Value Proposition: Allocator-Aware (AA) Software

Document #:	P2035R0
Date:	2021-01-12
Project:	Programming Language C++

Reply-to:	Pablo Halpern <phalpern@halpernwightsoftware.com>
	John Lakos <jlakos@bloomberg.net>

NOTE: This white paper (i.e., this is not a proposal) is intended to motivate continued investment in developing and maturing better memory allocators in the C++ Standard as well as to counter misinformation about allocators, their costs and benefits, and whether they should have a continuing role in the C++ library and language.

Abstract

The *performance benefits* of employing local memory allocators are well known and substantial. Still, the real-world costs associated with integrating allocators throughout a code base, including related training, tools, interface and contract complexity, and increased potential for inadvertent misuse, cannot be ignored. A fully *allocator-aware* (AA) software infrastructure (SI) offers a convincing value proposition despite substantial upfront costs. The *collateral benefits* for clients, such as object-based instrumentation and effective means of testing allocations, make investing in AASI even more compelling. Yet many other unwarranted concerns — based on hearsay or specious conjecture — remain.

In this paper, we discuss all three currently available AA software models, C++11, BDE, and PMR (C++17)[1]— each of which provides basically the same essential benefits but requires widely varying development and maintenance effort. We then separate real from imagined costs, presenting some of the many collateral benefits of AASI along the way. After all aspects are considered, we continue to advocate for the adoption of AA software today for all libraries that potentially have performancesensitive clients and specifically for the BDE/PMR model, even as we continue to research a language-based solution that might someday all but eliminate the costs while amplifying the benefit.

[1]The BDE (Bloomberg Development Environment) and PMR (polymorphic memory resource) models are very similar (the latter being derived from the former), and we often refer to them together as the BDE/PMR model.

Contents

1 Introduction

Allocating memory dynamically is an inherent aspect of practically every significant software system. Although the language-supplied allocation primitives (new and delete) typically provide acceptable performance, employing a custom allocation strategy would be advantageous, if not absolutely necessary, in many important cases. This paper explains why custom memory allocation supported via an allocator-aware (AA) software infrastructure (SI) can be practical, cost effective, and strategically advantageous today, just as it has at Bloomberg and elsewhere since 1997. [2]

Using thoughtfully-chosen local ("arena") memory allocators to provide custom allocation strategies is well known — both *anecdotally* and through repeated *controlled experiments*[3] — to potentially yield significant (sometimes order-of-magnitude) performance improvements over simply relying on even the most efficient state-of-the-art, general-purpose global allocators. This performance boost should come as no surprise since, when choosing an allocation strategy, developers can leverage in-depth knowledge of their application and its operational environment — an option that is clearly unavailable to any general-purpose allocator.

The two most common ways of achieving high-performing memory management in C++ today are to (1) write custom data structures (from scratch) each time a distinct allocation strategy is deemed necessary, and (2) build on AA components (provided by library developers) that readily support use of arbitrary per-object allocators as needed. The costs and benefits for option (1) are extreme: Custom data structures produce the highest possible levels of performance and flexibility yet have prohibitively high development and maintenance costs; are inherently not reusable (nor interoperable with their less-efficient counterparts); and introduce a steep learning curve (and typically a high bug rate).[4] Option (2) offers nearly the same performance advantages of option (1) and provides many other, collateral benefits yet requires from clients of AASI libraries only a tiny fraction of the engineering effort.

The introduction of an AASI can, by analogy, be compared to that of a middle-tier class of service in the airline industry, such as premium economy or business class. Many customers who would otherwise have flown economy might now opt for premium economy, which offers extra legroom or laptop room as well as other amenities at minimal increased cost. Other customers whose need for comfort would have necessitated flying first class can now get nearly the same accommodations by opting for business class at a fraction of the cost. Analogously, many software subsystems that would have performed acceptably without custom memory allocation might now, with little marginal client cost, be optimized (often significantly) by exploiting an AASI, and subsystems that would otherwise have been forced to create custom data structures can now get substantially the same exceptional performance, instrumentation, and placement capabilities at a fraction of the development effort for the client. Despite its simplicity, this analogy offers considerable depth of

[2] The polymorphic memory allocator model used at Bloomberg and now part of C++ was developed for real-world financial software applications in the F.A.S.T. (Financial Analytics & Software Transactions) Group at Bear Stearns & Co., Inc., in 1997, and has been in use at Bloomberg since 2001.

[3] [**?**], [**?**], [**?**]

[4] Such bespoke data structures could use generic components customized by policies to reduce the effort. The decision on which policies to use, however, is either made once per class, limiting the ability of clients to customize allocations according to their needs, or is exposed as a template parameter, essentially replicating the C++11 allocator model and thus sharing its same deficiencies. Moreover, such policy-based generic components are themselves notoriously difficult to write, maintain, and test thoroughly.

insight into the sound microeconomic arguments for adopting a robust AASI at large-scale software-development companies such as Bloomberg (see Appendix I)

2 Alternative Models for AA Software

The engineering costs associated with developing, maintaining, and using AA components vary widely with the model. We have extensive experience implementing and using three distinct interface models for AA software currently employed in C++ today. These models differ primarily in the syntactic details by which custom memory-allocation logic is injected when constructing a new object.

C++11 model (high cost). The custom allocator's implementation is embedded (at compile time) directly in the object's type, thereby delivering the most general and high-performing of these allocator models. This compile-time-centric model guarantees zero runtime and space overheads when using the default (global) allocator type[5] and enables the placement of objects in memory having nonstandard address types.[6] Widespread use of this model would, however, severely impede client productivity. Because the allocator affects the object's type, much of the code (including the application layer) would need to be templated, resulting in reduced maintainability due to acute compile-time coupling. Moreover, interoperability among subsystems using distinct allocator types would be suffer profoundly.[7] In fact, when even a small fraction of subsystems requires a distinct allocator, the development and usability costs required to support this model are so high that many organizations (including the C++ Standards Committee[8]) will not adopt it universally.

BDE model (moderate cost). A pointer to an allocator base class is embedded in every object that might directly allocate memory, even when using the default allocator. Although somewhat less general than the C++11 model,[9] the engineering effort required to manually "plumb" these polymorphic allocators throughout an object's constructors is substantially reduced. Moreover, because polymorphic allocators do not invade their (compile-time) types, objects using distinct allocator types can interoperate naturally in nontemplated contexts.[10] Furthermore, this classically object-oriented model enables clients to request an object's allocator (via its base-class address) without the clients themselves being templates.

PMR model (moderate cost). This model, sometimes also referred to as the C++17 model,[11] is derived from — and behaves essentially the same as — the BDE model; the only (syntactic) difference is that the pointer to the polymorphic base class is wrapped in an object that meets

[5] Other models *could* be made to approach zero space overhead by encoding the default allocator as a single bit; doing so, however, is counter-indicated by benchmarks of typical use cases.

[6] E.g., only the C++11 model supports shared memory.

