User Extensible Heap Manager for Heterogeneous Memory Platforms and Mixed Memory Policies

Christopher Cantalupo
Vishwanath Venkatesan
Jeff R. Hammond
Krzysztof Czuryło
Simon Hammond

March 18, 2015

Abstract

Memory management software requires additional sophistication for the array of new hardware technologies coming to market: on package addressable memory, stacked DRAM, non-volatile high capacity DIMMs, and low-latency on-package fabric. As a complement to these hardware improvements there are many policy features that can be applied to virtual memory within the framework of the Linux* system calls `mmap(2)`, `mbind(2)`, `madvise(2)`, `mprotect(2)`, and `mlock(2)`. These policy features can support a wide range of future hardware capabilities including bandwidth control, latency control, inter-process sharing, inter-node sharing, accelerator sharing, persistence, checkpointing, and encryption. The combinatorial range implied by a platform with heterogeneous memory hardware, and many options for operating system policies applied to that hardware is enormous, so it is intractable to have a separate custom allocator addressing each of them. Each layer of the application software stack may have a variety of different requirements for memory properties. Some of those properties will be shared between clients, and some will be unique to the client. We propose software that will enable fine-grained client control over memory properties through our User Extensible Heap Manager, which efficiently reuses memory modified by expensive system calls and remains effective in a highly threaded environment.

1 Introduction

The Linux operating system offers several system calls to enable user level modification of memory policies associated with virtual address ranges. These system calls will be the principle user level mechanism to support the new features available in future hardware. An important role of heap management software is to avoid system calls through reuse of virtual memory mapped from the operating system. The goal of the *memkind* library is to bring the control available through system calls that enforce memory policies to the interfaces used for heap management without sacrificing the performance that is available from other user level heap managers.

The POSIX* standard `mmap(2)` and `munmap(2)` system calls can be used to allocate and deallocate virtual memory. However, accessing the kernel from user space through a system call is expensive and reduces application performance if done too frequently. The finest granularity of allocation enabled through these is the page size, calling them acquires a global lock on the kernel memory subsystem, and the `munmap(2)` call requires knowledge of the extent of memory to be unmapped. For all of these reasons, `mmap(2)` is not generally the mechanism used to acquire virtual memory within a C program in the POSIX environment. Instead the ISO* C `malloc(3)` and `free(3)` family of APIs are used. An implementation of these interfaces is defined in the *libc* library, and many applications use the implementation offered by their compiler. In some cases there is a need for a specialized allocator, and there are many examples of `malloc(3)` implementations in use.

Given all of the custom allocators available, we must motivate the need for yet another heap manager. The Linux operating system offers even more system calls for memory control than are defined in the POSIX standard. The user is generally forced to use the system calls directly, rather than a heap manager, when precise control of memory properties is required. Some examples of common situations where *glibc*'s `malloc(3)` is insufficient are, explicit use of the Linux huge page functionality, explicit binding of memory to particular NUMA nodes on a system, and file backed memory. Custom allocators are most commonly used to achieve better allocation time performance for a particular application's usage pattern rather than to enable particular hardware features.

A number of hardware features challenge a homogeneous memory model. Several of these features are not at all new: the page size extension (PSE) and cc-NUMA support have been enabled in hardware for over ten years. Some features are currently available, but not extensively used: gigabyte pages in x86_64 and stacked DRAM. In the near future the integration of addressable memory and a low latency network interface controller (NIC) into the processor package and the availability of non-volatile high capacity DIMMs will add yet more hardware features to complicate memory management.

There are a wealth of established heap management solutions available. Rather than starting from scratch, we build on top of the implementation that best suits our needs. Our target users are in the high performance computing (HPC) community who require efficient support of highly threaded environments. We require a solution that offers some measure of encapsulation to enable the isolation of memory resources with different properties. Licensing is a consideration: a BSD or MIT style license is most acceptable to the HPC community. The opportunity to participate in an active open source development community is quite valuable. The *jemalloc* library fits these requirements very well.

