-AS2) was generally minimal (~1%-3%), but increased (to ~10%) above another apparent across-the-board "platform boundary" between 2^{14} to 2^{15} (composite) elements, in which the run time of the global allocator *decreased* by ~20%, while the run times of all local allocators *increased* by roughly the same percentage. This unusual trend continued between 2^{15} to 2^{16} elements.

The overhead for accessing local allocators via a base class varied, sometimes more, sometimes less, but, for the monotonic allocator alone (AS3-AS6) for data structures having at least 2^9 elements, the "overhead" was consistently *negative* (~5%-10%) – that is, access via a virtual function was typically faster than direct access. Finally we note that, for *all* local allocators, *"winking out" for this particular data structure was always a very significant win: ~10%-40%.* (Due to unexpected results observed for this specific data structure, we nominate it – in particular – as a prime candidate for further study.)

The DS9-DS12 experiments show similar relative performance to DS5-DS8 for the corresponding allocation strategies. Again, multipool allocators perform significantly better when the leaf-most element type is `int` rather than `string`, while the monotonic allocator retains most of its gains in such cases.

We now make several general empirical observations deliberately avoiding any (naïve) attempts at trying to explain their underlying causes. The most obvious result, looking at the heat-mapped charts, is that monotonic allocators are always a big win, generally giving a speedup in the range of 4x-20x. The relative cost of direct versus abstract-base-class access to allocators (where it exists at all) appears to be mostly in the ~2%-10% range. The multipool allocator appears to provide much less of a gain, although still observable, which is generally in the ~20%-100% range. Note, however, that the multipool allocator seems competitive with the monotonic allocator for the data structures incorporating `unordered_set<int>` (DS7 and DS11), generally offering more than a 5x speedup.

Similarly, the "wink-out" strategy generally offers a modest, but predictable win in most cases, with a particular affinity for combining some form of monotonic allocator with containers incorporating the composite element `unordered_set<T>`. When the type of `T` is `string`, best performance is achieved with a simple monotonic allocator, but when the type of `T` is `int`, best performance is achieved with a monotonic allocator backing a multipool allocator. In the two cases of data structures incorporating the composite element `vector<string>` (DS6 and DS10) with a simple monotonic allocator (AS3-AS6), however, the "wink-out" strategy seems to have no effect, neither positive nor negative.

As previously stated (section 7.8), looking at additional data (available online) from runs using a variety of compilers, operating systems, and hardware platforms, there is an odd effect for Clang specific to data structures DS5-DS8, `vector`s of containers. The time taken to run the benchmark for allocation via an abstract base class is two to three times that of using an allocator directly, although the monotonic allocator dispatched through an abstract base class still handily outperforms the standard (default) allocator by around a factor of 5x-10x (rather than by a factor of x20). N.B., we speculate that the likely effect is that other compilers are doing a better job at devirtualization in these examples. We also note that, at the time these experiments were conducted, devirtualization was an active topic on the Clang development lists.

A second outlier is the Microsoft platform, which shows a much lower benefit than the Unix platforms from applying custom allocation strategies (AS3-AS14), rarely showing more than a doubling of performance. Similarly, data structures featuring containers of `int` appear to pay a runtime cost of ~50%-100% for allocating through an abstract base class compared to using an allocator directly, while data structures of containers of `string` show a runtime overhead of around 5%. Comparing the run time for Microsoft Visual C++ 2015 with the Linux-Intel results, the (containers-of-)containers-of-`string` experiments complete in a similar time, while the (containers-of-)containers-of-`int` experiments complete in around ~10%-25% of the time when run on Windows. The final oddity on Windows is that, for the largest experiment

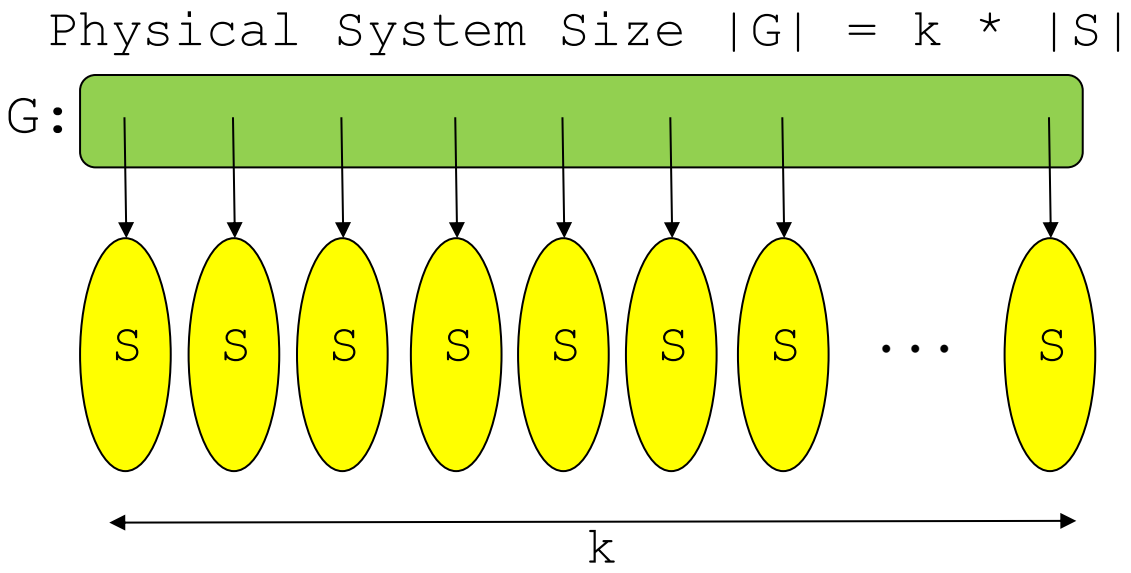
sizes, unless a monotonic allocator is used, the last 3 or 4 rows of many of the tables either fail to complete, or suddenly become excessively expensive, such as taking an hour to run rather than <30 seconds.

Comparing across platforms, the Linux/Power7 results show similar (relative) performance across the benchmarks when using the same compiler. However, the gcc results for the standard (default) allocator are substantially (3x-4x) slower for `vector<int>` and containers of `vector<int>` (DS1, DS5, and DS9). This specific result is not observed when using Clang. However, the poor results for Clang when allocating via a virtual-function interface are even more pronounced on the Power7 platform.

8 Benchmark II: Variation in Locality (Long Running)

Perhaps the most valuable aspects of local ("arena") allocators is that, besides speeding up short-running programs, as demonstrated in the previous benchmark, they keep long–running ones from slowing down over time. All global allocators eventually exhibit *diffusion* – i.e., memory initially dispensed and therefore (coincidentally) accessed contiguously, over time, ceases to remain so, hence runtime performance invariably degrades. This form of degradation has little to do with the runtime performance of the allocator used, but rather is endemic to the program itself as well as the underlying computer platform, which invariably thrives on *locality of reference*.

N.B., *diffusion* should not be confused with *fragmentation* – an entirely different phenomenon pertaining solely to ("coalescing") allocators (not covered in this paper) where initially large chunks of contiguous memory decay into many smaller (non-adjacent) ones, thereby precluding larger ones from subsequently being allocated – even though there is sufficient *total* memory available to accommodate the request. Substituting a pooling allocator, such as the one used in this benchmark (AS7), is a well-known solution to the *fragmentation* problems that might otherwise threaten long-running mission-critical systems.



To demonstrate this common degradation phenomenon resulting from memory *diffusion* across subsystems over time, we created a simple program that acts like a long-running time-multiplexed system, similar in nature to one employing `Boost.Asio`. Given that this experiment is all about *access* Locality (**L**) in a long-running program, the allocation Density (**D**) approaches (is effectively) zero, and the memory-size Variation (**V**), though entirely irrelevant here, happens to be nil as well. The overall system, `G`, will consist of an `std::vector<Subsystem*>` of size `k`, where each subsystem, `S`, is modeled as an `std::list<int>` initially having `|S|` links; hence, `|G| = k * |S|, and the` number of subsystems, `k`, will be the (integral) ratio `|G|/|S|`. At the start of the program, each subsystem, `S`, is `new`-ed in turn, and, when constructed, populates itself with the specified `|S|` links. The system, `G`, is now in its *initial state*.

