Getting a Handle on Unmanaged Memory

Anonymous Author(s)

Abstract

The inability to relocate objects in unmanaged languages brings with it a menagerie of problems. Perhaps the most impactful is memory fragmentation, which has long plagued applications such as databases and web servers. These issues either fester or require Herculean programmer effort to address on a per-application basis because, in general, ***heap objects cannot be moved in unmanaged languages***. In contrast, managed languages like C# cleanly address fragmentation through the use of compacting garbage collection techniques built upon heap object movement. In this work, we bridge this gap between unmanaged and managed languages through the use of handles, a level of indirection allowing heap object movement. Handles open the door to seamlessly employing runtime features from managed languages in ***existing, unmodified code*** written in unmanaged languages. We describe a new compiler and runtime system, Alaska, that acts as a drop-in replacement for `malloc`. ***Without any programmer effort***, the Alaska compiler transforms pointer-based code to utilize handles, with optimizations to minimize performance impact. A codesigned runtime system manages this new level of indirection and exploits heap object movement via an extensible service interface. We investigate the overheads of Alaska on large benchmarks and applications spanning multiple domains. To show the power and extensibility of handles, we use Alaska to eliminate fragmentation on the heap through compaction, reducing memory usage by up to 40% in Redis.

1 Introduction

A vast amount of code has been written in unmanaged languages such as C, C++, Fortran, and others. The billions of lines of code written in these languages are not going anywhere. C, C++, and Fortran are the 2nd, 3rd, and 14th hottest languages in the TIOBE index [10]. A major draw of these unmanaged languages, particularly C, is the direct control over memory management granted to developers. Even in application code, this control is nearly at the machine level, managing raw memory addresses. The programmer can carefully control object placement and lifetime, even specifying the representation of a pointer. For example, one can freely encode type information into unused address bits, store pointers as integers, multiplex pointers in an XOR linked list, or even send and receive pointers over a network.

This power brings with it the potential for bugs and security vulnerabilities, almost as if the programmer *were* working at the machine level. In this work we focus on the *limitations* of memory management in unmanaged languages compared to managed languages. In particular, managed languages have their own clear advantage: heap objects in managed languages can be straightforwardly moved by the runtime system.

The intentionally designed Wild West semantics of pointers in unmanaged languages makes it *virtually impossible* for the language runtime system to move a heap object. In a managed language, it is possible to algorithmically identify all pointers to an object, and thus patch them when the object is moved. To do this in an unmanaged language for an arbitrary program would require understanding pointer semantics *as they are for that specific program.* For example, a memory manager would need to be able to patch through XOR list encodings, union types, and address masking. Generally finding direct pointers is challenging enough. Generally finding bespoke-encoded pointers is beyond the pale.

The inability of an unmanaged language runtime to easily move heap objects makes their initial placement on the heap a matter of great importance. This has lead to a wide range of heuristics and compiler-driven allocation strategies [32]. If the initial placement is wrong, it can have cascading consequences throughout the lifetime of the badly placed object. One such consequence is external fragmentation of the heap, which often results in using far more physical memory to run the program over the long term than would have been possible with an oracle placement.

To complicate matters further, heap fragmentation often occurs in phases, and thus heap object placement decisions made in one phase of the program could very well induce fragmentation in another phase, even if a perfect oracle was available. It has been shown that *any* allocation strategy that is not free to relocate objects will suffer from fragmentation [37]. This reality forces the programmer to act.

The first option is to adopt the ostrich algorithm[1] and allow fragmentation to fester, punting the resulting memory usage issue down the road. This forces the user to deal with unexpected results such as application restarts or the kernel's out-of-memory killer. Worse even, the user may need to *pay for more memory* on a cloud hosting service just to run their fragmented application.

The second option is for the programmer to attack the fragmentation problem head on with an ad-hoc defragmentation strategy. For example, the programmers behind Redis, a popular in-memory key-value database system, have added `activedefrag` in version 4.0 [5, 6]. Because unmanaged pointers are so hairy and have tendrils everywhere, `activedefrag` required the addition of thousands of lines of code to handle all edge cases, mind you, just the ones in Redis. Through this considerable engineering effort, `activedefrag`

[1] To stick one's head in the sand and ignore the problem.

enables Redis to tackle fragmentation through modifications to the allocator (jemalloc), polling the *fragmentation ratio* (RSS over heap usage) of the program once a second to decide if it should defragment. This approach tends to reduce fragmentation substantially. Regrettably, the benefits of such a hand-crafted system cannot be transferred to other applications without a great deal of reengineering to handle new data structures and pointer semantics.

In contrast, programmers in higher-level managed languages such as Java or C# have long enjoyed *transparent* object relocation and heap compaction, allowing them to focus on application features rather than worrying about where objects are located. This is possible due to the guarantees about how pointers are used and stored—having very strict pointer semantics throughout the language. This vastly simplifies the operation of patching references when a heap object is moved, and the runtimes of these languages often utilize read/write barriers, safe pointing, and forwarding pointers to increase efficiency [28, 34, 35, 47].

Recent work addresses the general problem of fragmentation in unmanaged languages *without* object mobility. Mesh [36] leverages paging to map multiple heap extents in the virtual address space to the same extent in the physical address space. The allocator is co-designed with this capability to locate objects (permanently) in the virtual heap extents such that there are no collisions in the physical address space, and that theoretical guarantees can be provided about the packing of the objects into the physical address space. In effect, the virtual heap is now much larger with considerable sparsity, but the physical heap is dense. We argue that Mesh does not solve the fundamental problem of object mobility, inherently limiting the defragmentation it can achieve.

In this work, we argue that it is possible to add the generic capability to move heap objects to the Wild West of unmanaged languages through the use of *handles*. This generic object movement capability in turn allows for the design of effective, application-independent defragmentation, and opens the door for other services. Adding this generic ability can be done in a manner transparent to the programmer, even if they make use of complex, custom pointer semantics — the hallmark of unmanaged languages.

The concept of handles, elaborated on in §2, dates back to at least the early days of personal computers, prior to the inclusion of virtual memory. In handle-based memory management, the allocator doles out opaque handles instead of raw pointers to heap allocations. Handles must be pinned by the application before use, and then unpinned afterwards. Pinning provides a (current) raw pointer to the object. If a handle is unpinned, its corresponding heap object can be moved. In classic Windows and MacOS, all applications shared a single heap managed in this manner, which was regularly compacted. Unfortunately, handles were a directly visible feature of these systems, and could be easily misused by programmers, bringing down the whole house of cards.

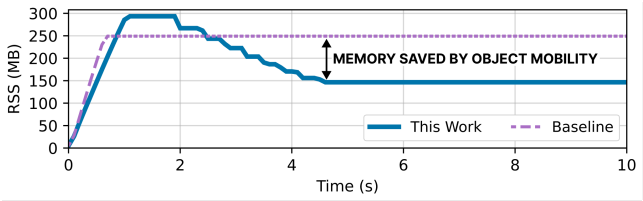

Figure 1. Through object mobility, Alaska can save considerable memory — up to 40% in Redis.

Our work has three key differences. First, we show how the Herculean effort of using handles correctly and efficiently is avoided entirely through compiler analysis, transformation and optimizations. Second, our method makes handles *entirely transparent*. Programmers can simply write pointer-based code with whatever specialized semantics that fit their fancy, with handle-based code running under the hood. Even better, the programmer cannot possibly get handles wrong — because they don't even see them — lifting the Sisyphean burden of handle-based code maintenance from their shoulders. This transparency also means our work can be directly applied to *existing, unmodified applications*. Finally, we show how a careful compiler/runtime co-design can implement transparent handles with low overhead — a surprising result as handles add a layer of indirection.

This paper makes the following contributions:

- We make the case for revisiting handle-based memory management as a mechanism for bringing general purpose heap object mobility from managed languages to unmanaged languages, particularly C and C++. (§2)
- We describe the design of a handle-based memory management system that relies on compiler/runtime co-design to *automate* the use of handles, and to make the use of handles *transparent* to the programmer (§3) We implement Alaska, an extensible proof-of-concept system for C and C++ (§4).
- We evaluate the overhead of Alaska on a number of well-established C and C++ benchmarks and applications with *zero code modifications*. The 8% overhead (geomean) suggests the practicality of automatic transparent handle-based memory management (§5). We show that the software engineering effort of Alaska is on par with custom application-specific mechanisms for object mobility (§5.3).
- We implement Anchorage, a memory allocator that enables dynamic runtime heap compaction in unmodified C and C++ applications on top of Alaska (§4.3).
- We show that Anchorage can reduce memory usage in Redis by up to 40% through compaction, on par with activedefrag (§5.5, also Figure 1).