[7] E.g., a function template expecting two arguments of (exactly) the same type would fail to instantiate if instead passed, for example, two vectors employing different allocator types.

[8] C++11-model allocators are not currently used in every class within the Standard Library where allocators would be appropriate (e.g., `std::path`).

[9] BDE allocators do not support placement in shared memory, which is a rare and highly specific use case.

[10] In particular, objects employing BDE-model allocators can serve as output parameters where proxy objects, such as `std::string_view`, would be unsuitable.

[11] Although C++17 supports the entire C++11 model, we sometimes use the term C++17 model as shorthand for the polymorphic-memory-allocator (PMR) model within C++17

the requirements of a C++11 allocator (and hence can also serve as an adapter to C++11 code that uses allocators).[12] Moving forward, this PMR model is expected to increasingly supplant the older BDE model at Bloomberg.[13]

By maintaining an appropriate AA subset of our code base using any one of the aforementioned models, we provide essentially all the benefits typically sought from custom solutions at greatly reduced client effort and (depending on the model) greatly reduced overall engineering cost.

3 Performance Benefits

Reductions in overall run times resulting from local ("arena") memory allocation can manifest both during memory allocation/deallocation itself and also during access to allocated memory (irrespective of the manner or cost of its allocation); the modality of use (e.g., short-running versus long-running) will govern which aspect of reduced run time dominates. Unlike general-purpose global allocators, which must perform acceptably in all circumstances, special-purpose local allocators (such as monotonic allocators), where applicable, can afford unique advantages.[14].

Allocation/Deallocation. Memory-allocation profiles vary widely across applications. One common pattern for short-running programs is to build up a data structure, access it briefly (often without modification), and then destroy it. In such cases, maintaining an ability to delete individual parts (as required of a general-purpose allocator) is unnecessarily costly in both time and space. In contrast, using a local monotonic allocator instead,[15] for which deallocation is a no-op (no operation) and memory is reclaimed only when the allocator is destroyed, affords significant performance benefits.[16] Depending on the data structure, improvements of up to 5 times have been realized.[17]

Another common pattern is to repeatedly allocate and deallocate memory blocks having a few distinct sizes (e.g., for distinct object footprints) throughout the lifetime of the program. Such memory-allocation profiles benefit from the use of a local multipool allocator, which internally manages dynamically growing pools of fixed-size memory chunks, caching any deallocated chunks for efficient reuse, with or without thread synchronization (see item 3).

Each of these kinds of local allocators is a form of managed allocator, which supports reclaiming all memory allocated by it in a single (client-invokable) operation. Thus, memory blocks

[12] [**?**], [**?**]

[13] A joint management-sponsored initiative between the BDE (Bloomberg Development Environment) and the recently announced Bloomberg Software Engineering teams is the creation of what is being tentatively dubbed bde4.0. This inclusive effort — explicitly involving London and New York — is intended to produce a "stepping stone" that will allow us to bring the older in-house versions of (mostly) standard-compliant libraries in sync with modern C++ facilities, eliminating incompatibilities (e.g., resulting from nonstandard namespaces) and thereby facilitating easy integration with opensource, third-party, and other standard-compliant libraries.

[14] For expert advice on how to design an effective local allocator, see [**?**]

[15] Paul Williams observed (c. 2006) the first widely recognized (albeit anecdotal) evidence of the dramatic performance benefits afforded by monotonic allocators at Bloomberg in Bloomberg's front end [[**?**], Part I, approximately 7:28].

[16] BDE-model monotonic allocators were first employed at Bear Stearns (c. 1997) where they reduced the destruction time of complex financial-model objects from over 9 seconds to such a small time interval that it didn't even show up on the IBM/Rational Quantify (formerly Pure Quantify) profiler [[**?**], Part I, approximately 5:10].

[17] [**?**], [**?**]

allocated via managed allocators need not be deallocated individually, and objects that manage no resources other than memory need not even be destroyed![18] Using this (admittedly advanced) en masse technique, which we will hereafter refer to as winking-out), additional runtime performance gains of as much as 20% have been observed.[19]

Access Locality. To realize high runtime performance, modern computers organize memory hierarchically into multiple levels — L1, L2, and L3 caches, main memory, and secondary storage (disk or flash memory) — where each level of the hierarchy is typically one to four orders of magnitude slower than the one above it. The more densely packed the memory blocks within a working set,[20] the less likely the program is to overflow a specific cache and rely on a slower layer of the memory hierarchy. Additionally, (hardware or software) prefetching, which heuristically anticipates the next line to be brought into cache or the next page to be brought into main memory, provides benefits only when the data items being accessed are close together in the address space.[21]

Achieving *locality* (i.e., physical proximity of separately allocated memory blocks accessed repeatedly over a relatively short period of time) often p allocation/deallocation of the memory itself — especially for long-running programs. Absent the "corralling effect" of internal memory boundaries afforded by local arenas, the initially high-locality memory organization of tightly accessed subsystems may, over time, *diffuse*[22] across virtual memory, leading to performance degradation often exceeding an order of magnitude.[23] Using local memory arenas to ensure locality of reference allows us to realize (and to maintain throughout the lifetime of the process) the full runtimeperformance potential of the underlying hardware.[24]

Thread Locality. Multithreaded programming introduces additional inefficiencies to memory allocation and access that can be mitigated using local allocators. Whenever a structure or set of related structures is to be used (i.e., created, accessed, modified, or destroyed) by only a single thread at a time, costly synchronization (e.g., via mutexes or atomic instructions) required of general-purpose allocators can be avoided through judicious use of local allocators. Microbenchmarks of such scenarios consistently demonstrate performance improvements of roughly 4 times.[25] Moreover, by sequestering (into a separate arena) memory that is known a priori to be accessed by only a single thread at a time, local allocators naturally avoid ac-

[18] Per the C++ Language Standard, reusing undestroyed object memory is explicitly not undefined behavior as long as the abandoned object never again accesses such reused memory.

[19] [**?**], [**?**]

[20] The working set of a process is the collection of information referenced by the process in a specific period of time [**?**].

[21] Locality of reference [**?**] enables the processor to bring data into or to keep data in faster layers of the memory hierarchy when they are likely to be accessed in the immediate future [**?**]. An easy-to-read (albeit less definitive) description of locality of reference can be found in [**?**]

[22] Diffusion is the *spreading out* of related memory blocks and should not be confused with *fragmentation*, which is a phenomenon that occurs (most typically in coalescing allocators) when ample memory is available but not as sufficiently large contiguous blocks.

[23] [**?**], [**?**]

[24] Local arenas typically ensure that all blocks within a page have been allocated from the same arena. Additionally, because most local arenas are intended for use within a single thread, blocks can be packed tightly into contiguous cache lines without fear of creating false sharing. Indeed, multiple (complete or partial) blocks sharing a cache line can be beneficial in this case, as the blocks belonging to the arena are likely to be used together.