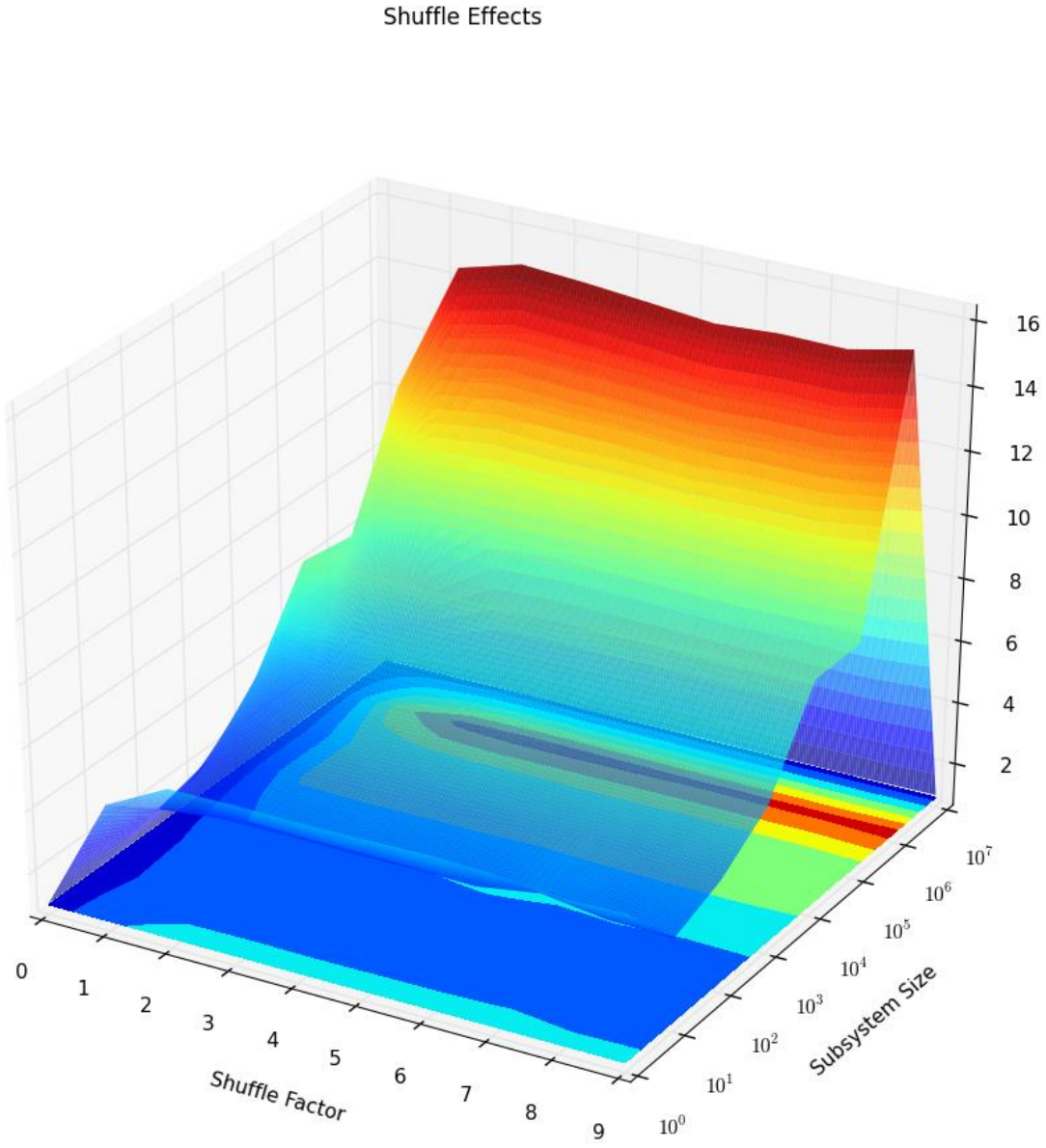
The first experiment is geared towards identifying *opportunities* for the use of allocators – specifically a multipool-allocator-based strategy (e.g., AS7 or AS9) – before actually plugging one in. To that end, we want to contrast the runtime performance of subsystems where memory has been allocated contiguously and then accessed immediately, and where it has been first "shuffled" (which inevitably occurs over time in practice) to be less so, and then similarly accessed. We therefore define a parameter, `sf`, that represents the *shuffle factor*.

Specifying a shuffle factor of 0 leaves the system in its initial sate. A shuffle factor of one (`sf = 1`) means that each `S` (linked list) is visited (in turn) and popped exactly once (from the front), immediately after which a new value is pushed onto (the back of) the list in some randomly chosen `S` in `G`. After each `S` has been visited, this traversal process is repeated until each element in each list has been popped exactly once – i.e., a total of `sf * |G| = 1 * |G|` pop/push operation pairs has occurred. A shuffle factor of two means that the process is repeated until `sf * |G| = 2 * |G|` pop/push operation pairs have been executed (though there is no longer any

assurance that all of the lists still have the same length that they had initially). The larger the shuffle factor, the more non-contiguous and "random" the memory associated with 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 two denotes two complete passes through a subsystem's linked list before moving to the next one in the vector of subsystems comprised by `G`. 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 physical system size, `|G|`, constant.

Keep in mind that this first experiment was done entirely using the default (global) allocator (AS0).



	0	1	2	3	4	5	6	7	8	9
10^7	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
10^6	1.0	11.4	15.6	16.2	16.0	15.8	15.6	15.8	15.7	16.2
10^5	1.0	7.7	7.8	7.8	7.8	8.0	8.0	8.0	8.0	8.2
10^4	1.0	8.0	8.1	8.1	8.1	7.9	8.2	8.2	8.3	8.2
10^3	1.0	5.4	5.4	5.4	5.4	5.7	5.3	5.4	5.3	5.4
10^2	1.0	3.8	4.0	4.1	4.2	4.1	4.2	4.3	4.2	4.2
10^1	1.0	3.4	3.6	3.6	3.6	3.6	3.6	3.9	3.6	3.6
10^0	1.0	4.7	5.7	5.8	5.7	5.8	5.8	5.7	5.5	5.6

Table 15: Shuffle Effects

The graph and corresponding table (above) illustrate the effect of shuffling on an overall problem size of 10^7 links. Each row (back to front in the graph) corresponds to a different system size, where 10^7 (top row) represents a single subsystem of size 10^7 links, and 10^0 (bottom row) represents 10^7 subsystems, each having (on average) just a single link. The columns (left to right in the graph) represent the number of times each linked list (on average) was shuffled.

The element values in the table – and corresponding height of the graph at the various (shuffle factor and subsystem size) coordinates, represents the ratio of access times – after shuffle / before shuffle – once the shuffle times themselves (which should be the same for both) have been, respectively, subtracted. For this initial experiment, the access factor, `af`, was held constant at `10`. Again, keep in mind that all memory accesses so far in this benchmark are via the default allocator (AS0). It will turn out (below) that what's important is whether the memory is *accessed* before (**-**) or after (**+**) it is "shuffled".

Each entry in every row of the table is scaled to the run without prior shuffling (`sf = 0`); hence, column 0 is (by definition) identically `1.0` for each subsystem size `|S|` in the range `[`10^7`..` 10^0`]` (shown, top to bottom, on successive rows in the table). Similarly, when the subsystem size `|S|` is the same as the overall system's size `|G|` (top row), there is no distinction between a local and a global allocator; hence, each of the entries in the top row of the table, corresponding to a single subsystem `S` of size `|S| = |G| =` 10^7 is naturally expected to be `1.0` as well.

Recall that the physical size of each overall system `|G|` is held constant at 10^7 (links), and that the access factor is maintained throughout at 10 (i.e., each linked list of a subsystem, `S`, is accessed sequentially 10 times before moving on to the next subsystem), and that each subsystem is visited, in turn, exactly once, leading to high *temporal* locality, while the *physical* locality varies from low (top row of the table, back edge of the graph) to high (bottom row of the table, front edge of the graph).

The graph was provided to help to visualize the data in the table. What the graph fails to demonstrate, however, is how quickly the shuffling effects take hold before reaching a horizontal asymptote (left to right), after which no additional performance degradation is observed; it turns out that the table makes this specific point much more lucidly.

What the graph does clearly indicate, however, is that the adverse effect of shuffling on memory access times is more pronounced for fewer, larger subsystems (e.g., `|S| =` 10^6) than for many smaller ones (e.g., `|S| =` 10^3). For any given non-zero shuffle factor, the data indicates that the deleterious effects due to memory *diffusion* over the middle of the range of `|S|` are generally increasing with respect to increasing `|S|` – i.e., with decreasing *physical* locality (per subsystem).

Given a demonstrably ample degree of memory "shuffle" (say, `sf = 5`), we next seek to determine more precisely under what specific circumstances locality (*logical* as well as *physical*) within subsystems most adversely affects the relative runtime of accessing memory, and therefore fairly begs for a local allocator.

So far, we have been able to 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 specifies the number of times to traverse the vector of subsystems – each time performing the appropriate number of local accesses as governed by `af`. By keeping the product of the local accesses (`af`) and the subsystem iterations (`rf`) constant (e.g., `af * rf = 256`), we can observe the relative effects of high versus low *temporal locality* for the same number of total accesses. The repeat factor can also be used to increase the run time of the relevant part of the experiment. Note that, for this first revision of the paper we increased this product by an order of magnitude (i.e., to 2,560) in order to reduce noise in the observed results (at the cost of literally weeks for dedicated run time).

If we are to make a fair comparison regarding the relative runtime overhead due to diffuse (i.e., "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 the access occurs before (**-**) or after (**+**) the indicated data access pattern:

(`|G|`, `|S|`, `af`, `sf`, `rf`)

Try to remember that the sign of `sf` (**-/+** -> before/after) applies to the *access*, and not the *shuffle*. (This interface was clearly a horribly bad design, sorry.) One more time: A ***negative*** shuffle factor, **`sf`**, **implies** the **data access** occurs **before** the **shuffle**. (Another, somewhat less arbitrary, way to remember `sf` is that, in terms of run time, *negative* should be less than *positive*.

For additional syntactic convenience, we will also assume that a *negative* global physical size for `|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 32 -3 8`: The global physical system size, `|G|`, is 2^{20}. The initial size of each of the (four) subsystems, `|S|`, is 2^{18}. The number of times the link-list within a subsystem will be traversed (before proceeding to the next one), `af`, is 32. The number of shuffles that will occur ***after*** the data is accessed, `sf`, is 3. The number of times the sequence of subsystems in the overall system, `G`, will be traversed, `rf`, is 8.
- `-20 18 32 +3 8`: Same as above, except that the shuffling of data occurs ***before*** accessing the data (i.e., the access comes after, and is typically slower).
- `-20 18 8 +3 32`: Same as above, except that the number of times each of the subsystems' linked lists is traversed is decreased to only 8 times before moving to the next subsystem, whereas the number of iterations over the sequence of