Alaska and the test suite to reproduce results will be made publicly available as an open source project upon publication.

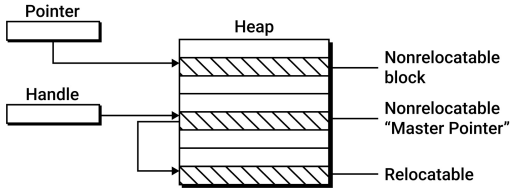

Figure 2. MacOS handles pointed to a relocatable block.

2 Handle-Based Memory Management

We now describe classic handle-based memory management, whose motivation to move heap objects mirrors our own. Unfortunately, in classic systems, this capability came with high programmer effort and, *because it was manual*, the possibility for disastrous programmer error.

2.1 Manual Handles in The Past

Handles were originally a solution to a hardware problem of their time, namely that early Motorola and Intel CPUs which did not provide hardware MMUs. Without the hardware underpinnings of virtual memory, PC operating systems, such as early versions of classic MacOS and Windows, required an alternative. Classic MacOS had only *one (physical) address space*, with a single global heap shared between applications [27]. Thus, if one application created fragmentation on the heap, the entire system felt the consequences–*including the kernel*. This was especially problematic given the memory restrictions of the time: the original Mac had only 128 KiB of RAM. Handles enabled these systems to more efficiently use their already limited memory through defragmentation.

In this system, an allocated block in the global heap may either be *relocatable* or *nonrelocatable*. Relocatable blocks could be moved within the heap to make space or reduce fragmentation through compaction so long as all users followed the contract: these blocks may be either *locked*,[2] preventing movement, or *unlocked*, permitting movement.[3] When a relocatable block was allocated, the memory manager would create and maintain a single nonrelocatable *master pointer* to that block. A pointer to this master pointer was returned to the programmer, being a handle that must be translated before use. This relationship is shown in Figure 2.

Simply put, handles provided a single level of indirection between logical references to an application's heap data and the actual backing storage. This indirection *allowed objects to be freely moved on the heap by only patching a single reference*, the master pointer.

[2]We adopt the classic terminology here, but it is important to keep in mind that handle locking is unrelated to concurrency control. The modern terminology is *pin*.

[3]Beyond movement, blocks could also be marked "purgeable", allowing the block to be entirely discarded to make space (e.g., a code block). Some systems, such as Windows, also allowed "swappable" block, locking a handle to these blocks would trigger a fetch for their contents. Using such a block without first locking its handle would cause disaster.

```
1 int **handle = NewHandle(...);
2 HLock(handle); // Lock/pin
3 int *ptr = *handle; // Translate
4 *ptr = 42; // Use memory
5 HUnlock(handle); // Unlock/unpin
```

Figure 3. Manual handles required significant effort with uncomposable changes that permeate the application.

2.2 Manual Handles Are Cumbersome

While handles gave the system a great deal of control with regards to object placement, manual handles were considered a nightmare to program for several reasons.

Figure 3 shows an example use of MacOS handles (translated from the original Pascal to C). Manually adding handles to a program required significant effort—the majority of the lines shown are concerned with their management. Handles are allocated with the `NewHandle` function. In order to safely access the data behind the handle the user must first call `HLock`, setting a bit in the handle to indicate it has been locked/pinned. This allows the programmer to safely access the backing memory of the handle without concern for object movement. When done accessing the object, programmers must call `HUnlock`.

With manual handles, programmers were required to consider an additional lifetime—that of their pins—which came with tradeoffs. Being overly cautious with pins (i.e., surrounding individual memory accesses with pin/unpin) is easier to understand, but produces terrible performance from the additional memory updates needed to pin, unpin and translate handles. Conversely, pinning across large intervals (e.g., an entire loop or function) reduces the performance issue, but may result in many nonrelocatable blocks, limiting the effectiveness of compaction when needed.

In addition to thinking about when to free memory—avoiding double frees, use-after-free, and memory leaks—the programmer must also be concerned with the effects of unpinning. When an object is unpinned, it *can be moved*; unpinning too early could lead to the object being moved without the application's knowing. The programmer also had to ensure they were not unpinning a handle multiple times, and that they unpin before the handle goes out of scope, lest it be pinned forever.

Finally, manual handle-based memory management puts the onus of correctness entirely on the programmer. Worse, on a modern system, programmers would need to consider correctness in the presence of preemptive threads and signals, neither of which existed in classic systems.

3 Design of the Alaska System

We assert that automatic handle-based memory management holds the key to achieving object mobility in unmanaged

languages, empowering the runtime to freely relocate heap objects and more. Guided by this, we present the design of Alaska – a compiler and runtime system whose goal is to achieve *automatic transparent handles*. Handles managed by Alaska gracefully coexist with pointers. This seamless integration ensures that programmers can unknowingly wield handles with confidence just as they would traditional pointers. This also means that **entirely unmodified applications can now automatically leverage handles**.

3.1 Design Goals

A key objective of Alaska is to enable movement of heap objects using handles with zero additional programmer effort. Such a system would allow the billions of lines of code written in unmanaged languages to benefit from managed techniques currently held out of reach.

Handles and pointers coexist. In order to support all existing applications written in unmanaged languages, raw pointers and handles must coexist. This is necessary because a function written to accept a pointer must behave in the same way regardless of it being passed a pointer or a handle. Thus, a variable can hold either a raw pointer—as it would originally—or a handle. As far as the programmer is aware, then, handles are effectively pointers with no additional semantics.

Handles have the same semantics as pointers. To maintain the functionality of the original application, handles *must* have the same programmer-visible semantics as the pointers they replace. This means that handles must support the majority of pointer encoding and multiplexing techniques outlined in §1 that keen programmers may employ. So long as the application does not violate the assumptions outlined in §3.2, no changes must be made in most applications to use handles.[4]

The compiler does translation for you. The most important component of Alaska is the compiler, which manages the translation and tracking of handles for the programmer. Unlike the handles of old, described in §2, the programmer should not need to modify their application in any way to take advantage of the benefits of handles. This is achieved via the Alaska compiler, which automatically translates handles and ensures they are both correct and optimized.

As discussed in §2, small pin intervals (individual load/-stores) grant the most freedom for memory management, but can incur a worrying performance overhead. Conversely, large pin intervals (loops or functions) reduce the performance impact of pinning, but reduce the degrees of freedom available to the memory manager. Through the compiler, the tradeoff between small and large intervals can be explored automatically, leaving one less dragon looming over the development process. We present one such optimization to hoist pins outside of loops when beneficial—detailed in §4.1.

The runtime efficiently tracks pins for you. Alaska's runtime automatically manages the tracking of which handles are in use and which are not. However, a runtime system that naïvely records pins as in Figure 2 would perform poorly today in multithreaded environments. Concurrent updates to the shared structures using atomics would lead to contention across the machine, especially when handle pins occur at a high rate. The runtime system must correctly handle such concurrent access without significantly affecting other concurrency control logic within the program itself.

3.2 Assumptions

While handles in Alaska do not affect the programmer-visible semantics of pointers, Alaska imposes a limited set of restrictions on their usage, to enable a more performant implementation. We argue that these assumptions are not fundamental to transparent handles, but are rather design decisions specific to our implementation.

We assume that a program will not access memory outside the bounds of the allocation returned by the allocator. If the assumption is not true, Alaska can potentially translate the wrong handle, or generate an out-of-bounds access. This is the same assumption made by LLVM's memory model [31] and can be considered a requirement for any meaningful memory transformation. Similarly, the optimizations in our compiler rely on the application not breaking strict aliasing assumptions, which GCC and Perlbench from SPEC unfortunately do [7, 8].

We also assume that programs do not rely on the bit representation of a pointer outside of the alignment guarantees specified by `malloc`[5]. With this, implementations are permitted to utilize lower address bits for their own purposes. They are not, however, allowed to assume the top bits of a pointer are free to use (as they are on x86). This unfortunately rules out the use of NaN-boxed handles, as such systems often rely on virtual addresses being only 48 bits.