[25] [**?**], [**?**]

cidental cache-line contention (also called false sharing) caused by inadvertently interleaved allocations of small unrelated memory blocks that might subsequently be accessed by distinct concurrent threads.[26]

Maximizing performance, therefore, requires global knowledge of the application as well as a solid understanding of both when to use which kinds of allocators and how to do so correctly.[27]

4 Costs

Two *meaningfully distinct* types of costs naturally form when *creating* and effectively *exploiting* an AASI: (1) the *upfront* costs to library developers of creating (and maintaining) an AASI, and (2) the *incremental* costs to application clients of exploiting (or ignoring) AA aspects of the SI. While this section details the costs as they exist today, an effort is underway to integrate allocators into the C++ language such that both the upfront and incremental costs are drastically reduced — and in many cases eliminated — for both the library and application development teams, shifting essentially all of the burden onto the compiler itself.[28]

4.1 Upfront Costs

The upfront costs to create an AASI include making each memory-allocating infrastructure class AA, e.g., by applying a scripted series of modifications to a previously allocator-unaware class. Such modifications typically[29] amount to adding an optional trailing pointer-to-allocator parameter to every constructor and assiduously forwarding those parameters to base classes, data members, and any other managed subobjects as well as tagging the overall type as being AA via an allocator-trait metafunction.[30]

- The principle upfront cost is the added maintenance burden. Making software AA using BDE-model allocators, albeit tedious and potentially error prone, is straightforward, increasing line-count by 4 to 17 percent,[31] with 10 percent commonly recognized as typical source-code overhead. Training developers on how to write AA types (let alone how to test AA types properly) is incontrovertibly a significant upfront cost. Moreover, any time spent on allocator mechanics imposes a real opportunity cost on the organization as those same experienced software engineers become less available for other tasks.

[26] In other words, when objects "travel" among threads together with their local allocator (e.g., tasks dispatched to a thread pool), cross-thread synchronization, false sharing, and other needless pessimizations (such as padding that deliberately wastes space in local cache lines) are naturally and effortlessly eliminated.

[27] To learn much more about how to make effective use of local allocators in application as well as infrastructure development, see [**?**].

[28] An approach being considered for integrating allocators into the C++ language is described in [**?**]. The effort to develop a compiler supporting these new features — much like the effort to develop self-driving cars — requires significant upfront investment yet (when finally available) will so dramatically lower barriers to creating and maintaining an AASI (similar to autonomously operating a vehicle) as to be dispositive toward this paper's thesis.

[29] Class templates require metaprogramming in their constructors to correctly handle objects of dependent type that are not known to be AA until instantiation time. Container class templates must further add metaprogramming to their insertion methods for propagating allocators to potentially AA elements of dependent type. The implementation of these more difficult AA class templates is typically performed by standard-library suppliers or core development groups (e.g., BDE at Bloomberg) and is facilitated by library utilities such as those described in [**?**].

[30] To learn how to augment classes to make them AA in the BDE model, see [**?**].

[31] This data pertains to BDE (library) source code (c. May 2017).

- The concomitant additional risk of introducing AA-related software defects is mitigated by the use of static-analysis tools such as `bde_verify`,[32] which can alert developers to missing or misapplied transformations. A prototype of bde_verify that automatically transforms typical[33] components to BDE-model allocator awareness is available today. Enhancing `bde_verify` is a onetime technological-advancement cost that, although substantial, is expected to substantially reduce risk along with other upfront costs.

4.2 Incremental Costs

The incremental costs for a typical client of an AASI over an allocator-unaware SI are relatively small. An application client that chooses to take an active role in managing the memory of an AASI class need merely supply the address of the desired allocator to the class's constructor. Clients who elect not to partake simply ignore all mention of allocators and write their code as usual.

- While the substantial net savings in development effort for application clients who currently exploit AA software is clear, the overall savings for all application developers, including the many who don't exploit AA features (currently an overwhelming majority), is less so. Beyond just the initial upfront development costs (typically born by SI developers), making an infrastructure class AA enlarges its (programmatic) interface and (English) contract;[34] the additional parameters and methods required to support AA software impose new complexities, thereby increasing the cognitive burden on clients — even when they are indifferent to the benefits afforded by an AASI.

- Naïvely or otherwise carelessly supplying local allocators will invariably lead to client misuse via programmer errors, some of which can be surprisingly subtle. An obvious (albeit infrequent) user error is for the lifetime of an object to be allowed to exceed that of its allocator.[35] Much more commonly, however, a client will try to employ a special-purpose allocator that is ill-suited to their needs.[36] For example, declaring a monotonic allocator outside of a loop and then using that allocator to construct an object defined within the loop's body (and hence recreating that object on each iteration) might cause memory consumption to grow without bound. The likelihood of client misuse is exacerbated when a (substandard) implementation unexpectedly allocates temporary memory.[37] Even when the misuse is not catastrophic, mis-

[32] `bde_verify` is a Bloomberg tool that checks code for deviations from a number of best practices and style guidelines, in particular ensuring that AA components are put together correctly.

[33] I.e., components that are not especially difficult, e.g., those *not* implementing (general-purpose) *generic*, *templated*, or *container* types.

[34] [**?**]

[35] The effect of an object outliving its local allocator is similar to that of returning (from a function) a "dangling" pointer (to an automatic variable). When using the scoped allocator model [[**?**], pp. 2–6], however, a local allocator is typically created on the program stack just prior to creating the object (or objects) consuming memory from that allocator. Hence, when using this common idiom, the C++ language itself automatically guarantees that the lifetime of the allocator spans that of all such objects that depend on it.

[36] That monotonic allocators necessarily consume excessive memory when used with containers, such as `std::vector` and `std::string`, that grow geometrically is a popular misconception; the consumption of memory for such a container when coupled with a monotonic allocator is in fact always just a constant factor larger (i.e., similar in magnitude to that of a single extra reallocation)

[37] An especially insidious case of this sort of inadvertent "misuse" involved assignment between two `BitArray` objects within a loop. Assuming two `BitArray` objects hold the same number of bits, no reallocation is needed. As a development expedient, however, it is not uncommon to use the idiom of always first creating a temporary

application of AA technology can easily increase both development and maintenance costs without improving runtime performance.

- Making C++ types AA sometimes comes at the price of *incompatibility* with C++ *language features*, inhibiting the programmer's ability to use some of the more modern features of the C++ language.[38] For example, creating AA types requires additional consideration during construction and assignment and therefore cannot currently take advantage of compiler-generated constructors or assignment operators (a problem exacerbated in C++11 by rvalue overloads), nor can such types be constructed using aggregate initialization.

- To properly dispose of the memory used by an AA object, the lifetime of the object must not exceed the lifetime of its allocator. This *lifetime management* not only requires additional care from the developer, but also limits allocator applicability when using standard-library facilities that manage object lifetimes, such as `shared_ptr` and `weak_ptr`, since they neither track nor extend allocator lifetime. This cost, though non-zero, is typically less than the cost of avoiding dangling pointers or references in general because the normal structure of a program that uses local allocators is such that an allocator and the objects that use it are created and destroyed in a nested fashion within the same scope. Preventing dangling allocators when using `shared_ptr` is simply a matter of using the default allocator or, better, designing the program well enough that the maximum scope of the shared objects is well understood.