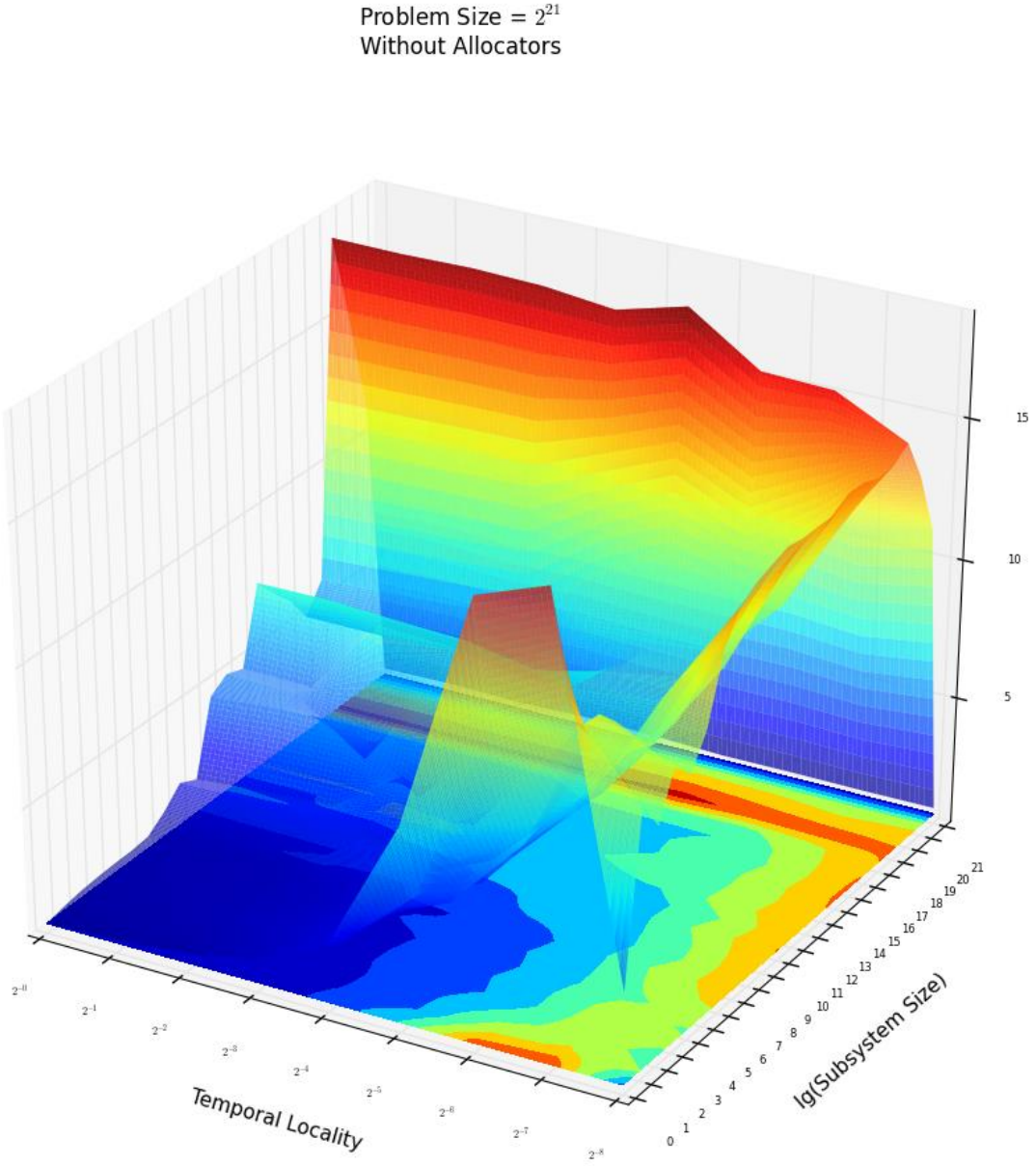
subsystems (comprised by `G)` is increased to 32, thereby reducing temporal locality, while keeping the overall number of memory accesses the same (i.e., `256 * |G| =` 2^{28}).

- `-21 18 8 +3 32`: Same as above, except the overall physical size of the problem, `|G|`, has doubled, yielding eight (2^{21}-2^{8}) subsystems, each of size 2^{8}.
- `-21 19 8 +3 32`: Same as above, except the size of each individual subsystem, `|S|`, has doubled (resulting in half as many subsystem, `k = |G|/|S| = 4`).
- `-21 19 8 +5 32`: Same as above, except the number of times each subsystem is shuffled (before the data is accessed) has increased by two.
- `-21 19 0 +5 32`: Similar to the above in that there are again 32 traversals of the (four) subsystems, however, no accesses are performed (this is how we determine the combined shuffle and traversal runtime costs in calculations, which are then subtracted from the total runtime).
- `-21 19 8 +5 0`: Similar to the above except that there is no traversal of the subsystems (what we could have used to determine just the shuffle, but not the traversal costs, which would have somewhat less accurately reflected the relative runtime costs of pure access).

In order to explore the entire space, we assumed (based on the previously presented data) a constant shuffle factor, `sf`, of 5, and examined a sequence of increasingly large physical problems sizes, `|G|`, contrasting both physical and temporal locality for each. From section 5.3, we conclude that *physical locality* is proportional to the ratio of the number of instructions, **I**, executed within a subsystem to the size of the subsystem, **M** ~`|S|`, holding the number of transitions away from the subsystem, **T**, constant (all with respect to the duration of interest), whereas *temporal locality* is proportional to **I**/**T**, holding **M** constant.

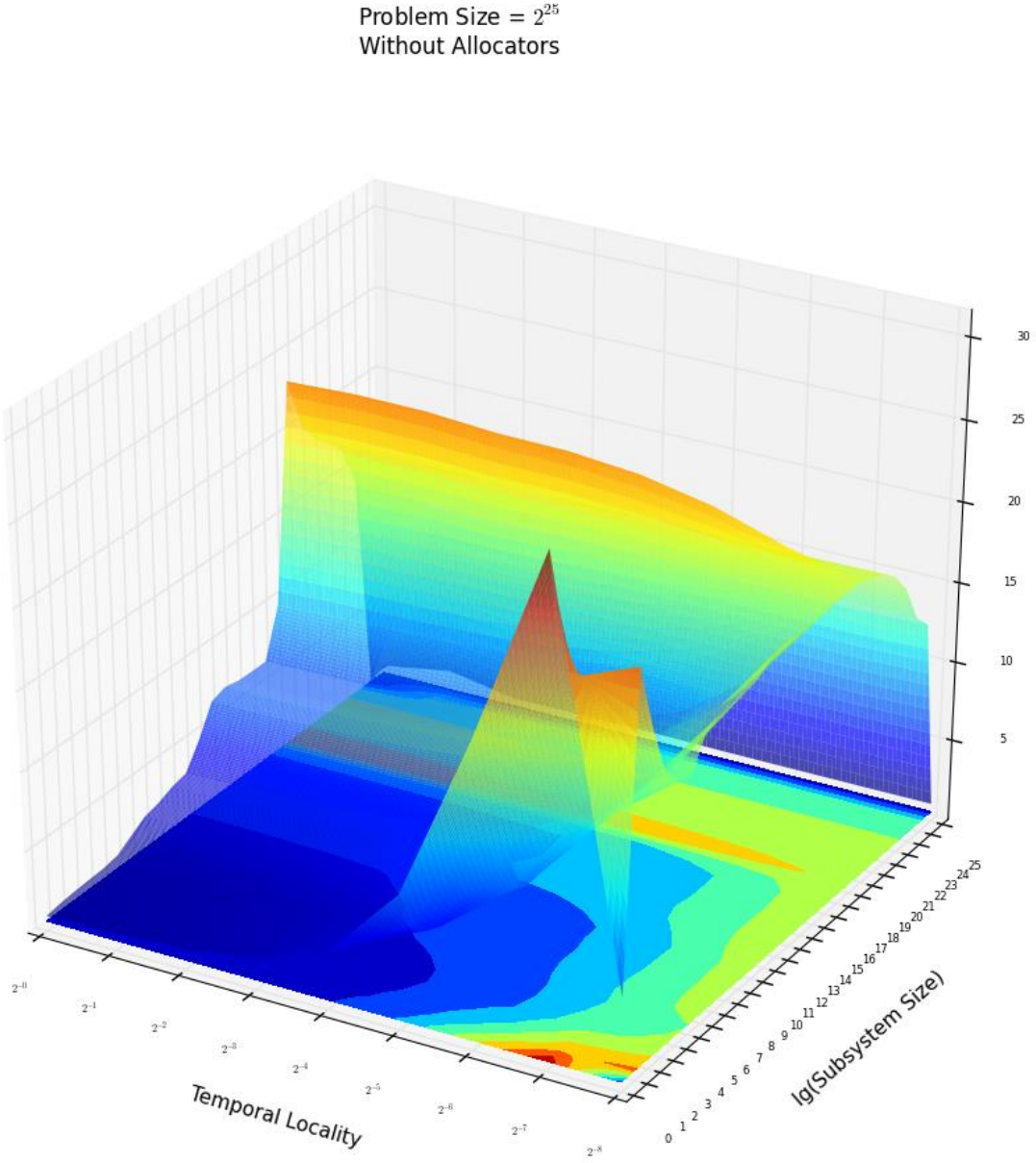
When the size of a problem is sufficiently small, one might reasonably assume that all relevant memory fits in high-speed cache, and there is no need for a local memory allocator. The data we observed bears this hypothesis out. For physical sizes, `|G|`, below 2^{18}, there was no observable benefit for using local allocators on any of the platforms on which we ran this benchmark. Once the problem size exceeds a certain threshold, however, local memory allocators become relevant.

The results of two specific runs of this benchmark, the first of size `|G| =` 2^{21} and the second of size `|G| =` 2^{25} follow. The shuffle factor, `sf`, as discussed above, is held constant at 5, the product of the access factor, `af`, and the repeat factor, `rf`, are held constant at `256 * 10 = 2,560` (varying inversely by powers of 2) and subsystem size, `|S|`, varies (also by powers of 2) from 1 to `|G|`.