The final assumption of the system relates to external libraries: we assume that either all library code is subject to our compilation, or that it does not store pointers to the backing memory nor free that memory. More on this assumption and our approach for handling it are discussed in §4.1.4.

3.3 Efficient Handle Translation

Alaska is built as a pure software solution, meant to run on commodity hardware. Without hardware acceleration, as is the case for virtual memory, *translation* from handles to pointers must be performed in software using standard

[4] As our implementation is at the IR level, this is probably also true for other languages, but has not been tested yet.

[5] In our implementation we relax this considerably. Our handle representation has an "alignment" of 4 GiB instead of just the `malloc/memalign` alignment, so this should not be a significant concern.

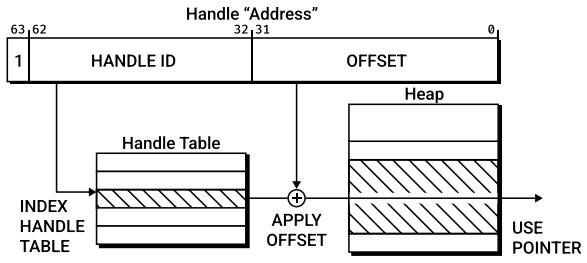

Figure 4. Handles in Alaska are pointers (hardware-level addresses) with the top bit set. The Handle ID is used to index into the handle table, and offset is added to the resulting address.

```
1    cmp   -2,%rdi      ; Handle check
2    jg    skip         ; Not a handle
3    mov   %rdi,%rax
4    shr   0x1d,%rax    ; Extract handle ID
5    mov   %edi,%edi    ; Truncate offset
6    add   (%rax),%rdi  ; HTE Load + offset
7  skip:
8    mov   (%rdi),%rdx  ; Perform access
```

Figure 5. x64 instructions to perform a handle translation.

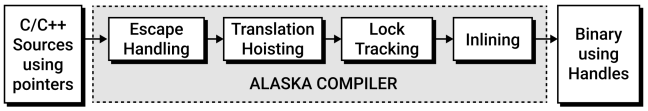

Figure 6. Alaska's transformation pipeline.

instructions. A poor design could be catastrophic for performance. As such, design of the handle representation leans upon the understanding of how pins will be lowered into the ISA, minimizing the number of additional loads.[6] We specifically consider x64, ARMv8.3, and RISC-V 64 bit architectures in our design–but only evaluate x64 for brevity.

Given these requirements, we chose the bit representation for handles as shown in Figure 4. With this representation, a handle in Alaska is differentiated from a pointer by the top bit—being set to 1 for a handle, or 0 for a pointer[7].

If the top bit is set, bits 32 to 62 represent the *handle ID*, which acts as an offset into the *handle table* data structure. Conceptually, this is the same way a virtual address is broken into offsets into the various levels of the page table hierarchy, but with only a single level. Each allocation in the system has a unique handle ID, and the design effectively limits the number of active handles in the system to 2^{31}. The decision to restrict the number of bits in the handle ID is influenced by many architectures' ability to efficiently truncate values to 32 bits, making it a practical choice. The handle representation's lowest 32 bits are used to denote an *offset* into the object, capping the maximum handle size to 4GiB. We contend that this limitation is not a significant concern, as relocating objects larger than 4KiB can be more efficiently handled by paging.

These design decisions allow us to implement Alaska's handle translation logic in only 6 instructions on x64 (Figure 5). We investigate the overhead of these additional instructions in §5.

3.4 Efficiently Tracking Pinned Handles

On top of managing translations, a key functionality of Alaska is to record which objects are in use and which are not, a process referred to as "pinning". This is required because Alaska's compiler transformation will leave references to the raw *backing memory* in CPU registers or spilled on the stack. As such, pinned handles are viewed as a constraint in the runtime system, and cannot be moved.

3.5 Extensible Service Interface

Beyond the core functionality of the Alaska runtime, an extensible interface allows for the addition of *services*: pluggable and configurable libraries that manage the allocation and movement of objects. The separation of these services from the core runtime enables exploration into different techniques that can make use of the object mobility provided by handles (§7). In this paper we developed one such service, Anchorage, to perform defragmentation. Anchorage provides Alaska with an allocator and barrier routine designed to perform heap compaction. We describe it's implementation details in §4.3.

4 Implementation of Alaska

The Alaska System is comprised of three logical components: a compiler that transparently automates the use of handles in unmodified pointer-based code, a core runtime that manages the handle table and tracking, and a generic interface to allow the construction of services such as Anchorage.

4.1 The Alaska Compiler

Alaska's compiler transforms programs to use handles by replacing allocation routines (§4.1.1), optimizing the translation of handles (§4.1.2), and inserting tracking for pinned handles (§4.1.3). It also maintains correctness with external libraries through an *escape pass* (§4.1.4). We operate on the LLVM IR [31] and use abstractions from NOELLE [33].

4.1.1 Replacing malloc with halloc. Conversion of memory allocations to handle allocations is straightforward. Each

[6] This means we can't use a traditional radix tree as seen in virtual memory, as it would quadruple memory accesses.

[7] Note that on all architectures considered, this will result in a handle being accidentally used as an address to fault.

call to `malloc` and `free` is transformed by the compiler to a call to `halloc` and `hfree`–their Alaska counterparts. This is also the case for proxy functions such as `calloc` and `realloc`. We perform this replacement in the compiler and not the linker to avoid introducing handles into code that is not visible to Alaska. This behavior can be disabled to allow the programmer to decide which allocations are made with `malloc` and which with `halloc`. In our evaluation we force handles on all allocations through `malloc`.

4.1.2 Automatically Translating Handles. The translation insertion compiler pass transforms the program so that each instruction that *may* access a handle-allocated object *must* perform a translation beforehand. To simplify this transformation, Alaska's translation function has different functionalities depending on the incoming value. If the incoming value is a pointer, the translation function simply does nothing and returns the pointer. If it is a handle, the function performs the translation as described in Figure 4.

A naïve implementation of translation insertion would be to translate immediately before each memory access in the program. However, this would incur a large runtime overhead: a bit test for every memory access in the best case and a load from the handle table in the worst case. To prevent this, we present Algorithm 1, which analyzes and transforms an LLVM program to minimize the number of dynamic translations. The shorthand $ptr(inst)$ represents either the address of load/stores, results of a `translate`, or the operand of transient values (i.e. ϕ and `getelementptr`).

Algorithm 1 The translation insertion algorithm.

procedure TranslationInsertion(F : function)
  $PG \leftarrow$ pointer flow graph of F
  $PG' \leftarrow \{p \in PG \mid \text{incoming}(p) > 0\}$
  $DT \leftarrow$ dominator tree of F
  $DF \leftarrow \{n \in DT \mid n \in PG'\}$ ▷ A dominator forest.
  for all $t \in DF$ **do** ▷ For all trees in forest.
    $r \leftarrow \text{root}(t)$
    $l \leftarrow$ Translate(r, F)
    for all $n \in t$ **do**
      $ptr(n) \leftarrow l$ ▷ Replace handle with the translation.
procedure Translate(i, F)
  $LT \leftarrow$ loop nesting tree of F
  $L \leftarrow \{L \in LT \mid L$ is the innermost loop containing $i\}$
  $L' \leftarrow$ FindNestingLoop(L, LT, i)
  $BB \leftarrow$ preheader of L'
  $t \leftarrow$ terminator of BB
  Insert `translate(ptr(`i`))` immediately before t
  return The result of `translate(ptr(`i`))`
function FindNestingLoop(L, LT, i)
  $L' \leftarrow \text{parent}(L) \in LT$
  if $i \in L' \ \wedge \text{ptr}(i) \notin L'$ **then return** L'
  else if $i \notin L'$ **then return** L
  else return FindNestingLoop(L', LT, i)

This transformation ensures that each memory access to a handle will operate on the translated pointer to its backing memory as each access is dominated by a `pin`. For memory accesses within loops, InsertPin hoists the requisite pins outside of loops when possible. This pass relies on LLVM's canonical loop form (`-loop-simplify`).

Following the insertion of translations in the program, the compiler inserts *releases* which indicate when the handle is no longer in use. These translations are only inserted to simplify the implementation of the tracking pass later (§4.1.3), and are removed before the program is run. This is performed with the results of a liveness analysis [13] on each of the translated handles. For each `ptr = translate(handle)` inserted into the program, `release(handle)` calls are inserted immediately at the end of `ptr`'s lifetime.