The ISO C programming language standard provides a user-level interface employed for memory management in many applications, either implicitly or explicitly. These APIs are the well known set: `malloc(3)`, `calloc(3)`, `realloc(3)`, `free(3)`, and `posix_memalign(3)` (`posix_memalign(3)` is a POSIX extension to the ISO C standard). The *memkind* library co-opts these APIs while prepending `memkind_` to the names, and extending the interface with an additional argument: the "kind" of memory. The details of what is represented in the structure of this additional argument will be discussed later, but the interface enables a plug-in architecture that can be extended as hardware and policy features evolve. It is important to note that the "kind" of memory determines both hardware selection and the application of policies to allocated memory.

The *memkind* library is built upon *jemalloc* – a general purpose malloc implementation. The *jemalloc* and *memkind* libraries are open source and are available from the memkind organization at github: `https://github.com/memkind`. Both *memkind* and *jemalloc* are distributed under the two clause BSD license described in the `COPYING` file. In the future, these libraries may be bundled as an Intel® product called the "User Extensible Heap Manager."

2 HPC Middleware

In HPC, middleware facilitates application development by abstracting portability and performance optimizations from the application developer. Popularly used middleware are MPI implementations like MPICH [15], MVAPICH and Open MPI [13], portable data format libraries like NetCDF [23] and HDF5 [12], computational libraries like BLAS [18], Intel® Math Kernel Library and numerical solvers like PETSc [2], Trilinos [16] and Hypre. Most of these libraries have special memory usage requirements and some even define their own custom allocator to handle memory optimizations. Furthermore, some middleware intercept calls to malloc/free to customize them. This can potentially cause conflicts in the software stack if more than one library or the application itself attempt this method of customization. The *memkind* library introduced here can easily address these issues by providing a uniform set of interfaces that enable efficient use of the underlying memory resources without major modifications to the middleware layers or the application.

Usage models for a feature-rich memory manager exist as a result of (1) physical memory type, (2) virtual memory policy, and (3) virtual memory consumers (clients). Examples of (1) include on-package memory and nonvolatile memory, which are now or will soon be integrated into systems in addition to the standard DRAM technology (*i.e.* off-package memory). Page protection, size, and pinning/migration are all examples of (2). Libraries and runtime systems fall into (3); obvious examples include MPI and OpenSHMEM, both of which have at least one API call that allocates memory. In this section we will discuss some important clients that we think can benefit and hope will make use of this library.

2.1 Mathematical Frameworks

Traditionally, library-based solver frameworks such as Trilinos [16] and PETSc [2] have provided memory allocation routines to provide guarantees to internal uses of the data for issues such as operand alignment and interactions with distributed message passing such as MPI. Recently, the addition of Kokkos arrays [9] to the Trilinos framework has provided compile-time transformation of data layouts to support cross-platform code development on machines as diverse as general-purpose GPUs, many-core-architectures such as the Intel® Xeon Phi™ coprocessor and contemporary multi-core processors such as the Intel® Xeon® processor family and IBM's POWER* lines. The addition of compile time data structure usage hints – through C++ meta-template programming – to Kokkos Views allows for the insertion of *memkind* routines to explicitly allocate and manage individual data structures (Views) as needed. This approach therefore abstracts away almost all of the specific details relating to NUMA-awareness and multiple memory types for library and application developers while maintaining the performance and portability desired.

2.2 MPI

MPI is a ubiquitous runtime environment for high-performance computing that presents a number of examples where a flexible memory allocator is useful.

First, MPI provides its own allocation and deallocation calls, `MPI_Alloc_mem()` and `MPI_Free_mem()`, which are sometimes nothing more than wrappers around `malloc(3)` and `free(3)`, but can allocate inter-process shared memory or memory backed by pages that are registered with the NIC (and thus pinned); in either case, the memory may come from a pre-existing pool or be a new allocation. A useful feature of `MPI_Alloc_mem()` is the `MPI_Info` argument, which allows a user to provide arbitrary information about the desired behavior of the call with key-value pairs; it is natural to use this feature to enable the user to lower *memkind*-specific features through the industry-standard MPI interface.

Second, inter-process one-sided communication and direct access require special allocators, *e.g.* `MPI_Win_allocate()` and `MPI_Win_allocate_shared()`, both of which can leverage *memkind* to provide symmetric heap memory, network-registered memory or inter-process shared memory. Like `MPI_Alloc_mem()`, these calls take an `MPI_Info` argument, which gives the user extra control over the behavior of their MPI program.