Problem Size = 2^{21}
Without Allocators
Temporal Locality
lg(Subsystem Size)

	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	1	1.06	1.01	1.09	1.04	0.96	0.98	0.97	0.99
20	11.3	11.5	11.3	11.4	11.2	11.3	11.3	11	11.6
19	14.8	15	14.7	14.8	14.7	14.8	14.3	13.1	13.8
18	18	18	18	17.9	17.7	18.4	16.7	16.6	15.4
17	6.04	6.17	6.3	6.51	8.64	9.95	9.17	11.5	15
16	5.07	5.07	5.13	5.19	7.16	7.24	7.52	10.8	14.9
15	6.08	6.08	6.15	6.05	5.37	7.3	7.72	10.4	15.2
14	6.77	6.81	6.78	6.67	6.25	7.23	7.73	10.9	15.2
13	7.55	7.59	7.46	7.36	6.92	7.51	7.99	10.8	14.9
12	4.82	4.79	7.7	7.6	7.08	7.26	7.55	11.4	14.9
11	5.05	4.99	3.21	6.66	6.23	5.85	6.27	9.83	14.9
10	4.65	4.87	4.93	2.92	5.71	5.99	6.15	10.7	15
9	2.01	2.23	2.38	4.15	3.03	6.14	6.18	9.67	14.8
8	2.32	2.4	2.6	2.08	3.63	4.86	6.01	9.25	14.6
7	1.68	1.75	1.92	2.36	2.3	3.51	6.12	10.5	14.2
6	1.22	1.31	1.44	2.06	2.76	4.18	6.16	9.93	13.2
5	1.15	1.24	1.39	1.75	2.4	3.45	6.35	9.5	10.9
4	1.13	1.23	1.37	1.72	2.53	4.05	6.6	11	9.77
3	1.1	1.19	1.37	1.72	2.55	3.66	6.42	11.6	10.5
2	1.04	1.14	1.36	1.79	2.43	4.61	8.51	11.7	8.91
1	0.93	1.06	1.26	1.66	2.55	4.86	11.6	12.9	10
0	0.78	0.9	1.1	1.61	2.88	7.75	16.2	17.2	4.06

Table 16: Problem size 2^{21}, without allocators


Problem Size = 2^{25}
Without Allocators
Temporal Locality
lg(Subsystem Size)

	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	0.97	1.72	0.98	1.02	1.04	1.00	1.00	1.00	1.01
24	0.5	12.71	12.81	13.02	13.08	13.02	13.01	12.95	13.11
23	14.2	14.03	14.10	14.07	14.14	14.05	14.07	14.13	14.14
22	16.7	16.71	16.50	16.55	16.49	16.62	16.55	16.56	16.53
21	17.8	17.90	17.87	17.72	17.85	17.83	17.83	17.76	17.89
20	18.6	18.60	18.52	18.54	18.45	18.64	18.43	18.62	18.66
19	20.1	20.16	19.96	19.85	19.80	19.64	19.25	19.12	18.93
18	23.4	23.54	23.51	23.20	23.13	22.72	21.85	20.45	19.34
17	9.81	10.00	10.06	10.29	10.69	11.53	13.01	15.53	19.30
16	6.81	6.87	6.98	7.21	7.70	8.64	10.41	13.75	19.33
15	6.8	6.88	7.00	7.17	7.66	8.63	10.38	13.66	19.32
14	6.82	6.86	7.00	7.20	7.66	8.56	10.39	13.66	19.18
13	7.03	7.08	7.19	7.39	7.85	8.79	10.56	13.77	19.15
12	6.72	6.75	6.90	7.12	7.62	8.52	10.21	13.62	19.30
11	4.86	4.92	5.07	5.35	5.88	6.92	9.01	13.00	19.16
10	3.36	3.49	3.71	4.07	4.69	5.91	8.25	12.32	18.43
9	3.15	3.29	3.51	3.86	4.54	5.87	8.25	12.37	18.28
8	2.76	2.89	3.13	3.54	4.33	5.66	8.12	12.43	18.16
7	2.52	2.66	2.96	3.45	4.25	5.67	8.16	12.65	18.10
6	1.94	2.14	2.49	3.03	3.91	5.45	8.02	12.72	17.64
5	1.34	1.49	1.78	2.33	3.24	4.79	7.54	12.41	15.84
4	1.17	1.28	1.51	1.96	2.83	4.39	7.39	13.22	16.25
3	1.12	1.24	1.45	1.89	2.73	4.30	7.85	14.74	17.32
2	1.06	1.19	1.43	1.90	2.71	4.70	9.98	18.32	20.54
1	0.97	1.11	1.36	1.78	2.91	5.68	13.74	22.91	24.73
0	0.82	0.97	1.22	1.88	3.50	8.30	18.92	30.99	5.68

Table 17: Problem size 2^{25}, without allocators

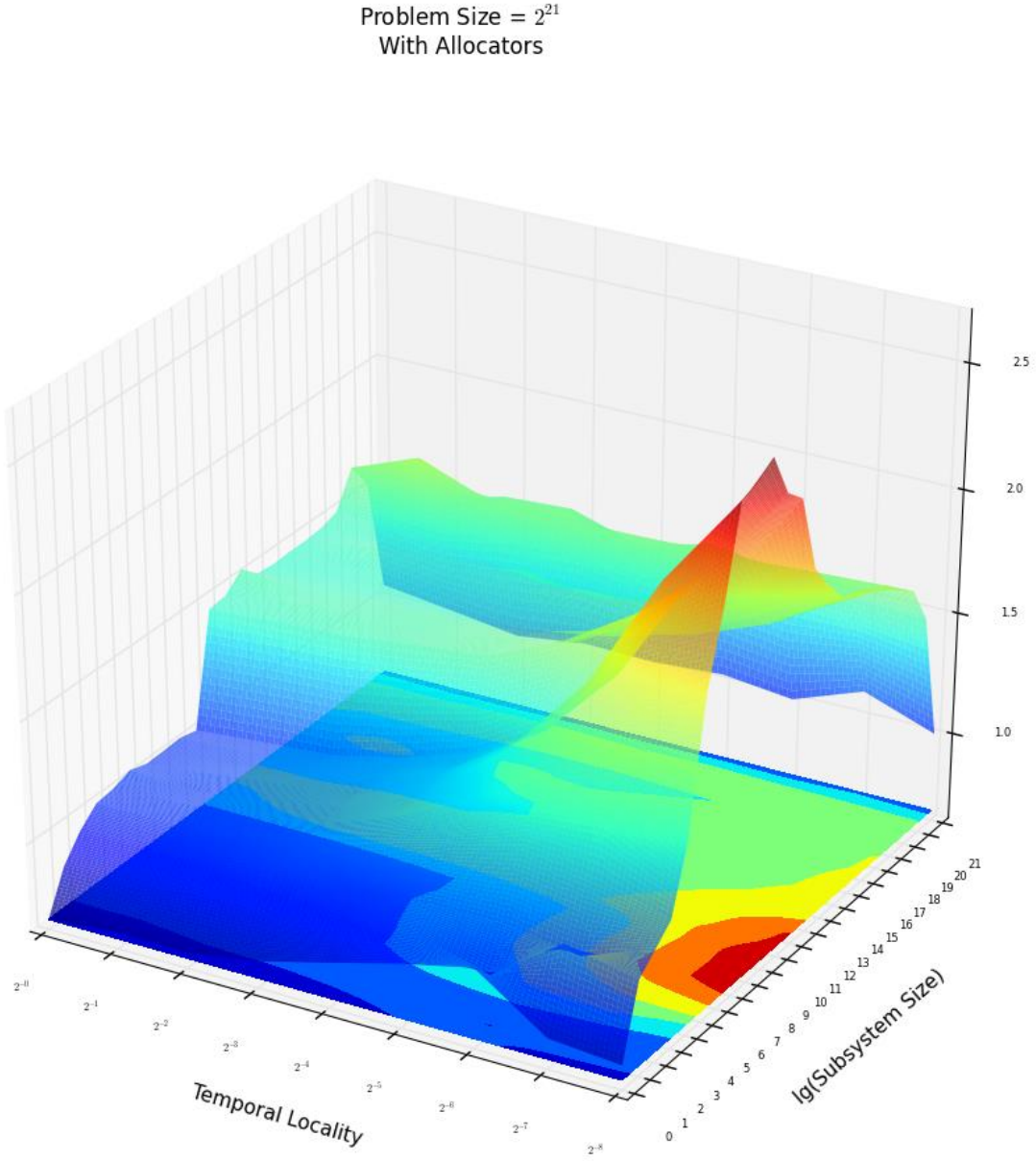
Each of these two runs (above) clearly shows that the greatest opportunity for effective use of local memory allocators occurs when subsystem size, `|S|`, is relatively (but not maximally) large – i.e., physical locality is low (as shown near the back of the graph, top of the table), and quickly tapers off (towards the front, bottom, respectively) with reduced subsystem size (i.e., increasing physical locality). On the other hand, when temporal locality is minimal (right side), the opportunity for significant performance improvement using local allocators spans a much wider range of subsystem sizes as evidenced by the impressively high ratio values (~10x-20x) observed near the extreme right of the graphs/tables.

The graphs based on data sets for system sizes of 2^{21} and 2^{25}, using just the global allocator (AS0), are reminiscent of the middle of the process of inflating a hot-air balloon: The area of low *temporal* locality (towards the right) and low *physical* locality (towards the back) is fully inflated, while the area of higher *temporal* locality (towards

the left), and higher *physica*l locality (towards the front) is only partially so. We assert that, the greater the value in the table (depicted as the vertical height of the surface), the more opportunity there is for a *local* allocator to be useful at improving runtime performance by preserving *access* Locality (**L**).