4.1.3 Tracking Pinned Handles. Once a handle is translated, it must be pinned to ensure that the backing memory block represented by the handle cannot be relocated for the lifetime of the translation. This is required because during the translation's lifetime, raw pointers (i.e., virtual addresses) to the backing memory exist, for example in CPU registers. In the common case most applications do not have a large number of pinned handles at any point in time, and thus the runtime is free to move most objects at any time.

An intuitive, but naïve implementation of tracking pinned handles might atomically increment a `pin_count` attached to each handle in the handle table when a translation occurs, and atomically decrement it when the translation's lifetime ends. When `pin_count` > 0, the handle would be considered pinned and the backing memory would be immobile. Unfortunately, these pin/unpin steps would naturally incur undue overhead in applications that exhibit many translations–especially in a multithreaded application, and as core counts grow. An alternative mechanism is necessary.

In Alaska, pinned handles are tracked *privately*, on the call stack, requiring *no atomic instructions*. For any function that translates handles, we generate entry code that allocates a single *pin set* in the current stack frame. At runtime, each invocation of the function thus has a private pin set. The pin set stores the translated handles for the invocation. The compilation statically decides the size of each function's pin set so that is large enough to contain at least as many pinned handles as may statically overlap at any point in the function's control flow. Each *static* translation in the program is allocated a single entry in the function's pin set using a greedy interference graph-based allocation strategy similar to a register allocation algorithm. At runtime, before a handle is translated, the handle is stored in the in the pin set.

The pin set is stored as an array on the stack, requiring no additional instructions to construct and imparting no additional heap memory usage. An experimental feature of LLVM's garbage collector infrastructure, StackMaps [9], is

used to record the array's location relative to the stack or frame pointers (e.g., `%rsp` or `%rbp`) for each return address in the program. This information can then be used during runtime to walk the current stack with Libunwind [2] to find the corresponding stack of pin sets that is embedded in it.

Barriers and Pin Set Unification. Because each thread tracks its pin sets privately, there must be a mechanism to pause all threads and unify all their extant pin sets into a global pin set before using the information in that set to determine which backing memory blocks are currently immobile. We refer to this mechanism as a *barrier* to reflect our current implementation: a stop-the-world pause event, during which the runtime is free to relocate objects. Because LLVM StackMaps are only valid at certain points in the program, the threads cannot be simply interrupted using POSIX signals. As such, Alaska uses safepointing and polling to ensure that the local pin set is valid at certain points in the program [11, 12].

The Alaska compiler inserts calls to an LLVM safepoint intrinsic provided by the StackMaps infrastructure that dictate points at which the StackMaps information must be valid. We place safepoints throughout the program on the backedges of loops, the entry point of certain functions, and before calls to external libraries. The LLVM backend recognizes these safepoints and emits a *patch point* at each corresponding ISA-specific location. On x64 a patch point is a `NOP` instruction. In the best case, these safepoints have no overhead and produce no register or data cache pressure. However, as is to be discussed in §5, this API is *experimental*, and unfortunately does not have zero overhead in all cases.

When the runtime wishes to unify pin sets, a barrier is started and each patch point's `NOP` instruction is replaced with a `UD2` instruction, which, when executed, causes an illegal instruction exception that the kernel in turn delivers to the thread's `SIGILL` handler, which is part of the Alaska runtime. In this signal handler, the StackMaps information is valid, and all pin sets can be parsed safely.

This would be sufficient if all code were transformed by the Alaska compiler. However, as a practical matter, we often need to support external library code, and, at minimum, the system call wrappers need to considered. Here, the problem is that there are no safepoints, and, in some cases, blocking in the kernel, e.g., during an I/O operation, can also occur. Consequently, we might end up waiting for an arbitrary amount of time. To handle this situation, the barrier mechanism does not simply wait forever for all threads to join. Instead, if a straggler is detected, that thread is signaled using a POSIX signal, and the handler for that signal effectively contains the safepoint. This works because there is no handle translation occurring within the external code, and thus no pin sets can exist "below" the immediate external call, no matter how deep the call stack is below that point.

Once all threads are in the runtime, they synchronize and determine which handles are pinned and which are not. The runtime is then free to move backing memory blocks however it sees fit, so long as it obeys the pin status of each handle. When done, the runtime returns each patch point to it's original `NOP` state, and all threads are resumed.

4.1.4 External Functions.

The Alaska compiler assumes a whole-program representation, but calls to precompiled libraries such as libc are commonplace. The most common implementation, `glibc` [1] cannot be compiled with `clang` due to its reliance on GNU extensions. This leaves the compiler with code that *may* break the assumptions in §3.2. Rather than prohibiting users from using `glibc`, we handle cases of broken assumptions in turn.

For external functions, the compiler performs *escape handling*. An *escape* occurs whenever a handle is passed into precompiled code as an argument. For each escaped handle, the compiler inserts a `pin` before the call and passes the resulting, raw pointer to the function. This ensures that the precompiled code will operate correctly.

For cases where assumptions are broken, the compiler must transform the function. An example of this can be seen in parsing applications, where the size of a token is computed by subtracting the non-handle result from the escaped handle argument of `strstr`. This results in integer underflow and often leads to a segmentation fault. To remedy this, the Alaska compiler replaces the external function with its `musl` [4] implementation and transforms it with the rest of the program. With this solution, `strstr` operates as expected and the aforementioned issue is avoided.

4.2 The Core Runtime

Alaska's runtime manages low level operations, including handle allocation/deallocation, pin/unpin tracking, and the delivery of "barrier" operations. Handle allocation is exposed to the programmer through two functions, `halloc` and `hfree`, which mirror the functionality of `malloc/free` respectively. These allocation functions are automatically used in place of the system allocation functions in any code that is transformed by the compiler, as mentioned above.

4.2.1 The Handle Table.

At the center of the core runtime is the *handle table*, a metadata structure which enables efficient handle translation. The *handle table* is analogous to the page table in virtual memory systems, albeit with one handle table entry (HTE) per-object instead of per-page. Unlike a hierarchical page table in x86, ARM or any other modern system, the handle table in Alaska is a single-level table. This eliminates the need for page-walk-style translations in software, which would impart unreasonable performance overheads. A simple linear array enables translations to occur with a single load.

The handle table lives in the virtual address space of the user program, and all translations are performed in software using unprivileged instructions. The table is placed at a specific virtual address so that translations need not mask off the

top bit of the handle representation (Figure 4). To ensure the handle table is placed at the right location and *can* use all 2^{31} entries it is virtually allocated via `mmap` in its entirety at the start of program execution. This ensures that no other memory mappings will block the expansion of the table. Generic *demand paging* is used to do actual memory allocation to support the table as it grows. This is important as the table *cannot be moved* once in use.

For translation, an HTE must contain a pointer to the backing memory of a given handle. This imparts about eight bytes of overhead per object. It would be possible to reduce this overhead by compressing the pointer representation as done in prior work [16], but we have not investigated this potential. Because all entries of the handle table are the same size, allocation of entries is $O(1)$. At startup, handle table entries are allocated according to a bump allocator strategy starting from index zero. When a given HTE is deallocated it is added to a free list for quick reuse by future allocation requests before. The free list is consulted before bump allocation is used.

While the handle table can always fulfill HTE allocation requests if it has space, it can suffer internal fragmentation from the kernel's perspective. In the worst case, where the kernel is allocating 2 MiB pages, an adversarial allocation/free pattern could result in a single active HTE per page. This is unlikely to occur in practice, and active HTE density is quite high in our evaluation. Regardless, defragmenting the handle table at the page granularity could be addressed with Mesh [36].

4.2.2 The Service Interface.

As described in §3.5, Alaska does not manage the allocation of backing memory itself. Instead, it defers this task to *the service*. With services, Alaska enables a testing ground for future research into the benefits and capabilities of handle-based memory management. The service interface consists of eight callback functions: two lifetime management functions (`init`/`deinit`), two backing memory management functions (`alloc`/`free`), and four metadata functions (e.g., query the size of an object). When the application calls `halloc`, Alaska allocates a handle from the handle table and then requests backing memory from the service via the `alloc` callback. The service can later invoke Alaska to easily query pin status and other information when moving objects.