2.3 OpenSHMEM

OpenSHMEM is a one-sided communication API that has some of the features of MPI-3, but warrants special consideration because of the user expectation (albeit not a requirement of the current OpenSHMEM specification) that virtual addresses returned by `shmalloc()` be *symmetric*. That is, that they be identical across all processing elements both within a node and across nodes. Supporting this expectation requires an implementation to reserve a large portion of the address space and suballocate from it.

Welch *et al.* describe an extension of OpenSHMEM to support memory *spaces* [27] that is naturally aligned to the features of *memkind*. While supporting one symmetric heap is straightforward by modifying an existing allocator, supporting an arbitrary number of symmetric heaps across different subsets of processes is more complicated, but the spaces API maps naturally to *memkind*.

2.4 Hybrid OS Kernels

Wisniewski *et al.* [28] describe a hybrid operating system designed for HPC where two coupled but largely independent operating systems are resident on the same platform. One OS is a fully featured Linux implementation and the other is a light weight operating system running on the CPUs designated for compute intensive operations. They give a simple solution for partitioning the hardware address space at boot time, but leave open the question of how virtual addresses will be shared between operating systems. The obvious solution is to handle this distinction through virtual memory policies and *memkind* can be used to track these within the context of a user level heap manager.

2.5 Intel® Math Kernel Library

The Intel® Math Kernel Library offers a wide range of mathematical operations. One set of operations that is particularly interesting in the context of on package memory are the sparse matrix computations which are often memory bandwidth bound. The APIs that the Intel® Math Kernel Library offers for sparse matrix solving include functions that allocate space and copy sparse data provided by the user into layouts that are optimized for access patterns used when computing. This functionality would be well served by the high bandwidth characteristics of on package memory, and the *memkind* library can be used to locate the structures there.

3 Related Work

To date there are many examples of software designed to deal with problems that can be generalized to the condition of having a heterogeneous memory architecture. Some important examples of this in the context of the Linux operating system are system calls and user libraries for enabling cache coherent Non-Uniform Memory Access (cc-NUMA) as well as the Linux huge page functionality. Additionally, there are the PGAS family of programming models which present the user with a global view of memory, either through load-store (in a PGAS language, *e.g.* [22, 7]) or a one-sided API (in a PGAS library, *e.g.* [3, 21]).

Of the new memory hardware features on the horizon, on package high bandwidth memory will be available soonest, and as a result, there is a growing body of literature discussing software modifications for enabling a two tiered memory hierarchy. There have been attempts to address this challenge within the operating system [20], and other discussions of user level software modifications [19]. Here we discuss user-level software which addresses not only the problem introduced by a two-tiered memory hierarchy, but the more general problem of user selection of memory hardware and policies applied to memory through a unified customizable allocation interface.

3.1 OS Abstractions

There have been many attempts to preserve a homogeneous memory model while using memory hardware that is heterogeneous by enabling the operating system with an abstraction layer. Some examples of this are the Linux Transparent Huge Page (THP) feature, or the techniques of [20] to support on-package addressable memory. In these examples a heuristic is executed by a system daemon which tries to opportunistically use a hardware feature when possible or to predict future memory access patterns and shift resources to optimize performance in the case where the prediction were true. Note that in recent Linux kernel versions the primary mode of operation for THP is allocating huge pages at time of fault and the daemon is a secondary mechanism. In this paper we posit that although these techniques have the advantage of preserving existing memory management APIs and they enable some performance benefit without application modification, the proliferation of features available in memory hardware requires a more sophisticated software interface for user-level memory management. Additionally, in the context of high performance computing (HPC) the general trend is to simplify and defeature the operating system to enable the full utilization of resources by the application [24][28] rather than pushing complexity into the operating system.