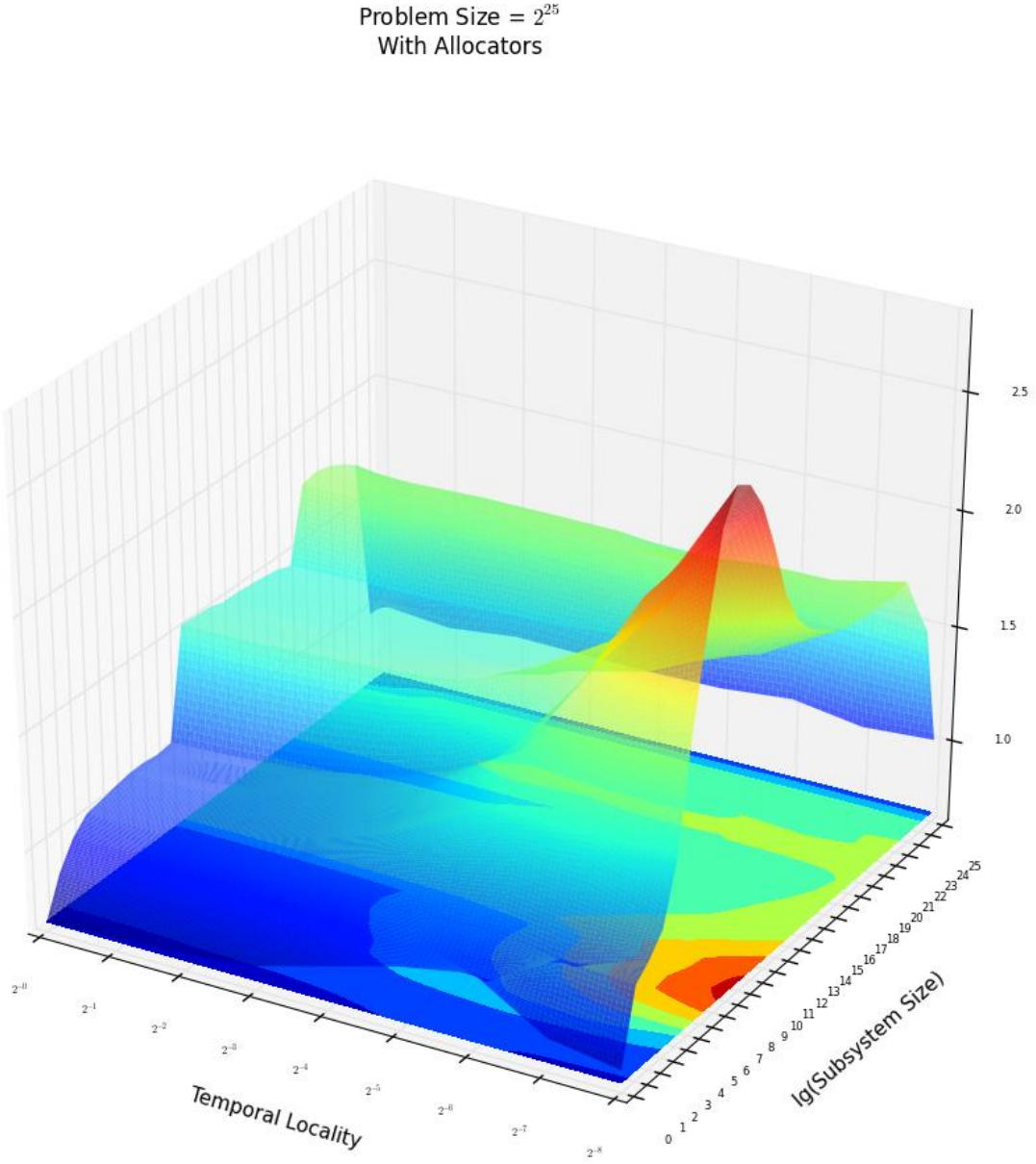
We see, however, that there are some anomalies with the data that we are, so far, unable to explain. In particular, the entire family of graphs we have looked at shows an unexpected spike when the temporal and especially the physical locality are pathologically low. In particular, we are unable to explain why the input parameters `|S| = 1` and `rf = 20` produce such a disproportionately high result value, seen in both runs, and then just one step to the right (lower locality) produces such an unexpectedly low one. Given that this is a pathological "corner" of the graph – i.e., a subsystem, `S`, consisting of just a single link accessed just twice before a context switch to another subsystem versus a similarly tiny subsystem accessed exactly once – we do not feel that these results, although reliably repeatable, impact the validity of our overall conclusions, but clearly they warrant further investigation.

Now, suspecting that (and where) there may be substantial opportunities for runtime improvements, we re-ran benchmark for the two example system configuration sizes (2^{21} and 2^{25}) above, but this time providing each subsystem, `S`, with its own local multipool-based allocation strategy (AS7) used directly and without "winking out" the remaining data. The results are compelling: Providing a local allocator uniformly kept degradation below a factor of three, and – in almost all cases – well below a factor of two! Compare these results with degradations shown in the previous pair of graphs (and corresponding tables) reflecting the increased run times with no local allocator, which often exceeded an order of magnitude!


Problem Size = 2^{21}
With Allocators
Temporal Locality
lg(Subsystem Size)
2.5
2.0
1.5
1.0

	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	1.06	1.02	0.98	1.02	1.02	1.04	1	1.11	1.01
20	1.53	1.52	1.62	1.63	1.55	1.63	1.58	1.54	1.53
19	1.65	1.75	1.65	1.65	1.65	1.66	1.66	1.68	1.69
18	1.51	1.49	1.46	1.42	1.43	1.47	1.52	1.66	1.75
17	1.48	1.48	1.48	1.51	1.48	1.54	1.59	1.65	1.81
16	1.48	1.52	1.49	1.5	1.55	1.56	1.56	1.67	1.82
15	1.48	1.48	1.48	1.49	1.51	1.55	1.6	1.69	1.88
14	1.47	1.48	1.48	1.49	1.5	1.54	1.61	1.72	1.9
13	1.48	1.49	1.5	1.5	1.53	1.58	1.66	1.79	1.99
12	1.54	1.51	1.54	1.55	1.57	1.65	1.72	1.91	2.11
11	1.48	1.53	1.53	1.55	1.6	1.65	1.82	2	2.42
10	1.47	1.49	1.51	1.54	1.57	1.7	1.88	2.11	2.49
9	1.02	1.04	1.06	1.13	1.22	1.39	1.69	2.14	2.67
8	1.03	1.05	1.08	1.13	1.22	1.42	1.73	2.18	2.62
7	1.03	1.05	1.09	1.14	1.24	1.43	1.75	2.22	2.59
6	1.03	1.06	1.09	1.12	1.24	1.44	1.72	2.08	2.23
5	1.05	1.03	1.08	1.13	1.22	1.38	1.61	1.84	1.94
4	1.02	1.04	1.06	1.11	1.21	1.35	1.53	1.63	1.43
3	1.01	1.01	1.03	1.07	1.15	1.21	1.29	1.25	1.17
2	0.95	0.95	0.97	1.01	1	1.04	1.04	0.99	1.1
1	0.85	0.86	0.89	0.9	0.93	0.97	0.89	1.1	1.04
0	0.68	0.71	0.71	0.75	0.83	0.91	0.97	0.77	0.74

Table 18: Problem size 2^{21}, with allocators


Problem Size = 2^{25}
With Allocators
Temporal Locality
lg(Subsystem Size)

	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	1.00	0.97	1.01	1.00	1.00	1.01	1.04	0.99	1.01
24	1.00	1.54	1.57	1.55	1.55	1.53	1.55	1.56	1.53
23	1.71	1.67	1.70	1.69	1.68	1.68	1.68	1.69	1.67
22	1.75	1.75	1.76	1.76	1.75	1.72	1.76	1.76	1.83
21	1.79	1.78	1.78	1.80	1.79	1.74	1.80	1.80	1.80
20	1.80	1.80	1.80	1.81	1.81	1.82	1.81	1.81	1.82
19	1.79	1.78	1.79	1.80	1.79	1.80	1.80	1.82	1.82
18	1.47	1.47	1.47	1.49	1.50	1.53	1.58	1.67	1.83
17	1.49	1.49	1.49	1.50	1.51	1.54	1.59	1.67	1.84
16	1.50	1.50	1.53	1.51	1.53	1.55	1.61	1.70	1.88
15	1.51	1.51	1.51	1.52	1.53	1.56	1.63	1.74	1.92
14	1.51	1.51	1.52	1.52	1.54	1.58	1.65	1.78	1.97
13	1.51	1.52	1.52	1.53	1.55	1.60	1.67	1.82	2.05
12	1.53	1.54	1.54	1.56	1.59	1.64	1.74	1.92	2.20
11	1.54	1.54	1.55	1.57	1.60	1.67	1.84	2.08	2.43
10	1.54	1.55	1.56	1.58	1.61	1.74	1.93	2.25	2.63
9	1.07	1.08	1.11	1.16	1.26	1.44	1.87	2.22	2.76
8	1.06	1.10	1.12	1.18	1.27	1.47	1.85	2.29	2.80
7	1.07	1.06	1.12	1.17	1.28	1.48	1.82	2.32	2.67
6	1.07	1.08	1.10	1.16	1.26	1.46	1.75	2.13	2.31
5	1.05	1.06	1.09	1.14	1.23	1.40	1.62	1.86	1.93
4	1.04	1.05	1.07	1.12	1.22	1.38	1.54	1.65	1.44
3	1.02	1.03	1.05	1.08	1.15	1.23	1.30	1.29	1.20
2	0.96	0.97	0.99	1.02	1.02	1.04	1.05	1.00	1.12
1	0.85	0.86	0.89	0.90	0.94	0.99	0.90	1.08	1.06
0	0.69	0.70	0.72	0.75	0.84	0.92	0.98	0.79	0.74