4.3 Battling Fragmentation with Anchorage

The Anchorage service uses Alaska to implement a compacting heap allocator that relies on object movement.

Allocator Anchorage's allocator is designed with its freedom to perform heap compaction in mind. As such, Anchorage's allocator is much simpler than its modern counterparts, which have no choice but to include complex techniques to avoid fragmentation.

Anchorage uses a naïve bump allocator, and reuses freed memory through the use of a simple power-of-two free-list. When an allocation is made, this list is searched for an appropriate block in $O(1)$ time (only the front of the list is checked). If no block is found, allocation is made by bumping a pointer at the top of the heap. This simple design does not feature the thread caches and other initial-placement optimizations seen in modern allocators. However, this is merely an engineering limitation.

Anchorage subdivides its heap into multiple sub-heaps similar to a classical semi-space garbage collector[25]. Allocations are made in one subheap by first checking the free list before falling back to bump allocation. When that subheap cannot fulfill an allocation request or heap fragmentation is deemed too high, Anchorage triggers a *barrier* and performs compaction. The choice of this heap layout stems primarily from it's simplicity and ability to easily handle immovable, pinned, objects on the heap – a feature that poses an engineering challenge in a more conventional "sliding" approach to compaction such as Jonker's algorithm [30].

When the runtime barrier fires, all local pin sets are unified and compaction commences. Unpinned objects are moved from the top of one subheap (the from-space) into another subheap (the to-space) by simply copying their contents and patching HTEs. Dictated by a control algorithm (below), Anchorage can perform *partial compaction* to amortize the cost of relocating the heap across several pauses.

A problem with this subheap design is the potential doubling of the memory usage by the system. To combat this, after every round of compaction (both partial and complete) the from-space has as much of its memory returned to the kernel as possible using `MADV_DONTNEED`. This marks the pages of virtual memory as unneeded, and the kernel is free to reclaim them if memory is needed for another process. With this, resident set size (total physical memory used by a process) increases only momentarily during compaction, and is quickly reduced. Unfortunately, invoking the kernel with `madvise` each round of compaction can result in additional TLB shootdowns in multithreaded applications. A batched technique similar to jemalloc's memory reclamation system could be implemented to reduce this at the cost of additional memory usage [24].

Control system Anchorage can perform a compaction pass at any time, but the cost of a pass, and the rate at which they occur constitutes a performance overhead. In this section, we describe Anchorage's control algorithm to balance the goal of reducing fragmentation with reducing overhead. This algorithm measures fragmentation using an $O(1)$ metric: the virtual extent of the heap divided by total size of active objects.

The algorithm attempts to keep *fragmentation* and the *fraction of time spent compacting* within bounds set by the operator, $[F_{lb}, F_{ub}]$ and $[O_{lb}, O_{ub}]$, respectively. Both upper and lower bounds are needed to allow for using hysteresis.

The "aggression parameter" α controls the fraction of the heap that may be compacted during a single pass.

The control algorithm primarily controls T, the time to the next compaction event, and operates as a simple state machine. In the waiting state, the algorithm wakes up every 500ms and checks if the current fragmentation is $> F_{ub}$. If it is, the algorithm switches to the compacting state, where it begins running partial compaction passes, each being limited by α. When a partial compaction pass completes, the algorithm measures how long it took, $T_{compact}$ and controls overhead according to O_{ub} by going to sleep for $T = T_{compact}/O_{ub}$. When the algorithm either reaches a fragmentation $< F_{lb}$ or runs out of compaction opportunities, it returns to the waiting state to efficiently observe the system. A subtle point is that it may not always be possible to achieve the fragmentation bounds, and in this circumstance, we will bounce between waiting and compacting quite often, but always stay within O_{ub}.

5 Evaluation

We now evaluate the efficacy of Alaska on a battery of 49 benchmarks from popular suites spanning multiple domains (Embench [17], GAPBS [15], NAS 3.0 [14], SPEC2017). We also test two in-memory databases, Redis and memcached, using the YCSB workload generator [19]. These benchmarks are entirely unmodified, together constituting nearly 3 million lines of code. With this evaluation, we seek to answer the following questions:

Q1 Does the system truly deliver automatic transparent handles? How much programmer effort is involved?
Q2 What effects do Alaska's translations and tracking have on code size?
Q3 What is the software engineering effort required to build the Alaska system?
Q4 What is the performance overhead of software handles?
Q5 How effective is Alaska at enabling Anchorage to reduce fragmentation in a real-world application?
Q6 What is the effect of Anchorage's stop-the-world pause times in a multithreaded application?

5.1 Experimental Setup

Our evaluation was performed on a Dell R6515 with a single AMD EPYC 7443P running Ubuntu 22.04.1 LTS. Each processor has 24 cores with 2-way SMT, 64 entry store buffer, 768 KiB L1D$, 12MiB L2$, and 128MiB L3$ all with 64B line size backed by 512 GiB of DDR4 RAM at 3200 MT/s. Our implementation of Alaska is built atop LLVM 15.0.1. All performance results are gathered from 10 executions per configuration, and speedup/overhead metrics are relative to the median of baseline execution. Each compilation is performed by first applying the `-O3` optimization level with OpenMP disabled. Alaska transformations are then applied to the LLVM bitcode, followed by inlining and several optimization passes. Those passes are similarly applied to the baseline bitcode to ensure a fair playing field.

5.2 Alaska Achieves Programmer Transparency

To evaluate Alaska's ability to provide fully automatic, transparent handle-based memory management, we recorded how many edge cases we had to handle in developing the system. Throughout development, we constantly tested Alaska against both Redis and all of the aforementioned benchmarks. Outside of the issues solved by escape handling and `musl` substitutions (§4.1.4), ***no code in any benchmark or application was modified to support handles.*** Making these benchmarks handle-aware was entirely achieved outside of the application code, by the Alaska compiler and runtime.

Unfortunately, some benchmarks–namely perlbench and gcc from SPEC –violate the strict aliasing assumptions (§3.2) of Alaska . This is not a fundamental limitation of the handle approach, but rather a limitation of our implementation's hoisting technique in the compiler, which relies on the LLVM memory model. Both of these benchmarks *function correctly* if hoisting is disabled in the compiler, when `-fno-strict-aliasing` is provided.[8]

Given that no benchmark or application was modified to support Alaska, we argue that the answer to **Q1** is yes. Most applications, *including Redis*, can be transformed to use automatic transparent handles with a single command:

```
make CC=alaska CXX=alaska++
```

Q2 concerns itself with compilation overhead. Alaska's compiler does not significantly increase compilation time. Redis takes an additional 12 seconds to compile with Alaska (up 21% from 57 seconds, single threaded), however this is due to a relatively unoptimized compiler implementation. Most of this extra time comes from the requirement to compile the whole program to correctly handle calling library code. Performing Alaska's transformations per translation unit, as done in ThinLTO [29], could reduce this overhead drastically.

Executable files grew only about 48% (geomean) via the Alaska transformations and runtime system. The worst case is a doubling of executable file size, which only occurs for very small programs such as those in GAPBS where the binary sizes are already quite small–most of these codes are less than 200 lines long.

5.3 Building and Extending Alaska is Practical

We inspect the code size and development time to answer **Q3**. Alaska is not a particularly large codebase, and was built over a period of eight months. Its core runtime is only 1316 lines of C++, which manages structures like the handle table, tracking, and signaling for barriers. The largest effort was the compiler, which constitutes 2393 lines of C++. The Anchorage service is a mere 567 lines of C++, *half the size of*

[8] That is, if we translate handles before each and every load/store.

the bespoke defragmentation system in Redis [5]. Anchorage is general purpose and does not leak into application code, unlike activedefrag. We take away that, while Alaska's core may have been complex to design and implement, extending it with additional services is trivial, and we have already begun experimenting with future research directions.

5.4 Overhead Varies, but is Overall Low

To answer **Q4**, we evaluated Alaska's runtime overhead on benchmarks from several domains. Figure 7 compares the measured performance overhead (percent increase of wall-clock time) of the benchmarks running Alaska without a service (using malloc to allocate backing memory) against a baseline described in §5.1. The Perlbench and GCC benchmarks from SPEC2017 both violate the assumptions listed in §3.2, and have been compiled with hoisting disabled. The main takeaway is that Alaska's performance impact varies depending on the benchmark, but is overall relatively low, imparting a geomean of 10% overhead if perlbench and GCC are included, and an 8% geomean if not.