3.2 User Level Software

Having explicit control of the locality of physical memory backing in a NUMA environment can have a significant impact on application performance. Furthermore, being able to allocate memory from specific NUMA nodes in a system becomes imperative with upcoming new memory technologies. In [17] the authors try to make TCMalloc NUMA-aware. The issue with this solution is, the lack of arenas in TCMalloc stops this approach from providing partitions for each kind of allocation. Furthermore, the approach suggested is not extensible to future memory technologies and only looks at allocating from the nearest NUMA node. There are other solutions which try to solve NUMA awareness with heap managers. Pool allocator [25] implemented as a part of Boost [8] provides an approach to allocate memory from an underlying pool. This method is restricted to applications written in C++ with Boost libraries. Furthermore, the usage is complicated by the requirement to create a pool which ensures that allocations come from a desired NUMA node. Macintosh OS X* uses a "scalable zone allocator" [26] to make allocations. Even this approach requires application developers to define a zone to ensure allocation comes from the appropriate NUMA node and it is a solution specific to Macintosh OS X. The approach suggested in this paper provides not only a NUMA-aware heap manager, but also provides an extensible architecture which supports partitioning of multiple memory *kinds*.

Two prominent Linux user libraries that define allocators which provide enhancements to the `mmap(2)` system call are libnuma and libhugetlbfs. Neither of these user libraries couple the feature they provide of simplifying the system call to map virtual address ranges with a heap management system. The purpose of heap management is to provide a data structure that enables reuse of virtual address ranges already reserved for the application by calls to the operating system once they are no longer in use by the application. In this way the heap is an interface designed to avoid making system calls.

3.3 Existing Heap Managers

There are multiple heap managers designed to improve the performance of allocators in multi-threaded environments. Berger *et al.* [4] discuss the effects of characteristics such as speed, scalability, false sharing avoidance and low fragmentation as the key issues which affect performance of allocators in multi-threaded environments. Apart from [4] there are many other heap managers designed to address these issues. *Jemalloc* [10] is one such heap manager which addresses these problems with the help of multiple allocation arenas. *Jemalloc* uses a round-robin approach to assign arenas to threads. This approach is suggested to be a more reliable approach compared to hashing thread identifiers as used in other allocators like Hoard [4]. Google* TCMalloc [14] is a heap manager designed to address the challenges associated with multi-threaded applications. It was initially used in production environments in Facebook* , but was replaced with *jemalloc* [1, 11] due to its inconsistent memory economy which is cured in *jemalloc* by dirty-page trimming with the help of `madvise(2)`.

In [5] the authors describe a customizable C++ heap management framework, heap-layers, where the purpose of the customization is to improve allocator performance for application specific allocation patterns [5]. A follow up article [6] shows the limited performance benefit of such an effort when compared to a good general purpose solution. In contrast, here we describe a C solution where the purpose of the customization is to exploit inhomogeneities in hardware and express OS policies that apply to the memory being managed. There is also the opportunity for allocator performance optimization through the customization interface described. By building our interface in the C language and based on syntax borrowed from the ISO C standard, we enable use of that interface by all languages and environments that can link to C object code, and provide an easy transition for those application which use the ISO C allocation interfaces.

4 Design and Implementation

The *jemalloc* library is the default heap manager on the FreeBSD* operating system, and is also prominently used by the Firefox* web browser, the Facebook back-end servers, and the Ruby programming language. There are also many other uses of *jemalloc* as it fills an important niche; to quote the *jemalloc* README:

"*jemalloc* is a general purpose `malloc(3)` implementation that emphasizes fragmentation avoidance and scalable concurrency support."

No heap management system is perfect in all use cases, but *jemalloc* is quite good in many, and especially in tackling the problems inherent in a highly threaded environment.

We chose *jemalloc* not only because of its performance characteristics, but also because of the partitioning provided by its arena structure. As of version 3.5.1, the *jemalloc* library creates four arena structures per CPU on the system. Each thread is assigned to an arena in a round-robin fashion and the associated arena index is recorded in thread local storage. When a thread requests a small allocation all locks and buffers used to service the request are local to the associated arena. This algorithm, which is primarily designed to avoid lock contention, also provides a level of encapsulation that ensures that buffers stored in two different arenas are not mixed. In this way we can associate memory properties with an arena without fear that these properties will be polluted. The *jemalloc* library enables the use of user-created arenas through its "non-standard interface." We leverage this capability to create arenas with specific memory properties, and then select an arena with the requested properties when doing an allocation.