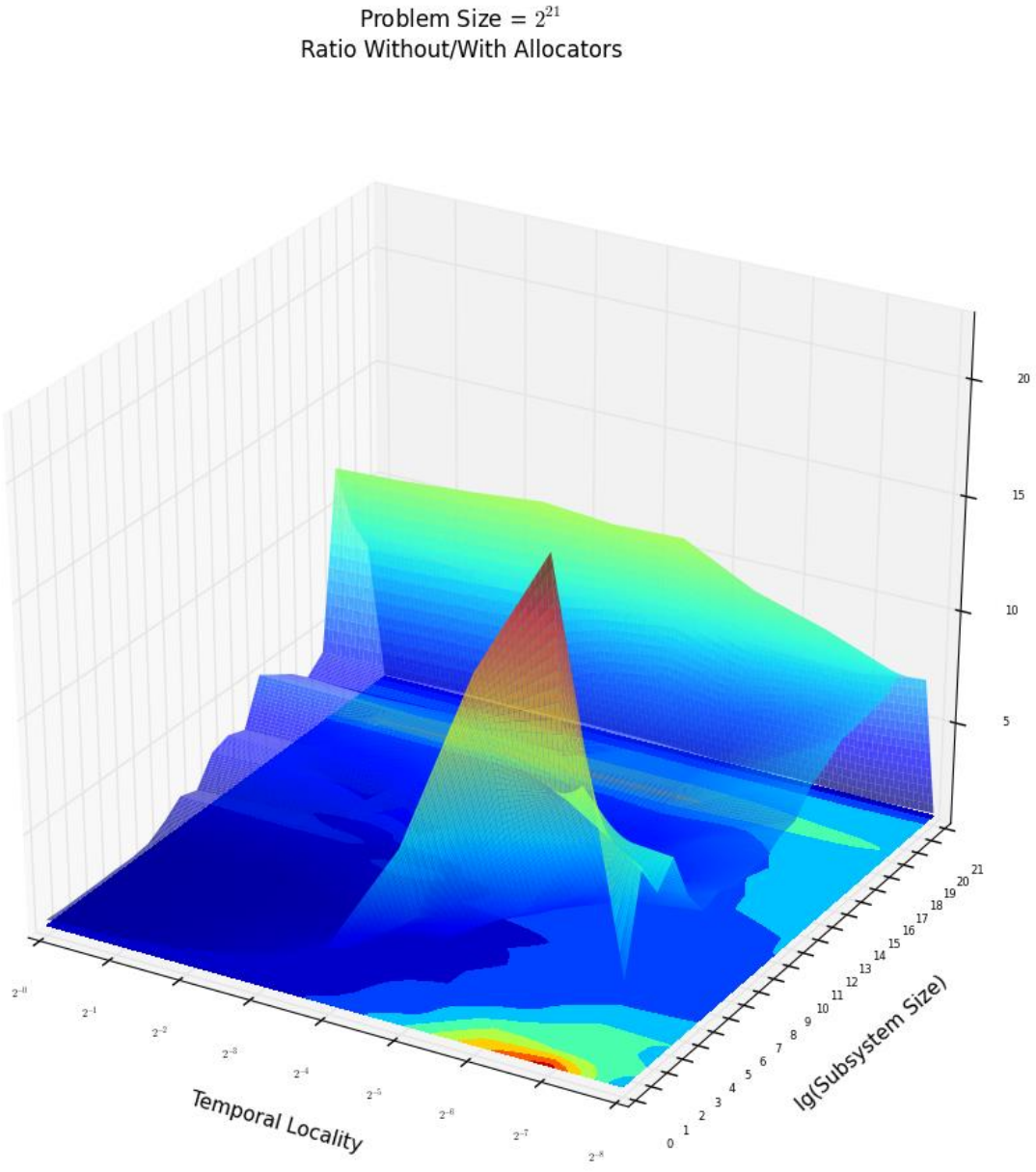
Table 19: Problem size 2^{25}, with allocators

Let's stop for a moment and take a closer look at the data presented in the graphs and tables above. The first thing to note is that the shape of the graphs is strikingly similar across the family of experiments based on overall system size `|G|`, leading us to believe that the remarkable salutary effects of local allocators to preserve locality are both robust and systemic. Whether or not we have local allocators, we observe that runtime performance degrades (albeit much more slowly) with decreasing temporal locality, but with allocators, seems to be more pronounced at the upper-mid ranges (low-mid rows) of physical locality, rather than the lower-mid range (upper-mid rows) without them.

It bears repeating that we've run these benchmarks on a variety of popular platforms (hardware and compilers) for a substantial range of problems sizes, and the results for this benchmark are astonishingly consistent. We conjecture that this consistent (dramatic) loss in runtime performance occurs because the efficiency with which the

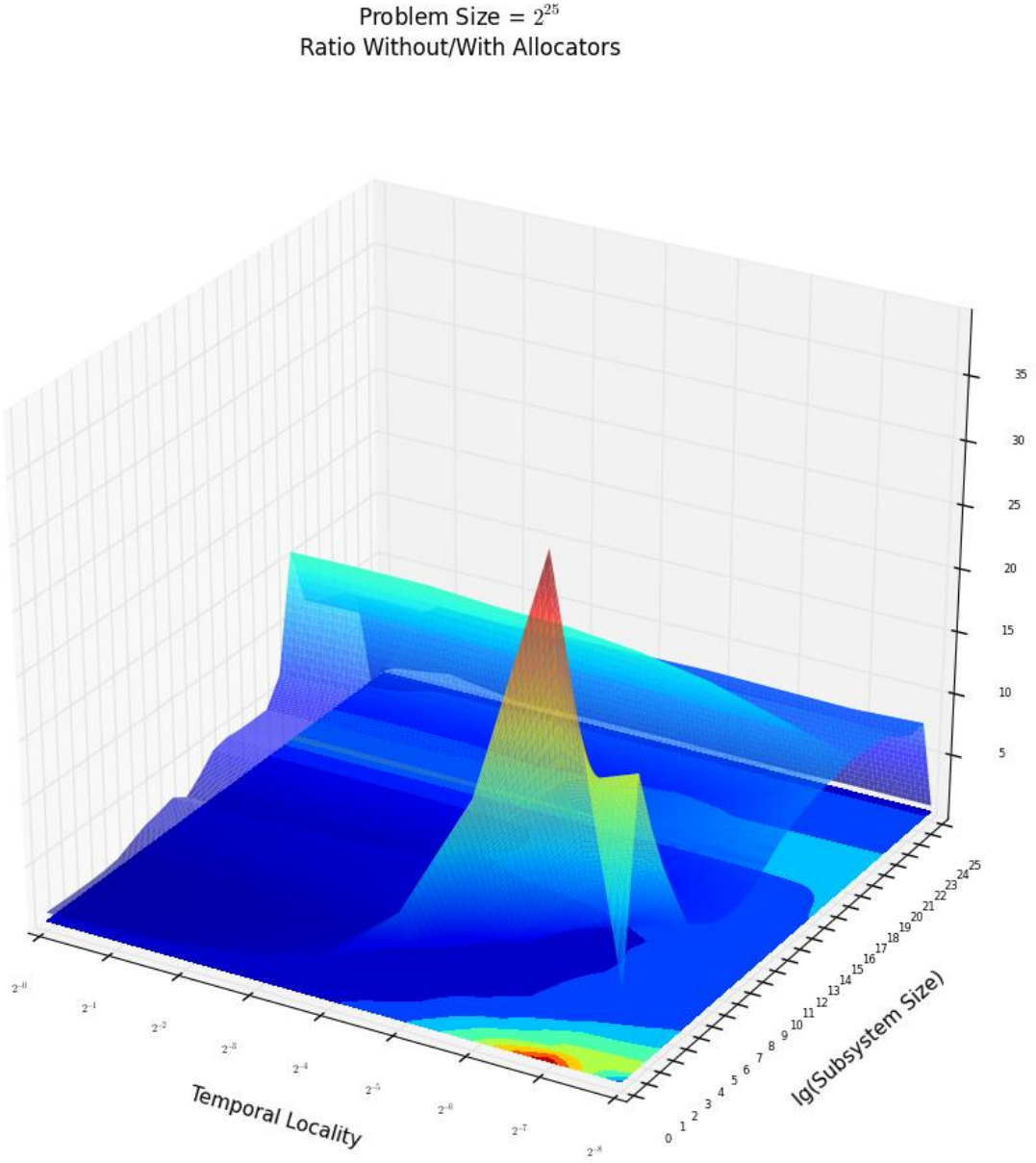
allocator yields memory along with underlying processor speeds are entirely inconsequential when compared to the latency resulting from a profound lack of access Locality (**L**).

Finally, we would like to provide a road map identifying where the use of local (arena) allocators is most indicated. To that end, we plotted, for each (temporal, physical) coordinate in the proceeding graphs, the ratios corresponding to a subsystem that does not employ a local allocator to one that does. The larger the value, the more relative benefit there is to having a local allocator. Even a cursory inspection of the data below shows the spectacular opportunities to recover lost runtime performance due to the *diffusion* of memory across subsystems over time.



	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
21	0.95	1.03	1.03	1.07	1.02	0.92	0.97	0.87	0.98
20	7.4	7.56	6.97	6.96	7.27	6.94	7.15	7.1	7.56
19	9.02	8.56	8.95	8.99	8.92	8.92	8.61	7.82	8.17
18	11.9	12.1	12.3	12.6	12.4	12.5	10.9	10	8.8
17	4.07	4.16	4.26	4.31	5.85	6.48	5.78	6.97	8.27
16	3.43	3.34	3.45	3.45	4.61	4.63	4.82	6.49	8.19
15	4.11	4.11	4.16	4.06	3.56	4.71	4.84	6.14	8.09
14	4.59	4.61	4.59	4.47	4.15	4.69	4.81	6.34	8.02
13	5.11	5.08	4.96	4.9	4.52	4.76	4.82	6.03	7.47
12	3.12	3.16	5.01	4.89	4.51	4.41	4.4	5.96	7.08
11	3.41	3.27	2.09	4.31	3.9	3.54	3.45	4.91	6.15
10	3.16	3.26	3.27	1.89	3.63	3.53	3.28	5.08	6.04
9	1.98	2.15	2.24	3.68	2.48	4.43	3.66	4.51	5.56
8	2.24	2.29	2.42	1.83	2.97	3.43	3.48	4.24	5.59
7	1.64	1.67	1.77	2.07	1.85	2.45	3.49	4.7	5.49
6	1.19	1.24	1.32	1.83	2.23	2.91	3.59	4.78	5.91
5	1.1	1.2	1.29	1.55	1.97	2.5	3.96	5.17	5.61
4	1.11	1.18	1.3	1.55	2.09	3	4.31	6.73	6.82
3	1.09	1.18	1.33	1.6	2.23	3.02	4.99	9.3	9.03
2	1.1	1.21	1.4	1.76	2.42	4.43	8.19	11.8	8.09
1	1.09	1.24	1.41	1.85	2.73	5	13	11.7	9.7
0	1.14	1.26	1.56	2.14	3.47	8.54	16.7	22.4	5.46