- Using allocators effectively in large industrial settings will of necessity incur significant administrative costs in *education*, *tools*, and *governance*. Code reviews, proper training, and allocator-use policies are essential costs that must be borne by any organization hoping to realize the full advantage afforded by an AASI. Investment in static-analysis tools (such as `bde_verify`) that detect erroneous or unwise allocator use (such as objects that outlive their allocators or a monotonic allocator servicing objects created repeatedly within a loop) are essential to reducing the burden on client programmers and to improving the robustness of the application codebase in their charge.

Despite all these very real and substantial upfront and incremental costs, a credible value proposition remains.

5 Collateral Benefits

If the aforementioned performance benefits alone were insufficient to justify the added costs associated with AA types, consider that several important collateral ones — not necessarily related to performance — go well beyond what even fully custom data structures can provide.[39]

object (using the original object's allocator) to build the result before swapping it with the object being assigned to (thereby automatically establishing the strongexception-safety guarantee). The combination of such low QoI (*Quality of Implementation*) and an unrealized (though not unrealistic) expectation on the part of the caller led eventually to memory exhaustion (thereby forcing process termination). See [**?**] and [**?**].

[38]The erroneous conjecture that allocators do not interact well with modern C++'s move *semantics* is, however, debunked in the "Common (But Unfounded) Concerns" section of this paper

[39]Changing the allocate/deallocate code path has been awkward due to its heretofore global nature. Some of the additional benefits of AA derive directly from the runtime polymorphism afforded by the BDE/PMR model, which enable an object's owner to inject arbitrary logic into this vital code path without having to restructure or even

Rapid prototyping and enhanced predictability. Maintaining a hierarchically reusable[40] AASI (along with a set of useful predefined allocators) encourages proactive experimentation. One could posit, for example, that a particular allocation strategy should yield an order-of-magnitude performance boost for a given subsystem. Having an AASI makes it easy for a developer to quickly inject into the subsystem a (typically off-the-shelf) allocator implementing the identified strategy.[41] If this quick-and-dirty experiment fails to produce the anticipated gains, the allocator can just as easily be removed or replaced by a different one. Such pragmatically valuable experimentation would be exorbitantly expensive if it required modifying a custom data structure with a hard-coded allocation strategy. Moreover, the business value of readily determining if (and to what extent) one or another custom allocation strategy would be beneficial should not be underestimated: Such prototyping removes much of the guesswork — and therefore the risk — associated with estimating the true effort needed to develop a product.

Modularity and composition (reuse). BDE/PMR allocators are typically chainable, i.e., one allocator provides some memory-management functionality and goes to another backing allocator when additional memory is needed. Chaining allows specialized allocators to be readily combined in myriad useful ways. For example, it is trivially easy to layer an allocation strategy that excels at allocating many small equal-sized blocks on top of one that is tuned for arbitrarily large ones. Moreover, chaining can incorporate various forms of instrumentation, e.g., for testing, metrics gathering, and monitoring.

Testing and instrumentation. A test allocator[42] can be an indispensable tool for ensuring the correctness of AA types. A `bslma::TestAllocator`,[43] for example, can be used to log memory-management-related activity, match deallocation with known allocations, check for memory leaks, or confirm exception safety, e.g., by throwing bsl::bad_alloc exceptions at strategic points in test scenarios. Using the allocator interface also makes possible the adding of instrumentation for debugging (e.g., detecting leaks or logging allocations), gathering metrics, and discovering the usage patterns needed to tune memory allocators in production systems.[44] Although offthe-shelf tools can be used to profile running programs, such external tools are often too heavyweight for use in production and are invariably incapable of providing object-specific information. The plug-in nature of polymorphic allocators, however, makes using an

recompile the code that uses those objects.

[40] [**?**], section 0.4

[41] [**?**]

[42] Since Lakos (employed at the time by the F.A.S.T. group at Bear Stearns & Co., Inc.) first conceived (c. 1997) of the BDE-model test allocators, they have caught innumerable memory- and exceptionrelated bugs early in the software development life cycle.

[43] A proposal for a PMR-model test allocator — based on our `bslma::TestAllocator` — has been submitted for consideration as part of the C++ Standard Library [**?**].

[44] The ubiquitous BDE-model AASI at Bloomberg was leveraged particularly effectively by what is known as the tagged allocator store (TAS), a framework Brock Peabody invented at Bloomberg (c. 2011) to track memory usage within individual instances of subsystems of a running application. In addition to `bslma::Allocator`, the custom allocator (which was also inherited from `gtkma::AllocatorStore`) allowed client objects that were aware of the possibility of such instrumentation to attempt a lateral dynamic_cast. If the cast succeed, client objects could use the alternate interface to report their subsystem memory usage on a per-object basis. These fine-grained per-object metrics (compared to typical static, scoped, or type-based ones) readily enabled real-time, at-scale monitoring of individual production subsystem instances that might potentially be overusing memory. The conspicuously successful TAS framework remains in active production use today [**?**]; [**?**].

instrumented allocator straightforward and practical in every conceivable context, from the smallest of unit tests to the largest of production programs.

Whole-object placement and garbage collection. Allocators enable client control over the placement of (entire) objects in special types of memory with relative ease. One can readily arrange for objects to reside in high-bandwidth memory, memory that is hardware protected (no read and/or write access), or persistent or filemapped (mmap) memory.[45] The same wink-out mechanism typically used to avoid calling individual destructors on a collection of AA objects can also be used for nonperformance-related purposes, such as a form of garbage collection: An entire set of objects having related lifetimes can be reclaimed at once by releasing them from a managed allocator, even when some of the objects are no longer referenceable. In contrast, consider a temporary graph of interconnected nodes that uses shared_ptr and weak_ptr to represent the edges of a graph such that deleting a key node causes the smart pointers to delete all of the connected nodes in a recursive destructor cascade. Using raw pointers to represent the graph's edges, allocating graph nodes from an arena allocator, and disposing of the entire graph at once simply by destroying the allocator would be simpler, less error prone, and far more efficient.[46]

Pluggable customization. The utility of allocators for obtaining both performance and nonperformance benefits is open-ended. These benefits depend largely (albeit indirectly) on the ability to plug new allocators into existing infrastructure at run time, though many of the benefits remain available (in whole or in part) when using even C++11-model allocators. Absent an AA infrastructure, realizing even a fraction of these benefits would require a prohibitively large expenditure of effort, especially when it involves modifying custom data structures. In particular, bespoke data structures lack the seamless interoperability necessary to be practicable at scale.

6 Common (But Unfounded) Concerns

Those new to BDE/PMR-model memory allocators might (understandably) be skeptical of adopting a ubiquitous AASI compared with other less invasive and more targeted approaches that claim similar performance gains.