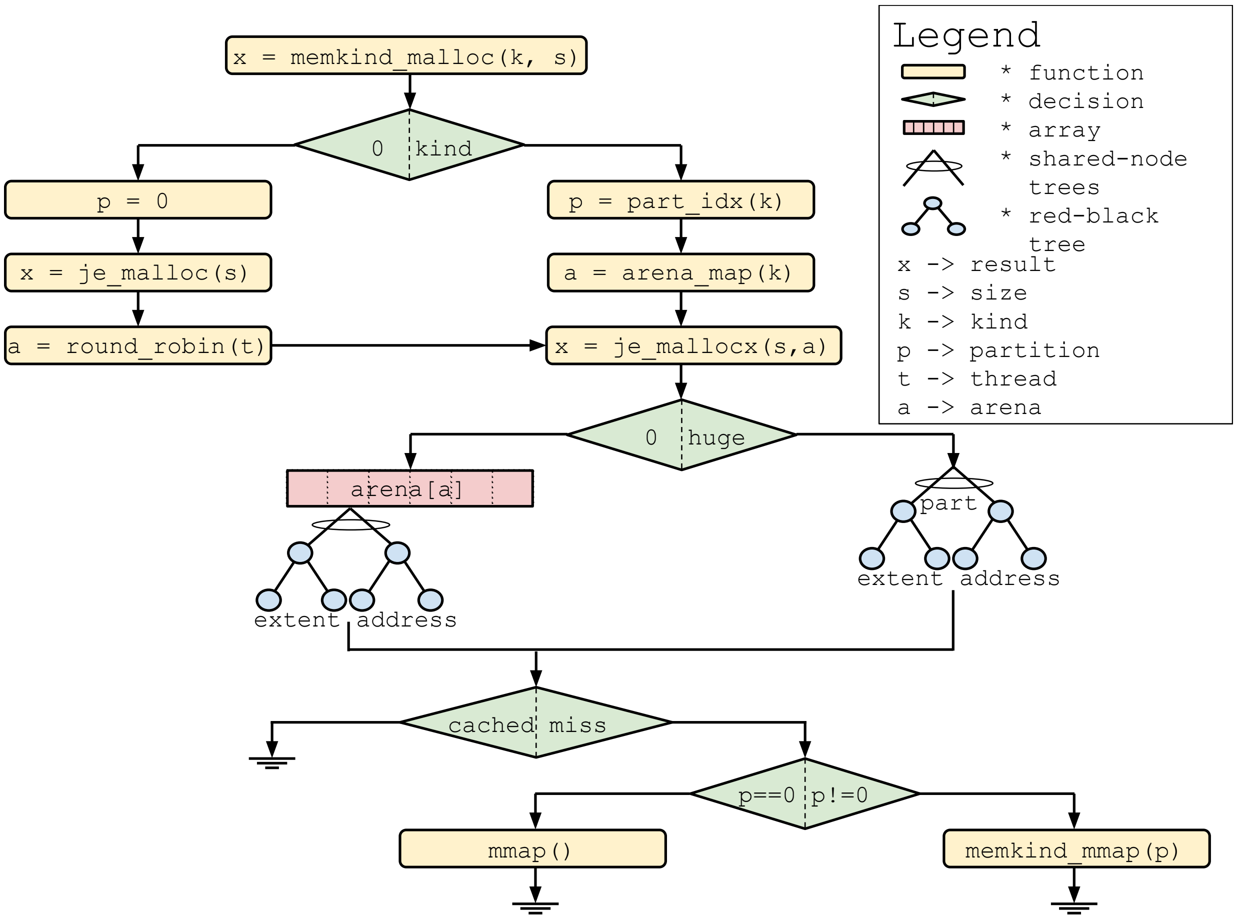

Figure 1: Program flow for `memkind_malloc()`.

4.1 The jemalloc Library Extension Interface

The *jemalloc* non-standard interface defines functions such as `je_mallocx()` which take an additional “flags” argument. The flags enable control of alignment, zeroing and arena selection. In this class of interfaces *jemalloc* also provides a `je_mallctl()` function that can be used for introspection of the library's state and modification of library behavior. This API enables the creation of arenas that are used exclusively when the user selects them through the flags argument of the non-standard interface allocation routines. When a new arena is created `je_mallctl()` returns an arena index which selects from the internal arena array data structure. Through bit manipulation this index is encoded in the “flags” for allocator arena selection.