Table 20: Problem size 2^{21}, Ratio


Problem Size = 2^{25}
Ratio Without/With Allocators
Temporal Locality
lg(Subsystem Size)

	2^{-0}	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}	2^{-6}	2^{-7}	2^{-8}
25	0.97	1.77	0.97	1.02	1.04	0.99	0.96	1.01	1
24	0.5	8.26	8.17	8.38	8.43	8.52	8.4	8.3	8.57
23	8.3	8.41	8.3	8.35	8.4	8.36	8.36	8.37	8.44
22	9.53	9.57	9.4	9.43	9.41	9.65	9.42	9.41	9.04
21	9.99	10	10	9.87	9.98	10.2	9.9	9.88	9.93
20	10.4	10.3	10.3	10.2	10.2	10.2	10.2	10.3	10.3
19	11.2	11.3	11.2	11	11	10.9	10.7	10.5	10.4
18	15.9	16	16	15.6	15.4	14.8	13.9	12.3	10.6
17	6.58	6.7	6.73	6.87	7.06	7.5	8.19	9.3	10.5
16	4.53	4.57	4.56	4.76	5.04	5.56	6.47	8.11	10.3
15	4.51	4.56	4.62	4.73	4.99	5.52	6.38	7.86	10.1
14	4.51	4.54	4.62	4.72	4.98	5.43	6.28	7.7	9.75
13	4.64	4.67	4.73	4.82	5.07	5.49	6.31	7.56	9.34
12	4.38	4.38	4.48	4.58	4.8	5.18	5.86	7.1	8.78
11	3.16	3.19	3.27	3.42	3.67	4.15	4.9	6.25	7.88
10	2.18	2.25	2.37	2.58	2.91	3.39	4.28	5.48	7
9	2.95	3.04	3.17	3.32	3.59	4.08	4.42	5.57	6.63
8	2.6	2.62	2.8	3	3.4	3.84	4.4	5.42	6.5
7	2.36	2.51	2.65	2.95	3.31	3.83	4.49	5.45	6.79
6	1.82	1.99	2.26	2.62	3.1	3.74	4.59	5.97	7.62
5	1.27	1.4	1.64	2.05	2.64	3.42	4.65	6.67	8.21
4	1.13	1.22	1.42	1.76	2.32	3.18	4.79	8.02	11.3
3	1.09	1.21	1.39	1.75	2.37	3.49	6.03	11.4	14.5
2	1.11	1.22	1.45	1.86	2.66	4.51	9.49	18.3	18.4
1	1.14	1.29	1.54	1.98	3.1	5.76	15.3	21.2	23.4
0	1.19	1.38	1.7	2.5	4.18	8.99	19.3	39.2	7.66

Table 21: Problem size 2^{25}, Ratio

The graphs and tables above help to illustrate where the "sweet spots" for local allocator usage in this benchmark reside. This data confirms that the use of local allocators is not particularly indicated when both the *physical* and *temporal* locality is high, but are especially effective when either the *physical locality* is low (but not completely minimal) or whenever the *temporal locality* is low (especially minimal), yet both graphs indicate that there is bit of a lull in opportunity for the lower-mid-range of subsystem size, `|S|`, in the presence of low temporal locality.

For practically relevant scenarios (e.g., where the subsystem size, `|S|`, is at least, say, 2^{12}), the improvement factor is almost always at least ~4x-8x, sometimes ~8x-12x, and occasionally even ~12x-16x or more. This data, we argue, provides compelling evidence that local allocators make a substantial difference in important practical use cases.

9 Benchmark III: Variation in Utilization

This benchmark was designed to demonstrate the effect of memory Utilization (**U**) – that is, the maximum fraction of the "total" amount of allocated memory "actively" in use at any one time (section 5.4) – on runtime performance. To that end, memory was allocated in chunks, of size `S`, until a first threshold was reached – the amount of active memory, `A`, to use at one time. Then, a chunk was deallocated and another one allocated until the desired total amount of allocated memory, `T`, was reached. After every allocation, the value at the first byte of the allocated memory was incremented (to deliberately access it). The data collected represents a wide variety of values for `A`/`T` – the definition of Utilization (**U**). Note that, since almost no other work is done, the Density (**D**) of this benchmark's allocations (section 5.1) is extremely high, and the memory-size Variation (**V**) is nil.

The three size parameters `T`, `A`, and `S` are measured in bytes. The results in each row are normalized to the result for AS1. Specifically, the results in the AS1 column are times in seconds, and the values in the columns for the other allocator strategies tested – namely AS2, AS3, AS5, AS7, AS9, AS11, and AS13 – each represent a percentage of the AS1 value, where 100 would imply the same run time as that for AS1, and lower values imply shorter ones.

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
30	15	10	0.063s	103	440	435	46	43	46	47
30	16	10	0.069s	102	401	395	42	42	41	45
30	17	10	0.064s	110	435	428	46	44	47	46
30	18	10	0.063s	102	440	434	46	39	54	47
30	19	10	0.063s	104	439	434	51	46	47	47
30	20	10	0.064s	110	433	430	46	42	46	52
30	20	11	0.035s	125	758	747	54	37	49	37
30	20	12	0.022s	101	1216	1206	51	31	52	32
30	20	13	0.013s	60	1985	1961	110	67	1996	1979
30	20	14	0.008s	77	3356	3304	110	58	3276	3314
30	20	15	0.004s	74	5985	6288	60	111	6016	6057

In order to better understand the data provided by this benchmark, let's take a closer look at the table above. The total amount of memory allocated (`T`), for each row in this table, is 2^{30} bytes. In the first row, the maximum amount of memory allocated at once

(A) was chosen to be 2^{15} bytes, and the size of each allocated block (S) to be 2^{10} bytes. What this means is that 2^5 blocks (each 2^{10} bytes) will be allocated initially (bringing the initially allocated memory to 2^{15} bytes; then, a block of the same size (S) will be deallocated and then immediately reallocated $(2^{30} - 2^{15})/2^{10}$ times. Finally, all of the 2^5 remaining allocated blocks (each of size 2^{10} bytes) will be deallocated (individually).

Using the default allocator (AS1) on this platform caused the operations indicated above to run in 0.063 seconds. Allocating using the same new-delete allocator via an abstract base class (AS2) took 3% longer than when that same allocator was used directly (AS1).

Next we tried using a monotonic allocator directly (AS3), and it took 440% of the time that the baseline allocator strategy (AS1) took (or 0.277s). Recall that a monotonic allocator doesn't release freed memory back to the system, and thus is easily demonstrated as being ill-suited to this kind of usage scenario.

We have chosen not to consider allocation strategies AS4, AS6, AS8, AS10, AS12, and AS14 because the "winking out" aspect, which each of the aforementioned strategies incorporates, when eventually applied to the comparatively small amount of remaining memory (A) out of a total (T) – even if it makes the release cost absolutely free – could not possibly (i.e., mathematically) make any meaningful difference in overall run time.

Then we employed allocation strategy AS5, which uses the same (monotonic) allocator used in AS3, but this time accessed via an abstract base class. The runtime cost is 435% of the reference allocation strategy (AS1), which happens to be just a tad less than direct use of the monotonic allocator (perhaps suggesting that – on this platform, at least – the use of virtual functions to perform the allocations were successfully elided by the compiler).

Next we used allocation strategy AS7, which employs a mulitpool allocator directly. Here we see that the runtime cost drops precipitously to just 46% of what the default allocator affords. AS9, the indirect use of this same allocator (via an abstract base class) is comparable at 43% (again suggesting that there is no penalty here for non-direct access).

Then we applied allocation strategy AS11, which employs a multipool allocator (accessed directly), backed by a monotonic allocator. The cost, relative to the baseline (AS1), shakes out at 46% and, when accessed indirectly (AS13), 47% – again no apparent statistically significant overhead with virtual-function-based access.

In subsequent rows, we first increased the size of (A) from 2^{15} to 2^{20}, and then (S) from 2^{10} to 2^{15}. The data speaks for itself, but we will make just a few observations:

(**1**) We believe the behavior of the AS1 column makes sense in that the runtime work done while we are increasing the allocation limit (A) remains fairly constant, while the work done as we increase the block-size (S) decreases proportionally.

(**2**) The AS2 column tends to indicate that there seems to be no systemic penalty for accessing the allocator via an abstract base class.

(**3**) AS3, AS5, AS11, and AS13 confirm that any use of a monotonic allocator where Utilization (**U**) is low, and problem size is not tiny, is typically suboptimal, if not a genuinely a bad idea. The reason for the abrupt change near (S) = 2^{12} is due to the internal boundaries within the multipool allocator's implementation, which provides for large allocations (larger than 2^{12}) to pass through to the backing allocator.

(**4**) AS7 and AS9 are clearly the winning allocation strategies until the block-size (S) exceeds the maximum size that can be accommodated internally by an adaptive pool (2^{12}), at which point there is a modest overhead (at most a few percent) to forward the allocation through to the backing allocator.

Subsequent tables present experiments involving increasing total allocated memory (T) with similar results, reinforcing the preliminary conclusions presented above. As memory demands increase, it is possible that the performance degradation for strategies AS3 and AS5 (and, as we will see, even AS11 and AS13) may deteriorate to outright failure.