6.1 State-of-the-Art Global Allocators

Advances in global memory allocators [47] *have led to dramatic performance improvements, especially with respect to real-world multithreaded applications.*[48], [49] *Wouldn't replacing the compiler-supplied*

[45] BDE-model allocators were a natural fit with Bloomberg's home-grown approach to saving/sharing (identical) process state on disk via saverange on fnch memory using the `GmallocAllocator`, which implements the `bslma::Allocator` protocol [**?**], Part I, approximately 6:40.

[46] A recursive destructor cascade resulting from smart pointers is not typically tail recursive and is hence susceptible to program-stack overflow; the wink-out approach, however, is always safe.

[47] E.g., since the seminal coalescing memory-allocation strategy proposed by Doug Lea in the late 1980s [**?**].

[48] Examples of smart multithreading global allocators are tcmalloc [**?**], ptmalloc, Hoard [**?**], and jemalloc [**?**]. Each has applications at which they excel and situations in which they are less adept, although our informal testing shows that jemalloc is slightly better than the others as a general-purpose choice. For optimal performance, these allocators should be tested and compared using your specific application.

[49] [**?**]

global memory allocator with a newer, state-of-the-art allocator achieve most (if not all) of the real benefits derived from assiduous use of local allocators designed into a program?

The short answer is no.[50] The design of every general-purpose allocator is still driven by the assumptions regarding specific runtime patterns; tcmalloc,[51] for example, is optimized for multi-threaded programs where the allocation pattern within each thread is not known in advance and allocations from one thread might be freed from another. Moreover, a global allocator that is linked as part of a library cannot significantly influence code generation, e.g., by reserving (automatic) storage on the program stack. In contrast, local allocators chosen for a specific usage pattern (e.g., many same-size allocations, single-threaded, or all-at-once teardown) that is known in advance to the application programmer can both avoid pessimistic assumptions and obviate runtime analysis. What's more, even when a global allocator would provide adequate performance, it would provide none of the collateral benefits afforded by an AASI.

6.2 Zero-Overhead-Principle Compliance

For all but the C++11 model, AA objects require maintaining extra state, even for the most common case (i.e., where the default allocator is used), and necessarily employ virtual-function dispatch when allocating and deallocating memory. Don't these overheads violate the zero-overhead principle?

Not at all. The parts of a program that do not make use of AASI components do not pay even the demonstrably small cost for extra allocator state or virtual function calls. Therefore, having an AASI available does not violate the zero-overhead principle (ZOP) any more than would, say, providing a class having virtual functions. This concern is misguided based on two separate aspects: object size and run time.

1. The overall object-size overhead that typically (but not necessarily) results when employing the underlying allocation model need not necessarily increase object footprint size. Some or all of it (and potentially even other object state that would normally reside in the footprint) may be relocated to the dynamically allocated memory itself.[52], [53] This relocation, however, is always subject to a space/time tradeoff, and such compressed-footprint implementations are rarely found in highly tuned code because performance measurements seldom favor them.[54]

2. Runtime overhead due to virtual dispatch is (perhaps counterintuitively) all but nonexistent. With simple, short-lived, allocate-use-deallocate patterns (where the overhead would matter most), the client's compiler typically has full (source-level) visibility into the implementations

[50] [?], Section 2, p. 4; [?]; [?]

[51] [?]

[52] On older Sun platforms supporting natural alignment [[?], Section 10.1.1, "Natural Alignment," pp. 662–665], for example, the footprint of an `std::vector` consisted of just a single word (e.g., four bytes) unless the sizeof the contained element was 1, in which case the template was specialized to yield two words instead of one. This design tradeoff favors compactness for the empty vector and therefore immense performance gains for sparse matrices (i.e., vectors of vectors) but at the expense of (more typical) nonsparse ones.

[53] Consider that general-purpose allocators commonly prepend small amounts of bookkeeping storage just below the (starting) address of the returned memory; storing the address of the allocator itself there is no different.

[54] For example, a too-clever-by-half design for short-string optimization (SSO) was conceived of separately by Lakos and Alexandrescu c. 2000 (see, e.g., [?]). Using this design, the last byte in the SSO buffer was intended to hold the bookkeeping state in such a way that it would become 0 (and hence dually serve as the terminating null character) for a maximally long string fitting into the footprint. This design is not used in BDE today because it was measured to be significantly slower for its intended clients than a more horizontal (less space-efficient) encoding.

of both the container and the allocation functionality being injected and is therefore able to devirtualize the calls, eliminating entirely any runtime cost of virtual dispatch.[55], [56]

Conversely, locality (or lack thereof) typically dominates runtime performance for longer-running processes, irrespective of any allocation/deallocation suboptimality (e.g., due to virtual dispatch).[57]

As a source of design guidance, the ZOP is less useful for library features than for language features. As SI developers, we often make practical performance-related choices and tradeoffs during library design.

1. An example of a tradeoff that offers benefits to some with (essentially) no cost for any (i.e., a "Pareto-optimal" performance improvement, which would not actually be a tradeoff) is the replacement of an O(N) algorithm with an O(log N) one for cases invariably involving only sufficiently large values of N.

2. An example of solid benefits for some but at a small cost to many or all is the Standard's requirement that std::list<T>::size() be O(1), which effectively mandates a larger footprint size for all such list objects.

   ]item And lastly, an example of a stark performance tradeoff, having benefits for the expected case, with significant cost for less-typical cases, is short-string (aka small-string) optimization (SSO). The increased footprint of a string that supports SSO is wasteful for strings that are either empty or too large to fit in the short-string buffer. Nevertheless, the benefit for the typical or expected case is so large that the specification for std::string was designed to permit implementation using SSO, and all major library vendors do so.

In addition to pure performance tradeoffs, we sometimes make design choices that trade off functionality for ease of use. An example of a design tradeoff heavily favoring functionality over ease of use is that of the C++11 allocator model. In addition to always guaranteeing absolutely zero runtime and space overhead when using global allocators, the C++11 allocator model is maximally general (e.g., enough so to support allocating even shared memory) but is also considered all but unusable for most programmers. On the other hand, a BDE/PMR-based AASI, which restricts usage to just conventional memory addresses, is an example of making this tradeoff in the opposite direction: A few hearty souls will fend for themselves, so that all can thrive.

In addition to deliberately excluding some forms of alternative (e.g., shared) memory, BDE/PMR-based AASI also makes another tradeoff: The cost of maintaining an extra allocator pointer (the so-called "allocator tax") is similar to mandating that everyone purchase automobile insurance, a modest cost required of all drivers so that funds are available in case of accident. Keep in mind that this runtime overhead where no special allocator is needed is typically negligible — e.g., compared with that of, say, SSO.

Developing highly performant (hierarchically) reusable libraries requires performance tradeoffs that

[55] [**?**], Section 7, pp. 12–28; [**?**]

[56] As a demonstration of devirtualization [halpern19], we defined a simple string-like class that uses a BDE-model allocator, and we observed that the compiler inlines the allocate and deallocate calls rather than invoking them through the vtbl. Some current compilers, however, surprisingly fail to devirtualize nearly identical code that uses a pmr::polymorphic_allocator, which simply wraps a pointer to pmr::memory_resource.