Despite the outliers, many benchmarks have *near zero overhead*. These benchmarks benefit substantially from the hoisting capability of Alaska's compiler, with many benchmarks having translations hoisted to their outermost loops.

For example, `619.lbm_s` from SPEC features a large grid allocation which is accessed inside a series of inner loops. The translations in this program are all successfully hoisted to their outermost possible loop level, and as a result the cost of Alaska is amortized to a high degree. Similar amortization can be seen in the NAS benchmarks and `xz` from SPEC2017, which feature similar structure in performance sensitive parts of the application.

Unfortunately, not all programs feature this friendly structure, and some programs do not benefit from any of Alaska's optimizations. These benchmarks stress Alaska's ability to hoist and efficiently translate pointer-chasing data structures such as linked lists and trees for which the compiler is unable to amortize the cost of translation. Additionally, several benchmarks in the Embench suite suffer from poor software design patterns, which block any hoisting from occurring. One such is in `sglib`, which passes all parameters to the core benchmark function through *global variables* instead of as arguments. In LLVM, this results in an additional load from the global variable every time it is used, and hoisting the translation across these may break correctness in concurrent applications.

Handle translation overhead is also seen in benchmarks which perform very little work per translation, such as `sglib`, `mcf` and `xalancbmk`. For example, the `mcf` benchmark from SPEC spends roughly 40% of runtime sorting an array of pointers, which results in 4 translations per comparison. Similarly, `xalancbmk` is written using C++ virtual methods, which block almost all optimizations that would optimize across call sites, and the `this` pointer is translated in almost every function – even when it is not required.

Interestingly, we see a *speedup* of 11% in NAS ep. In this benchmark, we see an increase in L1 instruction cache accesses and misses. The most probable cause of this is the differences in code layout by the LLVM backend that we observed. The effect of this is exacerbated by how small the application is, with the hot loop being only 256 bytes.

To investigate the source of the overheads in these benchmarks, we ran an ablation study on the SPEC2017 benchmarks where we remove features of Alaska and evaluate the resulting overhead. The performance overhead results of this study are shown in Figure 8. The metric marked "alaska" is the overhead seen in Figure 7 with both hoisting and tracking enabled. When the compiler's hoisting optimization is disabled, seen in the "nohoisting" metric, most benchmarks see their overhead nearly double. Those benchmarks that do not see a large increase–such as xalancbmk–do not have many opportunities to hoist, as mentioned earlier.

When the tracking system is removed, marked "notracking", several benchmarks see a large reduction in overhead. This is most apparent in `nab` and `xz`, and we attribute this to the experimental nature of the LLVM StackMap system, which has seen very little use outside of JIT compilers. This overhead mostly comes from the insertion of poll points (§4.1.3), which in the common case should incur no additional overhead–the polls are simple `NOP` instructions. However, it is possible that either the addition of these instructions causes undue stress to the instruction cache or, more likely, this experimental system leads to unexpected bugs in LLVM's backend for some application structures.

5.5 Anchorage Defragments without Black Magic

To evaluate the capabilities of the Anchorage service built on Alaska, we tested it using a large, unmodified application: Redis [6]. Redis is an extremely popular key-value in-memory database designed to be accessed over the network. As we described in the introduction, Redis has long suffered from fragmentation, particularly due to its tendency to be used as an LRU cache atop of other databases in large deployments, resulting in a heap which has allocations scattered everywhere. As a result, Redis includes activedefrag, a bespoke defragmentation system requiring *manual modifications throughout the application's code*, which is also considered "black magic"[9] and cannot be reused in other applications.

Response latency We begin by considering the impact on response latency. We drive Redis with the YCSB workload generator using two workloads [19]. We found that Anchorage on top of Alaska imparts an average of 13% overhead

[9] A term used by its designer:
https://twitter.com/antirez/status/1052590584102305792

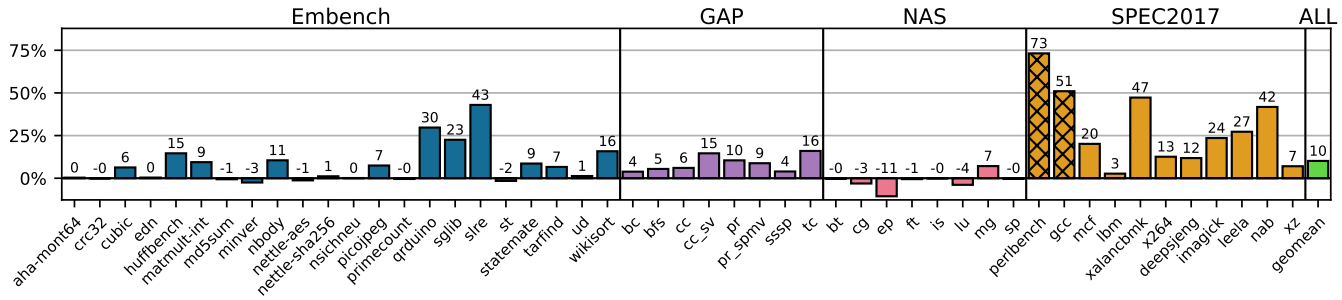

Figure 7. Alaska's performance overhead from translation and pin tracking is relatively low with several outliers.

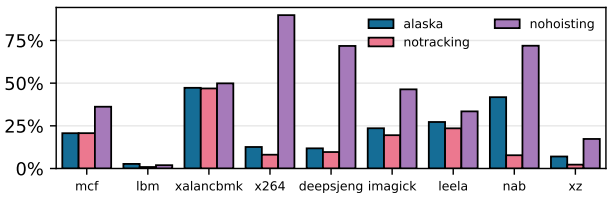

Figure 8. The hoisting optimization drastically reduces the overhead of Alaska handles, and careful design and engineering effort of the tracking system imposes negligible overhead over the cost of using handles with the exception of a few applications (nab, xz)

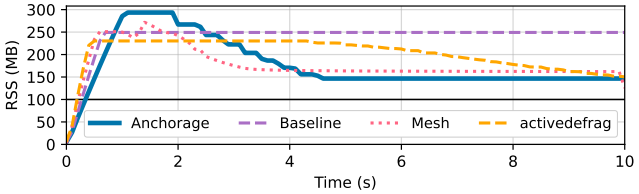

Figure 9. Anchorage can defragment memory under redis at least as well as the bespoke redis defrag implementation.

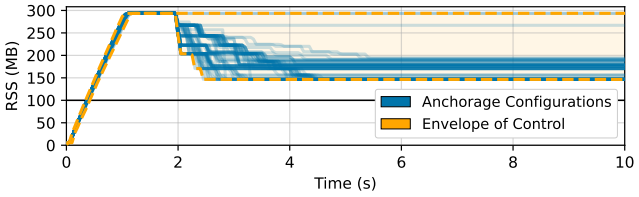

Figure 10. Anchorage gives the Redis users control over the tradeoff between overhead and fragmentation (and memory footprint).

on read latencies (Workload A), and an average 17% overhead on update/write latencies (Workload F). Aside from the overhead of Alaska discussed in §5.4, the lower throughput of the Anchorage allocator imparts an additional overhead relative to glibc malloc used by the baseline. Of note, the allocator overhead is not intrinsic, and could be rectified through improvements and optimizations in the allocator design.

Defragmentation To answer **Q5**, we used the same workload from Mesh [36] which configures Redis to limit memory usage to 100 MiB, then inserts more than that into the database using a workload generator. Redis then evicts keys from its dataset using an LRU policy until its memory usage falls below the 100 MiB threshold. Unfortunately, this creates significant fragmentation, as there are holes throughout the heap that are not filled – and the application's RSS does not decrease in the baseline allocator. Configuring Redis in this way is very common when it is used as a cache for other datasets. This benchmark runs for a total of 10 seconds, and includes results for the baseline allocator, Anchorage, `activedefrag`, and Mesh.

Figure 9 shows the results. Anchorage effectively reduces memory usage from almost 300MiB to 150MiB (40% less compared to baseline Redis). The main takeaway is that Anchorage, which is readily applicable to any codebase without any code modifications, is able to do as well as `activedefrag`, which required extensive, manual, hand-rolled changes to the codebase. We include the data verbatim from Mesh, which does very well in this test, but note that because it cannot move objects, it is unable to resolve the fundamental fragmentation issue.