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
31	15	10	0.127s	104	428	434	39	38	39	41
31	16	10	0.123s	102	442	446	42	42	41	40
31	17	10	0.124s	102	439	442	45	45	42	45
31	18	10	0.123s	102	442	447	47	46	41	42
31	19	10	0.123s	107	441	446	42	41	46	43
31	20	10	0.127s	99	431	434	44	42	41	41
31	20	11	0.064s	102	815	824	48	40	52	48
31	20	12	0.038s	93	1369	1387	57	51	47	54
31	20	13	0.021s	102	2368	2392	108	80	2376	2401
31	20	14	0.013s	61	3787	3833	109	67	3797	3844
31	20	15	0.007s	54	6621	6706	112	59	6651	6708

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
32	15	10	0.248s	103	fail	fail	38	39	38	41
32	16	10	0.248s	102	fail	fail	38	41	38	39
32	17	10	0.246s	102	fail	fail	40	39	39	39
32	18	10	0.246s	102	fail	fail	40	40	39	40
32	19	10	0.246s	102	fail	fail	40	42	40	40
32	20	10	0.246s	102	fail	fail	40	41	41	41
32	20	11	0.124s	102	fail	fail	46	44	41	47
32	20	12	0.062s	102	fail	fail	44	45	46	56
32	20	13	0.034s	108	fail	fail	127	110	fail	fail
32	20	14	0.022s	72	fail	fail	105	78	fail	fail
32	20	15	0.015s	87	fail	fail	99	60	fail	fail

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
33	15	10	0.495s	102	fail	fail	41	39	39	39
33	16	10	0.493s	102	fail	fail	38	39	38	41
33	17	10	0.492s	102	fail	fail	38	41	38	40
33	18	10	0.492s	102	fail	fail	40	41	39	40
33	19	10	0.492s	102	fail	fail	40	41	40	41
33	20	10	0.492s	102	fail	fail	40	40	40	41
33	20	11	0.248s	102	fail	fail	42	43	41	42
33	20	12	0.122s	101	fail	fail	43	47	45	47
33	20	13	0.062s	102	fail	fail	112	112	fail	fail
33	20	14	0.040s	89	fail	fail	96	88	fail	fail
33	20	15	0.022s	102	fail	fail	107	80	fail	fail

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
34	15	10	0.990s	103	fail	fail	41	39	41	39
34	16	10	0.986s	102	fail	fail	38	39	38	40
34	17	10	0.985s	102	fail	fail	38	39	39	40
34	18	10	0.984s	102	fail	fail	40	40	39	40
34	19	10	0.983s	102	fail	fail	40	41	40	40
34	20	10	0.984s	102	fail	fail	40	41	40	41
34	20	11	0.494s	102	fail	fail	42	42	41	42
34	20	12	0.241s	102	fail	fail	43	42	47	44
34	20	13	0.120s	107	fail	fail	114	113	fail	fail
34	20	14	0.064s	102	fail	fail	117	112	fail	fail
34	20	15	0.038s	96	fail	fail	103	95	fail	fail

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
T	A	S	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
35	15	10	1.981s	102	fail	fail	38	41	39	39
35	16	10	1.975s	102	fail	fail	39	40	38	39
35	17	10	1.970s	102	fail	fail	39	40	39	40
35	18	10	1.967s	102	fail	fail	39	39	39	40
35	19	10	1.967s	102	fail	fail	39	41	40	41
35	20	10	1.968s	102	fail	fail	40	41	40	40
35	20	11	0.988s	102	fail	fail	41	42	41	41
35	20	12	0.481s	102	fail	fail	42	42	44	44
35	20	13	0.240s	102	fail	fail	113	113	fail	fail
35	20	14	0.125s	102	fail	fail	113	112	fail	fail
35	20	15	0.070s	94	fail	fail	103	110	fail	fail

A striking result in this benchmark is that some of the tests failed to run to completion, because 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 values for (S) in all of these Benchmark III tables expose the effect of an implementation detail of the used multipool allocator: Allocations larger than an implementation-defined size – specifically 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 even for AS11 and AS13.

This utilization-focused experiment was purposefully simplistic. We reasoned that variations in allocated size were unlikely to affect either a monotonic or multipool allocator (this variation may have been of greater interest had a coalescing allocator been under consideration). Furthermore, altering the deallocation strategy from least recently allocated to some other (e.g., pseudo-random) one may have provided additional insight, but at the cost of conflating the effects of Locality (**L**) with those of Utilization (**U**).

10 Benchmark IV: Variation in Contention

This fourth and final benchmark was designed to demonstrate the effects of Contention (**C**) – i.e., 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**) – on runtime performance. In this experiment, a set of threads was created and used to repeatedly allocate and deallocate a chunk of memory. To emphasize the runtime 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 contended for the same allocator. For the other (local) allocation strategies considered (AS3, AS5, AS7, AS9, AS11, and AS13), each thread had access to its own private *unsynchronized* 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 (**D**) of this experiment is extremely high.

The chunk-size parameter, `S`, for this experiment is measured in bytes. The other parameters for this experiment are the number of iterations (`N`), and, from section 5, the number of active threads (**W**). The results of this experiment are normalized to the respective results for of AS1 in each row. Specifically, the results under AS1 are times in seconds, and the values under the other allocation strategies – AS2, AS3, AS5, AS7, AS9, AS11, and AS13 – are represented as percentages of the AS1 value, where, again, lower percentage values imply shorter run times.

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

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

Each of the runs represented in this first table (above) consist of 2^{15} repetitions – per thread – of allocating and then immediately deallocating a chunk of memory of size 2^6 bytes. The first row depicts a run in which the main program spawns just a single thread (**W** = 1). The runtime using the default allocator (AS1) is shown under the AS1 column as 0.022 seconds. Using the same allocator via an abstract base class (AS2), we observed a runtime that was 174% of this reference time, considerably more than for direct access When we used a local monotonic allocator directly (AS3), the relative cost was just 71% of that of using AS1. Accessing that same allocator via an abstract base (AS5), also yielded 71%. Switching to a multipool allocator – used directly and via an abstract base class – resulted in relative runtimes of 32% and 36%, respectively. Finally when the combination of a multipool allocator backed by a

monotonic one was employed directly, the runtime was measured at 35% and, when accessed via a base class, 34%.

In each successive row, we increase the number of spawned threads by 1, each executing the function performing 2^{15} iterations of allocating and then immediately deallocating a block of 2^6 bytes. Note that the hardware used had more available processors than the maximum number of threads (**W** = 8) considered.

A quick look at the tables below show that the global allocator (AS1-AS2) along with the monotonic one (AS3 and AS5) are poor candidates for this usage scenario. Incorporating additional threads did not generally increase the runtime cost of either of the global allocators, nor of any of the local ones. An early, fairly consistent pattern emerges, suggesting that there is fixed proportional speedup depending on the local allocator provided, with AS7 being the most consistent winner, yet any strategy that makes use of a multipool (AS7, AS9, AS11, and A13) is clearly preferable to the default (AS1) by a sizable factor (~4x).

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
15	7	1	0.023s	100	114	116	44	47	47	48
15	7	2	0.043s	101	46	69	26	26	26	26
15	7	3	0.041s	103	51	68	25	25	22	25
15	7	4	0.033s	121	78	95	26	19	20	23
15	7	5	0.031s	102	81	86	20	26	26	25
15	7	6	0.032s	99	84	84	18	23	19	25
15	7	7	0.029s	114	111	110	23	27	21	31
15	7	8	0.029s	117	114	120	27	35	31	29

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
15	8	1	0.043s	101	87	89	23	23	22	23
15	8	2	0.042s	102	61	59	23	23	27	26
15	8	3	0.046s	90	85	111	23	25	24	25
15	8	4	0.040s	84	100	98	18	18	19	22
15	8	5	0.028s	136	190	200	30	30	30	38
15	8	6	0.024s	125	209	201	33	33	31	29
15	8	7	0.033s	108	162	162	24	29	26	26
15	8	8	0.031s	114	184	188	34	33	36	42

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
16	8	1	0.085s	97	109	107	23	23	23	23
16	8	2	0.091s	101	104	106	22	21	22	21
16	8	3	0.093s	100	105	104	22	21	21	21
16	8	4	0.097s	94	93	121	20	20	18	17
16	8	5	0.078s	118	108	130	24	18	17	18
16	8	6	0.059s	87	138	136	21	26	22	26
16	8	7	0.063s	93	137	135	17	27	21	20
16	8	8	0.057s	109	162	164	29	28	28	26

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
17	8	1	0.090s	100	206	206	45	42	42	42
17	8	2	0.179s	101	107	106	22	22	22	22
17	8	3	0.179s	101	104	104	22	23	22	22
17	8	4	0.209s	109	89	70	16	15	11	11
17	8	5	0.177s	100	85	78	12	15	15	15
17	8	6	0.108s	142	147	178	27	28	25	25
17	8	7	0.140s	85	116	132	24	22	22	22
17	8	8	0.118s	100	142	150	22	21	25	26

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
18	8	1	0.177s	109	177	177	45	45	45	46
18	8	2	0.339s	100	95	95	24	24	24	24
18	8	3	0.333s	102	99	95	24	25	24	25
18	8	4	0.304s	98	93	93	24	21	26	21
18	8	5	0.311s	94	97	86	22	24	25	20
18	8	6	0.276s	95	118	122	16	17	18	17
18	8	7	0.297s	79	109	108	18	18	21	18
18	8	8	0.219s	114	176	186	26	21	21	23

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

				global	←monotonic→		←multipool→		←mono+multi→	
				virtual		virtual		virtual		virtual
N	S	W	AS1	AS2	AS3	AS5	AS7	AS9	AS11	AS13
19	8	1	0.421s	89	134	134	28	23	21	25
19	8	2	0.615s	101	93	93	25	26	26	26
19	8	3	0.631s	99	93	93	25	25	25	25
19	8	4	0.565s	107	95	103	28	28	29	28
19	8	5	0.575s	119	106	101	27	28	27	27
19	8	6	0.499s	114	126	113	17	25	28	22
19	8	7	0.558s	100	113	115	18	18	15	16
19	8	8	0.460s	105	149	148	19	21	18	21

Since modern default global allocators were designed with threading as a concern, the results are not jaw-dropping. This benchmark demonstrates, 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 increased (perhaps due to a dearth of physical locality within the monotonic allocator's buffer itself).

11 Conclusion

Object-level control over memory allocation is intrinsic to C++, and must always be so if we hope to maintain this language's supremacy as the best-performing high-level "systems" language. Supporting object-specific memory allocation is admittedly an added burden – exacerbated by an initially difficult-to-use 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*.

[4] Memory Allocator Benchmark Data, https://github.com/bloomberg/bde-allocator-benchmarks

[5] Graham Bleaney, *P0213R0: Reexamining the Performance of Memory-Allocation Strategies*.