[57] [**?**], Section 8, pp. 28–47; bleaney16

necessarily impose some degree of undesirable costs on some class of potential users. Would a library that is not somehow AA be considered to violate the ZOP because it imposes the cost of global (suboptimal, untunable) memory allocation on all users of that library? By that metric, every reusable library would violate ZOP, which is clearly an absurd assertion! We maintain that the tradeoff favoring a BDE/PMR-model AASI over either a C++11-model AASI or no AASI at all is sound.

6.3 Verification/Testing Complexity

Failure to properly annotate types or propagate allocators can undermine the effectiveness of the allocation strategy and can lead to memory leaks, especially when winking-out memory. Aren't extensive verification, testing, and/or peer review required to avoid such errors?

Use of virtually any new C++ library or language feature adds some amount of testing burden for the client. Use of allocators is entirely opt-in; client developers unconcerned with employing alternative allocators can simply ignore any optional allocator arguments and use the currently installed default allocator automatically, thus requiring no change to verification or testing methodology. Winking-out memory — with or without invoking destructors — is admittedly a powerful and potentially dangerous technique; like many other expert-level C++ features, engineer training and discipline will be essential to avoid inadvertently misapplying it.

Compared to other, more irregular techniques, correct use of allocators with a consistent AASI can, precisely because of their regularity, be more readily checked at compile time via static-analysis tools, such as bde_verify, or at run time (in test drivers) using, for example, bslma::TestAllocator. On the client side, quotidian use of bde_verify can prevent common errors, such as returning an object constructed using a local allocator on the program stack or repeated use of the same monotonic allocator from within a loop. Thus, BDE-model allocators can, in practice, be substantially less error prone than many other, less regular forms of custom memory allocation.

6.4 Compatibility with Modern C++ Style

C++11 encourages a style of programming where objects are more often passed and returned by value, sometimes relying on rvalue references to move these objects efficiently. In contrast, BDE style relies on passing AA objects (by address) as arguments to achieve optimal efficiency and control over the allocator employed.

Whether or not an object returned from a function is AA has absolutely no effect on the effectiveness or advisability of passing and returning objects by value. Employing solely the return-by-value style significantly impedes performance in cases where a function returning an allocating object is used repeatedly, regardless of whether allocator customization is desired.[58] For example, if a function returning a vector by value is called in a loop, then that vector object must necessarily be created and destroyed on each iteration, thus obviating any possibility of object pooling (an oft touted "alternative" to AA software, see "Compared to non-AA Alternatives," below).

By contrast, if the address of the resulting object is passed into the function as a modifiable argument, internal memory will typically be allocated during the first call to the function, occasionally

[58] For exactly this reason, the BDE style guidelines discourage returning objects (by value) that allocate memory (irrespective of whether or not it is AA).

grow (when necessary) during subsequent calls up to some high-water mark, and be repeatedly reused thereafter. Entirely separately, to support non-default allocators, the function should accept the would-be returned AA object as a pointer parameter (return-by-argument) so that the caller can construct (using the desired local allocator) the object that is to hold the result. A return-by-value–style interface (if desired for clients who are content with using default allocators) can then be built on top of an underlying pass-byargument interface (but not vice versa). [59]

6.5 Move vs. Allocate

When two objects use different allocators, move assignment degenerates to a copy operation and swap has undefined behavior; doesn't that imply that local allocators should be avoided to enable such operations?

On the contrary, because the time to access memory often overshadows the time needed to allocate and deallocate it,[60] copies commonly provide better overall runtime performance than movse[61] — particularly in large, long-running systems where smaller, densely packed subclusters of data are accessed (repeatedly) in bursts. Modern computer architectures exploit such locality to improve runtime performance[62] ; indiscriminate use of move operations tends to degrade performance by reducing locality of reference.[63]

When the current working set is sufficiently small (e.g., it can fit into main memory or perhaps even cache), the overhead due to diffusion is far less pronounced and might well favor moves over copies as is typically the case within a single container. If the entire program is sufficiently small, a single, global allocator — possibly even the default one[64]— will usually suffice. Conversely, when the working set is too large to fit within a specific level of the memory hierarchy (e.g., L1 cache, L3 cache, or main memory), the loss in performance (due to loss of temporal locality and thrashing) will invariably overwhelm any runtime overhead of copying data into a local arena.[65] Moreover, if the objects being accessed are smaller than an atomically fetched block of memory, such as a cache line or (more commonly) a page, then a significant speed-up (due to spatial locality and constructive interference) occurs when objects that are used together share a common block (irrespective of any specific access pattern or prefetching algorithms).[66]

---

[59] Having both styles colocated within the same scope, however, would needlessly clutter the interface for both sets of clients and would impose on clients of the lower-level interface an unnecessary physical dependency on the higher-level interface.

[60] [**?**], [**?**]

[61] [**?**]

[62] See Item 2, "Access Locality," in the "Performance Benefits" section of this paper.

[63] E.g., when the (overall) system size exceeded L3 cache size, one benchmark using move performed only one third as fast as the same program using copy [**?**]. When system size approached twice the size of physical memory, the move version was only one tenth as fast.

[64] Note that a performance improvement of roughly 4 times was observed [[**?**], Section 10, pp. 53–57] using an unsynchronized local multipool allocator compared to the default global one.

[65] A recent test of moves and copies [[**?**]] conducted at Bloomberg showed a 1.5 to 2.5 times speedup of copy over move for data sets from 4MB to 32MB with either a large number of small subsystems or a small number of large subsystems when each element was accessed 512 to 8096 times. The root cause of this effect is not yet fully understood, but hardware prefetching is suspected to have played a role in speeding up access to elements stored close together in physical memory.

[66] Given a sufficiently large L3 cache, the entire working set (in terms of cache lines) would theoretically fit in L3 and yet (due to pathological diffusion throughout virtual memory pages) fail to fit within physical memory, thereby resulting in page thrashing.

The BDE/PMR allocator model directly facilitates optimizing moves (and swaps) by copying data when it is most likely to be beneficial to performance and eliminating copies otherwise. In particular, because a container using the BDE/PMR model propagates its allocator to each of its contained elements, moves within such a container (e.g., during insert, delete, or sorting operations) never degenerate to copies and swaps remain O(1).[67] If different subsystems (i.e., architecturally significant subregions of a program having independent access patterns) use different allocators (as they typically should), logical move operations across subsystems result in data being physically copied, thus preserving locality within each subsystem, whereas moves within an arena (where locality already exists) do degrade to copies.

Because *access time* rather than *move/copy* time typically dominates runtime performance for large systems, programmers must manufacture a domain of locality for each subsystem. This need for locality, achieved via local (arena) memory allocation, naturally takes precedence over preferring moves to copies.

6.6 Compared to Non-AA Alternatives

Object pools and factories serve to reduce overhead caused by allocating memory, so why aren't these other approaches good (if not better) alternatives to allocators?