We have extended the arena creation facility from having a one directional return of an arena index, to a bi-directional exchange of indices between the *jemalloc* library and the *memkind* library. The *memkind* library provides the partition index which is stored in the created arena and can be passed to the `mmap(2)` wrapper function when *jemalloc* maps virtual addresses from the operating system as a result of using a *memkind* created arena for allocations. There is a one-to-one mapping between partition indices and kinds of memory, and by abstracting the kind of memory to an integer value we limit the impact on the *jemalloc* implementation. The current implementation uses a weak function reference to the *memkind* `mmap(2)` wrapper: `memkind_partition_mmap()`. This function reference is only used in the case where an arena is created by *memkind*, and this is enforced by tagging all other arenas with a zero partition index. In the future this may be implemented with a callback registration in *jemalloc* rather than a weak symbol.

4.2 Data Structures in jemalloc

Figure 1 shows the basic control flow of a call to one of the *memkind* allocation interfaces with some of the important data structures in the *jemalloc* library and how the *memkind* library interacts with them. Here zero designates the default option for a decision. Note that `je_malloc()` does not call `je_mallocx()` internally, but it does call a function with a similar call signature. The arenas, the kind and the red-black tree nodes are all tagged with a partition index. The partition index has the highest precedence in the comparison operator for both the extent and address tree ordering. The arena structure has been discussed at length, and the arenas are organized in an indexable array. Each arena has two trees associated with it: the extent tree and address tree. These are red-black trees which are a self balancing ordered binary trees where each node describes a freed virtual address range. These trees share nodes and each node references edges for traversing either tree. The nodes store information about virtual address ranges that have been mapped from the operating system and are available for servicing allocation requests. To select the best node to service an allocation request the edges for the tree ordered by extent are used. To check if a freed address can be coalesced with an existing extent, the edges for the tree ordered by address are used.

The arenas provide a partitioning of memory properties for small allocations, but additional modifications are required for larger allocations. The version of *jemalloc* that was forked to support *memkind*, version 3.5.1, supports “huge” allocations (bigger than two megabytes) with an extent/address tree that is shared by all threads and this data structure is not bound to an arena. The *memkind* extension to *jemalloc* partitions this tree by tagging each node with a partition index which is used by the insertion, deletion and query operators of the tree as the principle comparison operation. Additionally the coalescing algorithm of *jemalloc* is modified so that virtual address ranges tagged with different partition indices are not coalesced.

4.3 The Plug-in Architecture

The *memkind* library provides its plug-in architecture by allowing the user to modify each of the parameters for the enabled memory system calls through implementing a function to generate them. The fundamental data structure of the *memkind* library is a `struct` of the same name with the first element pointing to a constant vtable of function pointers called the `memkind_ops` structure. The functions in this structure are used to determine each of the parameters to the `mmap(2)` and `mbind(2)` system calls. This feature could be extended to include other memory system call parameters. This vtable provides the polymorphism required to modify the callback made by the heap manager to map virtual address ranges. This mechanism provides a temporal coupling of all system calls for modifying memory properties. Applications will only be forced to call into the kernel to modify memory properties upon the exhaustion of the free memory pool associated with that set of properties.

In cases where the partitioned *jemalloc* heap algorithm is insufficient the user can opt to implement their own completely independent allocator by defining functions that mimic the ISO C allocation APIs, as these functions (e.g. `malloc(3)` and `free(3)`) are also captured by the `memkind_ops` vtable. The only additional requirement of allocation routines which do not use the partitioned *jemalloc* implementation is that they must define a function that will determine if the associated `free()` implementation is capable of deallocating a given virtual address. This enables the freeing of a pointer in a context where the provenance of the pointer is unknown.

4.4 Static and Dynamic Kinds

The *memkind* library defines interfaces for "static kinds" which are available without requiring the user to define them. These are intended to be representative of requirements shared between clients. Some examples of these are the `MEMKIND_HBW` kind which targets high bandwidth memory and the `MEMKIND_HUGETLB` kind which targets the default Linux *hugetlbfs*. If a client has a unique set of requirements, or their requirements were not integrated as static kinds, they have the option to define their own "dynamic kinds" through the `memkind_create_kind()` interface. This interface takes as input a constant vtable of the operations that define the kind. By providing an interface that makes it easy for the client to define the combination of memory hardware and policy required, the library need not define every possible combination in its internal static kinds.