Control To evaluate the effect of our control system, we sweep the parameters $[F_{lb}, F_{ub}]$, $[O_{lb}, O_{ub}]$, and α (§4.3). In Figure 10, each solid curve shows Anchorage's behavior with a different parameter set, of which there are hundreds. Each parameter set stays within the overhead bounds ($[O_{lb}, O_{ub}]$). As shown in the figure, the envelope of control (dashed curves) is quite large, meaning that the parameters have significant effect. In a deployment, the user can tune these parameters (dynamically, even) for their desired tradeoff between overhead and fragmentation.

Stressing Anchorage To evaluate Anchorage in the face of workloads with a large amount of memory we adapted the test from Figure 9 to have a maximum memory policy of 50 GiB, instead of the original 100 MiB. This workload inserts

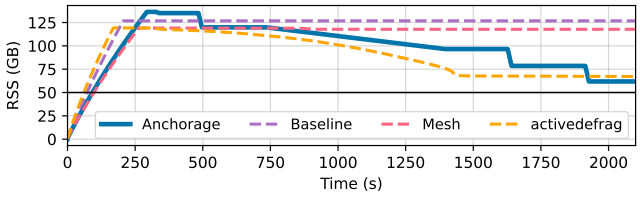

Figure 11. Anchorage successfully defragments workloads with >100GiB RSS.

100 GiB of data, 500 bytes at a time, causing over 2.5x fragmentation[10] when eviction begins – at around 250 seconds. Figure 11 shows the results of this experiment with Anchorage handily defragmenting the heap of this large workload, achieving similar steady-state RSS as `activedefrag`, albeit over a longer time frame.

The longer time taken by Anchorage to defragment the heap is caused by its control algorithm. Around 500 seconds into the test, the control algorithm begins defragmentation. Because the control system operates in units of *percentage of the heap*, the system immediately enters a *7 second pause* as it drastically mispredicts how long the defragmentation will take. The system then backs off for over 250 seconds to maintain the 5% overhead maximum–per its configuration–and begins slowly defragmenting the heap for the rest of the application runtime. It is important to note that this is not a fundamental issue of Anchorage, as the control system can be tuned – as illustrated in Figure 10. As such, careful parameterization of the control algorithm could aid in preventing this behavior. Alternatively, a common approach to hide GC pause times is via concurrent compaction algorithms, the potential of which is briefly discussed in §7.

We also evaluated Mesh in this environment and, in its default configuration, was either not aggressive enough–defragmenting only a few MiB at a time as can be seen around 750 seconds–or does not scale to workloads this large. Note, we had to modify Mesh's source code to allow heap sizes larger than 64GiB.

5.6 Anchorage's Stop-the-World Pauses

To answer **Q6**, we evaluated Anchorage's effect on request latencies on an alternative in-memory key-value database, `memcached` [3], which can be configured to run in parallel with multiple threads. We devised a synthetic test where ~1 MiB of memory is relocated at each pause, regardless of the fragmentation ratio, resulting in average pause times less than 2ms. `memcached` is driven by the YCSB test suite–specifically workload A, which has been scaled up to run for longer–which provides the latencies shown. Because this workload is driven through the loopback network, it does see considerable noise. The results of this test are shown in

[10]This is more than the expected 2x, as Redis' internal tree data structures provide some overhead.

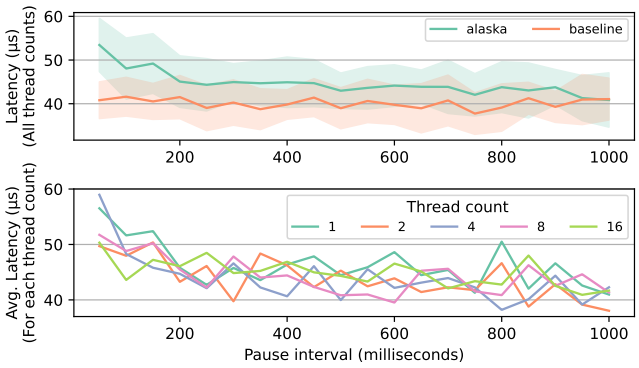

Figure 12. Latencies in `memcached` vary based on the pause time, and there is no trend between latency and number of threads.

Figure 12, gathered by varying the number of threads and the interval at which Anchorage performs a pause.

The upper plot in Figure 12 shows the overall effect on latency for all thread configurations combined, as well as the standard deviation, indicating that Alaska incurs an average of 10% overhead in this application across all configurations (including impractical 50ms pause intervals). This manifests as an average latency increase of $4\mu s$. With more practical intervals (above 500ms), the latency increase drops to less than 7%. The lower plot breaks the latencies down by thread count. The main takeaway is that, at low intervals, there is an expected effect on average latency driven primarily by outliers when requests are blocked by pauses. Importantly, Anchorage never naturally pauses at these low intervals in any other applications. Additionally, we measured no correlation between number of threads and pause time.

6 Related Work

Defragmenting Unmanaged Languages has seen resurgence in recent years. Mesh [36], the most pertinent example, works on existing binaries without recompilation. Mesh changes malloc so that heap objects are carefully positioned on virtual pages. A cooperating kernel takes advantage of this by mapping several virtual pages to the same physical one without overlapping the heap objects, reducing fragmentation and memory usage at the physical level. In contrast, Alaska enables heap object mobility in the virtual address space, a more general capability that is both necessary to achieve fragmentation limits [37], and can enable other services.

Compiler-managed Address Translation CARAT [40, 41] aims to replace paging via per-object RWX protections and object-level movement in the physical address space. Both functionalities are enabled by compiler/kernel cooperation, and are automatic and transparent to the programmer. Object mobility is based on allocation and pointer escape tracking, combined with patching of all pointers. In other

words, there is no indirection like with handles in Alaska. This avoids the handle pin and handle translation costs, but makes object movement more expensive because the update is not $O(1)$ as in handles, but at least $O(m)$, where m is the number of pointers to the object. While Alaska does not currently enable protections, it could do so in the future.

Programming Models with Indirection Interestingly, the historic manual indirection strategies described in §2 have returned. AIFM [38] features a programming model on top of C++ to allow the developer to take advantage of far memory data structures. While the system works well in enabling far memory, it requires the programmer to do error-prone heavy lifting, insert scopes and pinning logic similar to classic handle-based systems. ActivePointers and others [21, 23, 39] allow the runtime to relocate objects and invalidate references (a la paging) via a modification of C++'s "smart pointers". These approaches demand rewriting parts of the application or using libraries which cannot benefit from translation-aware optimizations in the compiler. Such systems could perhaps be easily built atop Alaska's service interface.

7 Discussion

Though we have only discussed handles in the context of fragmentation, we note that handles provide the more fundamental capability of managing object motion at the object granularity. Handles in the past have also given the ability to discard of swap memory at the object granularity, *rather than pages*. Managing memory at the object granularity has gained interest in several contexts, such as application-level remote memory systems [38], replacements for paging [40, 41], and security via capability-based addressing [44, 45]. It has also been shown that object mobility can be used to dynamically enhance cache locality [18, 20, 22, 42, 43]. Similarly, work has been done to place rarely used of objects in cheaper nonvolatile memory to optimize memory usage [46].

We posit that handles provide a powerful vehicle to implement such ideas with simple modifications to the compiler and runtime. The Alaska system has been designed to be extensible beyond what we describe in this paper with these capabilities in mind. Alaska can be configured with "handle faults", which very closely approximates the capabilities of a system that uses page faults. While this paper does not evaluate this feature in detail, initial investigation indicates that this check has minimal additional overhead (~1-2%), but enables advanced techniques in the runtime system. With this check enabled, objects can be swapped out of memory in the same way a kernel might do so for pages.

Further, this "swapping" mechanism could be utilized to *speculatively* move memory without stopping the world for the duration of movement. The runtime would occasionally mark entries in the handle table as "invalid" and speculatively copy their data to an alternative location. If another thread accesses that handle, it would trap to the runtime and atomically mark the object as "valid". The runtime would then attempt to CAS (compare-and-swap) the entry in the handle table, marking it as "valid" with a new address. If the CAS succeeds, the old memory can be freed, and the object has been relocated. If it fails, the relocation is aborted, and the speculative copy is freed, as some other thread has pinned that handle while the copy was being made. We see it as an interesting path forward to implement concurrent memory movement, as this closely resembles the concurrent compaction system seen in the Shenandoah GC [26]. This simple mechanism could be utilized to implement swapping objects to disk, compression, or even far memory.