First, while these other more specialized (perhaps more familiar) higher-level techniques serve a valuable purpose, they are not a general substitute for finegrained, articulate memory allocation. Where applicable, object pools avoid creating and destroying objects across disparate uses, thereby also reducing the frequency with which individual piece parts are allocated. Object pools and associated factories are often ideal for application-level purposes. For innumerable other purposes, however — e.g., implementing contiguous-storage containers, such as `std::vector` or `std::deque` — they are simply not applicable. Moreover, their utility in programs having high *utilization* (U) of the type being pooled would be dubious.[68]

Second, even when these more specialized techniques are applicable, they are never significantly faster and often are less performant than the standard pieces of a well-designed and well-implemented AASI. Unlike AA containers, *ad hoc* containers that are specialized to use these techniques are not ubiquitously interoperable and do not scale as well to large code bases. Other than where having such higher-level specialized constructs offers clear usability gains,[69] the development (e.g., training) overhead of maintaining multiple ways of accomplishing the same task is reason enough to generally avoid them.

Third, when object pools and factories are a good fit, they are most appropriately built on top of hierarchically reusable AASI components, such as an adaptive memory pool (which, of course, is naturally AA already). Given that the hard work has already been done, the object pool would

[67] When swapping individual objects that might not share a common allocator, however, using alternative (e.g., > O(1) and/or potentially throwing) swap routines that do not require allocator equality is appropriate. As with move, such an alternative swap would make full copies of the objects being swapped if and only if their allocators do not compare equal, i.e., exactly those circumstances where copying elements is likely to be preferred over simply swapping pointers.

[68] Memory utilization and its effects on performance are described in detail at 55:15 in Part I of [**?**]. See also [**?**] and [**?**].

[69] E.g., object pools used in long-running programs in which complex objects are routinely recycled to avoid the (gratuitous) runtime cost of repeatedly destroying and then recreating them.

naturally be AA as well. A factory or object pool that does not use a (plug-in) allocator is, in effect, doing the job of both the pool and the allocator, thus gratuitously reducing the modularity of the software.

Finally, even when these alternative approaches are the right answer, unless they themselves are AA, they forfeit all the collateral benefits of an AASI and are therefore hardly better — and often significantly slower — than their fully custom counterparts.

7 Conclusion

Custom memory-allocation strategies imbued in C++ software can have tremendous beneficial impact, principally in the form of enhanced runtime performance but also in flexible object placement, instrumentation, and so on. Historically, however, application clients have realized these various benefits only through use of custom data structures at truly exorbitant development and maintenance costs.

Whether to make use of custom memory allocation is an economic decision that affects both the developer making the decision and the business as a whole. When an application developer determines that the benefits of employing custom memory allocation outweigh the costs, then such custom allocation is indicated. Thus, more software will reap the benefits of custom memory allocation if the costs — both real and perceived — are kept low.

In this paper, we proposed having an (appropriately) AASI as an alternative way for application developers to realize nearly the same advantages of fully customized memory management but at a tiny fraction of the client cost. Furthermore, having such AASI in place affords less-needful clients these very same advantages with minimal added effort.

Supporting an enterprise-wide AASI does introduce associated fixed engineering costs. These extra costs are borne largely by SI developers, but they also result from added operational overhead (e.g., developer-facing tools, training, and documentation) and/or added risk due to accidental (client) misuse. Other concerns might call into question the desirability of having AASI at all. When looking at the BDE and PMR models, however, these understandable objections (and other, less well-reasoned ones) are readily dispatched by real-world experience and empirical data.

When all of the costs of having an AASI are weighed against its demonstrated benefits, one might reverse the question and ask, "Can we afford not to invest in an AASI?" By committing to maintaining an appropriate AA subset of our code base now, we stand to gain most of the benefits otherwise possible only through custom client solutions, while improving application-development time, interoperability, and stability. We hope to do better though; in the future, we can eliminate virtually all of the fixed SI costs by promoting our language-based solution and bringing it to fruition by investing in compiler development via the BB20V initiative[70]. Investing in such compiler development — just like investing in the development of self-driving cars — will dramatically

[70]Conceived by John Lakos in early 2018, Bloomberg's 2020 Vision (BB20V) initiative [**?**] is jointly supported by Bloomberg's Chief Technology Officer and its engineering services. BB20V includes a focused effort to bring C++23-like compiler technology (e.g., via GCC and Clang) to Bloomberg well before some features are part of the official C++ Standard through proactive development and deployment (at scale) of four specifically targeted business-critical features, namely concepts [sutton17, romeo18], contracts [[**?**], [**?**], [**?**], [**?**]], modules [[**?**], [**?**], [**?**]], and allocators [[**?**]].

reduce the effort required from infrastructure developers to make software AA while improving both the quality and the performance of the resulting components.

A Likening AA Software To Airline Business Class

The economic advantages afforded by AA software are in many ways analogous to the extra classes of seats available from modern airlines. Before the late 1970s,[71] airlines offered just two classes of service: Economy (also called Coach) and First Class. Given differentiated levels of service, clients in two distinct categories might individually derive increased benefit over access to just a single seat class, thereby improving the potential profit margin for the airline itself. First Class offered maximal comfort and flexibility but at a price few could afford. Economy provided basic service and functionality for those who could not afford the cost of First Class, but those customers might appreciate First-Class–level service were it less expensive.

Software that allocates memory could have analogous categories. First Class would be custom data structures, which offer maximal performance and flexibility managing memory for the few clients who can justify such exorbitant engineering costs. Economy would provide **`new`** and **`delete`** only, i.e., basic memory management for those clients who cannot justify writing custom data structures themselves though they might appreciate the added benefits such customizations (or suitable alternatives) offer were they able to afford them.

To further maximize customer satisfaction and their own profit,[72] airlines subsequently introduced two additional classes of service that compete directly with their respective classical counterparts. Business Class offers most of the comfort and flexibility of First Class at a fraction of the cost, and Premium Economy provides more comfort than Economy at a small additional cost.

The relationships among the new seating categories and the analogous category for software memory allocation are summarized in the table below. For software that allocates memory dynamically, our proposed AASI offers an analogous category that combines the two new classes of seats. Like airline-seat categories, this middle AASI will draw customers from both extremes: **`new`**/**`delete`** and full-custom. Projects that might never have been considered for custom memory management now have another option because the incremental cost (to clients) of exploiting available AA types (i.e., by supplying preexisting specialized memory allocators to AA objects) is small, just like the cost difference between Economy and Premium Economy. Similarly, developers, pressed by performance or other requirements into creating custom data structures, may rejoice in simplifying their development efforts with easy-to-implement, component-based AA solutions (our Business Class service) that effectively address their typical requirements far more economically than any custom (First Class) alternative.