4.5 The Decorator Interface

By unifying the interface for accessing different allocation techniques and policies we provide the ability to apply modifications to all allocations through a high level decorator pattern. This solves problems related to profiling, accounting, and buffer registration/de-registration in the context of mixing unrelated allocation methods. This is done by enabling weak function references to "pre" and "post" operations for each of the high level *memkind* heap management APIs. The pre operations can modify any input to the decorated function and the post operation can modify any output from the decorated function. This is a new feature that will be integrated with *memkind* version 0.3.

4.6 Heap Management on NUMA systems

As described earlier, a user-space heap manager is designed to enable the reuse of virtual address ranges previously obtained from the operating system. In Linux, the physical memory backing a virtual address range is mapped by the operating system when the memory is first written to. The default behavior in Linux on a NUMA system is for this physical backing to come from the NUMA node with the smallest NUMA distance from the CPU of the calling thread. While that memory is in use on that CPU the memory will localized due to the smallest NUMA distance constraint. If the thread frees the allocation and puts the virtual address range that has a physical backing into the heap managers free pool, then another thread running on a different CPU may have the same virtual address range returned by the heap manager when it makes an allocation. In this case the physical memory will already be mapped, and it may not be localized to the CPU of the allocating thread.

One simple solution to this problem is to have a separate free pool for each thread. This tempting solution has a side benefit: in addition to ensuring that allocations remain localized, it also eliminates thread contention for access to the recycle pool. The problem with this solution is the mismatch of the required granularity: there are generally many more threads than NUMA nodes. This implies that in the case where all of the physical pages of a NUMA node are mapped by a process, one thread may have plenty of space in its free pool while another thread running on the same CPU may be unable to allocate memory.

Rather than viewing thread contention for the free pool and NUMA locality as the same problem, it is better to separate them. These related problems are dealt with differently by different heap managers, and the strategy chosen by the developers of the *jemalloc* heap manager has evolved over time. The solution given by *jemalloc* in versions 3.5.1 and earlier is to have a thread local free pool for smaller allocations and a shared free pool for large requests. This asymmetry was changed in *jemalloc* version 3.6.0 so that even very large requests were serviced by independent "arenas" which are localized to threads which were assigned to the same arena as a result of the arena selection algorithm. This was changed again recently in the upstream *jemalloc* development branch (not yet been tagged as a release). These changes use a radix tree to enable sharing of large allocations between threads while also avoiding lock contention.

4.7 Determining the Amount Memory

Enabling applications to query the amount of total and free memory available for a given kind may be critical to the logic for determining which data structures will be allocated in which kind of memory. There are several problems with calculating the amount of free memory discussed below. The *memkind* interface offers the `memkind_get_size()` API. This can be used to query the amount of total and free memory associated with a kind. The current default implementation of `memkind_get_size()` is a wrapper around `numa_node_size64()` and, as such, there are some issues with this implementation. The `numa_node_size64()` API measures physical pages free within the operating system's pool. This implies that it does not account for the physical pages consumed by the application heap's free pool which can be used to service allocations without making an operating system request. Additionally there is no accounting for virtual address ranges in use by applications which have not yet been backed with physical pages. Note that the default behavior of Linux is that physical page backing is populated at time of first write not time of allocation (this behavior is configurable via the overcommit sysctl). Another issue is that the `numa_node_size64()`

implementation parses the text in sysfs entries which means calls to `memkind_get_size()` must be strictly outside of the critical path.

For most kinds of memory the "total" amount of memory associated with a kind could be easily cached at time of kind object creation (with some caveats for some file backed or hot-pluggable memory types). The free amount of memory is, of course, dynamic and must be calculated at time of query. Approximating free memory available without making a system call would be quite useful, but could only be relied upon under certain obtainable conditions. The time-of-query versus time-of-use problem makes the "free" number unreliable as well.

The Linux control groups (cgroups) functionality allows a process and its children to have a memory budget that is a fixed fraction of the total regardless of the memory used by other unowned processes on the system. This mechanism is typical in production MPI environments, and this brings us closer to being able to account for memory available without querying the operating system. If we could be sure that all memory requests made by the calling process went through the *memkind* APIs we could solve the accounting problem implied by the virtual/physical mapping, but if other methods are used to obtain memory (e.g. *glibc* `malloc(3)` or direct calls to `mmap(2)`) we must resort to system calls for getting the "free" value (a possible workaround is to intercept such calls via symbol interposition, but this can have negative side-effects). Another issue is that the memory container that cgroups provides can be consumed by operating system buffers dedicated to the process, *e.g.* I/O caching, and these buffers would not be accounted for. If, however, the kind of memory was configured in a way which disables operating system use (i.e. it is located on NUMA nodes which are not "closest" to any CPU), then this issue is mitigated.