8 Conclusion

We have described the design and implementation of Alaska, a prototype compiler and runtime system that automates the use of handle-based memory management while being entirely transparent to the programmer. Alaska is a platform that enables runtime features and services that rely on object mobility. Using this platform, we designed and implemented Anchorage, a compacting memory allocator which reduces memory usage in highly fragmented applications. In the future, we plan on using Alaska's extensibility as a vessel to enable additional transparent runtime services such as memory disaggregation, locality enhancement, and capability-based security, as well as as a light-weight alternative to paging.

References

[1] The GNU C library https://www.gnu.org/software/libc/libc.html.
[2] The libunwind project https://www.nongnu.org/libunwind/.
[3] memcached https://memcached.org/.
[4] MUSL libc https://musl.libc.org.
[5] Redis active memory defragmentation https://github.com/redis/redis/pull/3720.
[6] Redis https://redis.io/.
[7] Spec cpu 2017 - gcc_s description https://www.spec.org/cpu2017/Docs/benchmarks/602.gcc.html.
[8] Spec cpu 2017 - perlbench_s description https://www.spec.org/cpu2017/Docs/benchmarks/600.perlbench_s.html.
[9] Stack maps and patch points in LLVM — LLVM documentation https://llvm.org/docs/StackMaps.html.
[10] Tiobe index https://www.tiobe.com/tiobe-index/, Jun 2022.
[11] Ole Agesen. Gc points in a threaded environment. 1998.
[12] Ole Agesen, David Detlefs, and J. Eliot Moss. Garbage collection and local variable type-precision and liveness in java virtual machines. page 269–279, 1998.
[13] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.
[14] David Bailey, Tim Harris, William Saphir, Rob Van Der Wijngaart, Alex Woo, and Maurice Yarrow. The nas parallel benchmarks 2.0. Technical report, Technical Report NAS-95-020, NASA Ames Research Center, 1995.
[15] Scott Beamer, Krste Asanović, and David Patterson. The gap benchmark suite. *arXiv preprint arXiv:1508.03619*, 2015.

[16] Michael A Bender, Alex Conway, Martín Farach-Colton, William Kuszmaul, and Guido Tagliavini. Tiny pointers. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 477–508. SIAM, 2023.

[17] J Bennett, P Dabbelt, C Garlati, GS Madhusudan, T Mudge, and D Patterson. Embench: An evolving benchmark suite for embedded iot computers from an academic-industrial cooperative, 2022.

[18] Trishul M. Chilimbi, Mark D. Hill, and James R. Larus. Cache-conscious structure layout. *SIGPLAN Not.*, 34(5):1–12, may 1999.

[19] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, page 143–154, New York, NY, USA, 2010. Association for Computing Machinery.

[20] Robert Courts. Improving locality of reference in a garbage-collecting memory management system. *Commun. ACM*, 31(9):1128–1138, sep 1988.

[21] David Detlefs. Garbage collection and run-time typing as a c++ library. In *C++ Conference*, 1992.

[22] Chen Ding and Ken Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at run time. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, page 229–241, New York, NY, USA, 1999. Association for Computing Machinery.

[23] Daniel R. Edelson. Precompiling c++ for garbage collection. In *Proceedings of the International Workshop on Memory Management*, IWMM '92, page 299–314, Berlin, Heidelberg, 1992. Springer-Verlag.

[24] Jason Evans. Tick tock, malloc needs a clock. In *Applicative 2015*, Applicative 2015, New York, NY, USA, 2015. Association for Computing Machinery.

[25] Robert R. Fenichel and Jerome C. Yochelson. A lisp garbage-collector for virtual-memory computer systems. *Commun. ACM*, 12(11):611–612, nov 1969.

[26] Christine H. Flood, Roman Kennke, Andrew Dinn, Andrew Haley, and Roland Westrelin. Shenandoah: An open-source concurrent compacting garbage collector for openjdk. 2016.

[27] Apple Computer Inc, editor. *Inside Macintosh. Vol. 2*, volume 2. Addison-Wesley, 14. printing edition.

[28] Douglas Johnson. The case for a read barrier. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 279–287, New York, NY, USA, 1991. Association for Computing Machinery.

[29] Teresa Johnson, Mehdi Amini, and Xinliang David Li, editors. *ThinLTO: Scalable and incremental LTO*, 2017.

[30] H.B.M. Jonkers. A fast garbage compaction algorithm. *Information Processing Letters*, 9(1):26–30, 1979.

[31] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*., pages 75–86. IEEE, 2004.

[32] Chris Lattner and Vikram Adve. Automatic pool allocation: Improving performance by controlling data structure layout in the heap. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, page 129–142, New York, NY, USA, 2005. Association for Computing Machinery.

[33] Angelo Matni, Enrico Armenio Deiana, Yian Su, Lukas Gross, Souradip Ghosh, Sotiris Apostolakis, Ziyang Xu, Zujun Tan, Ishita Chaturvedi, David I. August, and Simone Campanoni. NOELLE Offers Empowering LLvm Extensions. In *International Symposium on Code Generation and Optimization, 2022. CGO 2022.*, 2022.

[34] Pekka P. Pirinen. Barrier techniques for incremental tracing. *SIGPLAN Not.*, 34(3):20–25, oct 1998.

[35] Filip Pizlo, Lukasz Ziarek, Petr Maj, Antony L. Hosking, Ethan Blanton, and Jan Vitek. Schism: Fragmentation-tolerant real-time garbage collection. *SIGPLAN Not.*, 45(6):146–159, jun 2010.

[36] Bobby Powers, David Tench, Emery D. Berger, and Andrew McGregor. Mesh: Compacting memory management for c/c++ applications. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 333–346, New York, NY, USA, 2019. Association for Computing Machinery.

[37] J. M. Robson. Worst case fragmentation of first fit and best fit storage allocation strategies. *The Computer Journal*, 20(3):242–244, 01 1977.

[38] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-performance, application-integrated far memory. In *Proceedings of the 14^{th} USENIX Symposium on Operating Systems Design and Implementation*, OSDI '20, pages 315–332, Berkeley, CA, USA, November 2020. USENIX Association.

[39] Sagi Shahar, Shai Bergman, and Mark Silberstein. Activepointers: A case for software address translation on gpus. *SIGARCH Comput. Archit. News*, 44(3):596–608, jun 2016.

[40] Brian Suchy, Simone Campanoni, Nikos Hardavellas, and Peter Dinda. Carat: A case for virtual memory through compiler- and runtime-based address translation. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, page 329–345, New York, NY, USA, 2020. Association for Computing Machinery.

[41] Brian Suchy, Souradip Ghosh, Drew Kersnar, Siyuan Chai, Zhen Huang, Aaron Nelson, Michael Cuevas, Alex Bernat, Gaurav Chaudhary, Nikos Hardavellas, Simone Campanoni, and Peter Dinda. Carat cake: Replacing paging via compiler/kernel cooperation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '22, page 98–114, New York, NY, USA, 2022. Association for Computing Machinery.

[42] Harmen L. A. van der Spek, C. W. Mattias Holm, and Harry A. G. Wijshoff. *Automatic Restructuring of Linked Data Structures*, volume 5898 of *Lecture Notes in Computer Science*, page 263–277. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[43] Zhenjiang Wang, Chenggang Wu, Pen-Chung Yew, Jianjun Li, and Di Xu. On-the-fly structure splitting for heap objects. *ACM Transactions on Architecture and Code Optimization*, 8(4):1–20, Jan 2012.

[44] Robert N.M. Watson, Jonathan Woodruff, Peter G. Neumann, Simon W. Moore, Jonathan Anderson, David Chisnall, Nirav Dave, Brooks Davis, Khilan Gudka, Ben Laurie, Steven J. Murdoch, Robert Norton, Michael Roe, Stacey Son, and Munraj Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[45] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. The cheri capability model: Revisiting risc in an age of risk. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 457–468. IEEE Press, 2014.

[46] Zhen Xie, Jie Liu, Jiajia Li, and Dong Li. Merchandiser: Data placement on heterogeneous memory for task-parallel hpc applications with load-balance awareness. page 204–217, 2023.

[47] Benjamin Zorn. Barrier methods for garbage collection. 11 1990.