[71] [**?**]

[72] price discrimination in microeconomics

Seat	Cost/Benefit	Analogous Meaning
Economy	Minimal cost but little (if any) flexibility	Creating/using allocator-unaware objects
Premium Economy	Costs slightly more than Economy yet affords substantially increased flexibility	Creating/using AA software rather than (classical) allocator-unaware objects
Business Class	Costs far less than First Class yet affords nearly everything one might reasonably want	Creating/using AA software rather than (classical) bespoke data structures
First Class	As good as it gets but prohibitively expensive	Creating/using bespoke data structures

Just as Business Class greatly lowers the cost for some would-be First-Class customers and Premium Economy lures less indulgent (yet discerning) Economy customers, making an appropriate subset of our software infrastructure allocatoraware would provide essentially all the benefits typically sought from custom solutions at greatly reduced developer costs (depending on the model[73]), thereby impacting both the new/delete and full-custom classical software-client categories.

A.1 Technical Details (for the Mathematically Inclined)

The figure below illustrates the economic picture as an AASI is made available to client developers. Along the x-axis is a curated set of client components, sorted in increasing order (depicted by the heavy dotted line) of the perceived value derived from having that component support the best possible local memory management. Components at the extreme left of the graph would derive little or no benefit from supplying a custom allocator (no matter how easy providing one would be), and components on the extreme right simply cannot fulfill their purpose absent handcrafted, custom-tuned memory allocation. In between these extremes lies a unique threshold percentile a, at which the perceived marginal value of customization first exceeds its marginal cost and rational and capable clients (absent an AASI) would theoretically be indifferent to creating customized, memory management data structures.[74]

Once we introduce an AASI, two other important percentiles on the x-axis materialize. To the left of α is a percentile, α-, below which the perceived (net) marginal value of exploiting an AASI is considered insignificant. To the left of α-, (Economy) clients wisely continue to avoid supplying custom allocators and hence derive no benefit from the new AASI. To the right of α is a percentile α+, above which (First-Class) clients will choose to write their own custom data structures, regardless of any reusable AASI alternative, and hence they too will gain no benefit from our new AASI.

Introducing an AASI adds value enjoyed by clients in the percentile range (α-, α+) who actively choose to exploit the AA aspects. The light gray block in the range (α, α+) shows the consistently significant potential cost savings over providing fullcustom solutions and retains almost the same

[73] Although all of the allocator models are significantly less costly for clients than custom data structures, the C++11 model is still difficult to use and prohibitively expensive for SI implementers. The BDE and PMR models are much less costly for clients and SI implementers alike.

[74] A variant of this graph, with thorough narration, appears in [**?**] starting at 4:50.

derived value.[75] (Compare this case to the cost savings of choosing Business Class rather than First Class.) The dark gray area under the potential-value curve in the range (α-, α+) shows the varying increased value derived from clients choosing to readily pass existing allocators into our new AASI compared to willfully choosing not to do so at almost the same client development costs.[76] (Compare this case to the increased value of choosing Premium Economy over Economy).

With the new AASI in place, client components naturally group into three distinct categories characterized by the solid, step-shaped gray line (increasing left-to-right) depicting (1) making do without flexible memory allocation (Economy class), (2) economically exploiting AASI (Business and Premium Economy classes), and (3) writing very expensive custom data structures (First Class). Category (1) is smaller than Classic Economy because the incremental cost of achieving near first-class value is sufficiently reduced by our new AASI such that a non-negligible proportion of Classic-Economy clients will appreciate the value proposition and opt into category (2). We assert that category (3) is tiny compared to Classic First Class because only those components having truly extreme performance requirements or those in need of special memory addressing (e.g., shared memory, proxies) would remain. Ideally, all client components that would naturally fall into category (2) would actively exploit AASI by providing (typically) pre-existing custom allocators as appropriate.

A.2 Appendix Summary

We assert that virtually all reasonable clients who, absent an AASI, would have chosen to write their own data structures would now instead opt for a readily available AASI (or at least use it for initial prototyping). On the other hand, we cannot know just how many Classic-Economy clients would voluntarily choose to exploit an AASI if they themselves did not perceive the need for the added efficiency. For this latter type of client, the benefit of passing in custom allocators becomes defuse ("It saves the company money on hardware with slightly more effort on my part") rather than immediate ("I need a high level of performance, flexibility of placement, instrumentation, and so on for my project to succeed!"). Hence, any active use of an AASI becomes altruistic. We suspect that the extent of such use of local allocators by otherwise Classic-Economy clients would be heavily influenced by (1) the client's ease-of-use of the particular allocator model employed and (2) the sophistication and training of clients, where training might need to involve inculcating a certain culture of altruism.

So far, we have discussed cost/benefit entirely from the *application clients*' perspective. To support this new class of AA service, significant additional (albeit comparatively fixed) per-*library*-component development and maintenance costs would be borne by SI developers (analogous to that for an airline) but, unlike the airline industry, a large-scale software company (such as Bloomberg) comprises both SI along with its own application clients; hence, the company as a whole benefits

[75] Note that the relative heights of the cost and value curves shown in the graph are irrelevant because they are measured on separate vertical axes (having distinct units that are not even directly comparable). Moreover, this graph is not to scale (the vertical axes are not even labeled) because value and cost can be evaluated and traded off in myriad ways. What's more, the actual client development costs will vary substantially, depending on which allocator model is chosen (C++11 or BDE/PMR).

[76] Also note that the cost of developing the AASI itself is not pictured in the graph. This image is intended to suggest qualitatively the enormous potential value and relatively small incremental cost for application clients' using a company-provided AA infrastructure once those fixed costs have been paid compared to ongoing (marginal) client costs associated with developing custom data structures ad hoc.

directly whenever its SI customers are more productive. To reduce per-component costs for SI and clients alike, we need to invest heavily in new compiler technology that might someday automate the task of creating AA components.[77] Bloomberg's `bde_verify` tool already supports an approximation of what will eventually become an ultra-efficient, language-supported model for realizing AA software at essentially no additional cost over a (classical) allocator-unaware SI, thereby eliminating any plausible argument against supporting ubiquitous control over fine-grained memory management throughout any library software from which clients might eventually benefit

[77] Cars (like C++20 compilers in our analogy) enable a licensed driver (experienced programmer) to get to a desired destination (efficient application-specific custom memory management). Yet, with each new journey, the car's owner (company comprising application and library developers) must bear a substantial ongoing and recurring opportunity cost: Either the owner (an application developer) must personally drive the car (write bespoke data structures) or pay a chauffeur (a library developer) to drive the car (make each SI component AA). Though still unproven, self-driving-car technology is almost universally considered to soon be widely available. When that happens, the cost of designing, making, and deploying self-driving cars will be scarcely noticeable over that of its nonself-driving predecessor. Moreover, any human opportunity costs associated with creating or using these modern cars will essentially disappear, rendering conventional (human-driven) cars of today entirely obsolete. What's more, as this burgeoning technology continues to improve over time, so too will the risk — i.e., the accident (defect and/or crash) rate due to the unchanging inevitability of human driver (programmer) error. tl;dr: Compiler support for AA software in C++ is a game changer.

References