Even in the best case, where all allocations are made through the *memkind* interface and careful use of cgroups is done, the amount of free memory is a global property shared by all threads. Therefore, updating the free value requires some sort of locking mechanism or use of 64-bit integer atomics.

5 Performance Evaluation

Since the *memkind* library is built on the *jemalloc* library, the latter one is a natural reference in the context of the allocator performance. We expected the performance characteristics of *memkind* library to be close to those offered by the original, unmodified version of *jemalloc*. In particular, this also applies to the scalable concurrency: providing increased performance with larger numbers of threads.

5.1 Benchmark

To evaluate the performance of the *memkind* library we used a synthetic multi-threaded microbenchmark, designed in the way that allows to compare the performance of various memory allocation libraries, including not only *memkind* and *jemalloc*, but also the standard (and usually default) *libc* implementation. For each library, benchmark generates the identical workload, simply substituting the `malloc(3)/free(3)` calls with `je_malloc()/je_free()` or `memkind_malloc()/memkind_free()` calls respectively. In case of *memkind* library, test were executed on a dynamically created kind using *jemalloc* arenas. This way, we could evaluate the overhead introduced by the *memkind* library.

Each test run consisted of the following three phases: (1) *malloc*: allocation of given number of blocks, (2) *mixed*: freeing of a randomly chosen buffer and allocation of a new buffer (repeated a given number of times), (3) *free*: release of all the allocated buffers. All the tests were run with increasing number of threads [1-32], and performance results were collected separately for each phase. The configured number of operations was evenly distributed among the threads. Due to the differences in handling the small and huge allocation requests in *jemalloc*, performance tests were run independently for various allocation size classes - small, large and huge, both for fixed allocation size (128B, 8KB and 2MB), as well as for randomized size (within a given range): [32B-512B] and [1KB-2MB]. Depending on the configured allocation sizes, there were $10^4 - 10^7$ operations (`malloc()`/`free()`) performed in each run. The most interesting is the performance in the second (*mixed*) phase, as this allows observing the benefits provided by the features like arena-based allocation algorithm and the thread-specific cache.

5.2 Environment

We ran the performance tests on a dual socket system equipped with two Intel® Xeon® processor E5 family @2.60GHz (20MB of L3 cache and 8 physical cores per socket), and with 128GB of DDR3 SDRAM @1600MHz. The system was running CentOS* Linux 7.0 with kernel version 3.18.0. In order to avoid NUMA effects influencing the performance measurements, the benchmark process was bound to run on a single socket only (8 CPUs) using the `numactl` utility.

5.3 Initial Tests and Performance Tuning

The first measurements show that the performance of *memkind* library was, depending on the scenario, up to 10 times worse than the vanilla *jemalloc* build. It appeared that the main reason for that was an algorithm of mapping arenas to threads. The initial implementation was using `sched_getcpu(3)` call to provide per-CPU arena association. In practice, however, the cost of reading the CPU number in each call to `memkind_malloc()` makes this solution extremely sub-optimal. For this reason, the original implementation was replaced with a solution similar to the mechanism used in *jemalloc*. As described in section 4, each thread is assigned an arena at the time it does its first allocation. Once the arena is selected, the arena index is stored in the thread local storage (TLS). Subsequent calls to malloc/free simply obtain the arena index from the TLS.

Introducing the new arena selection algorithm improved performance twice, but the results were still 2-5 times lower than for *jemalloc*. Further investigation showed that the performance gap was related to some limitations of non-standard *jemalloc* API. First, the extended functionality of `je_mallocx()` introduces some minimal overhead (approx. 15%), but the primary reason of the lower performance was the fact that the thread-specific cache (aka *tcache*) is by design not used in case of extended arenas. To fix this problem, the prototype thread cache support for extended arenas was implemented in *jemalloc*, providing the significant performance boost for small and large allocations (Figure 2).

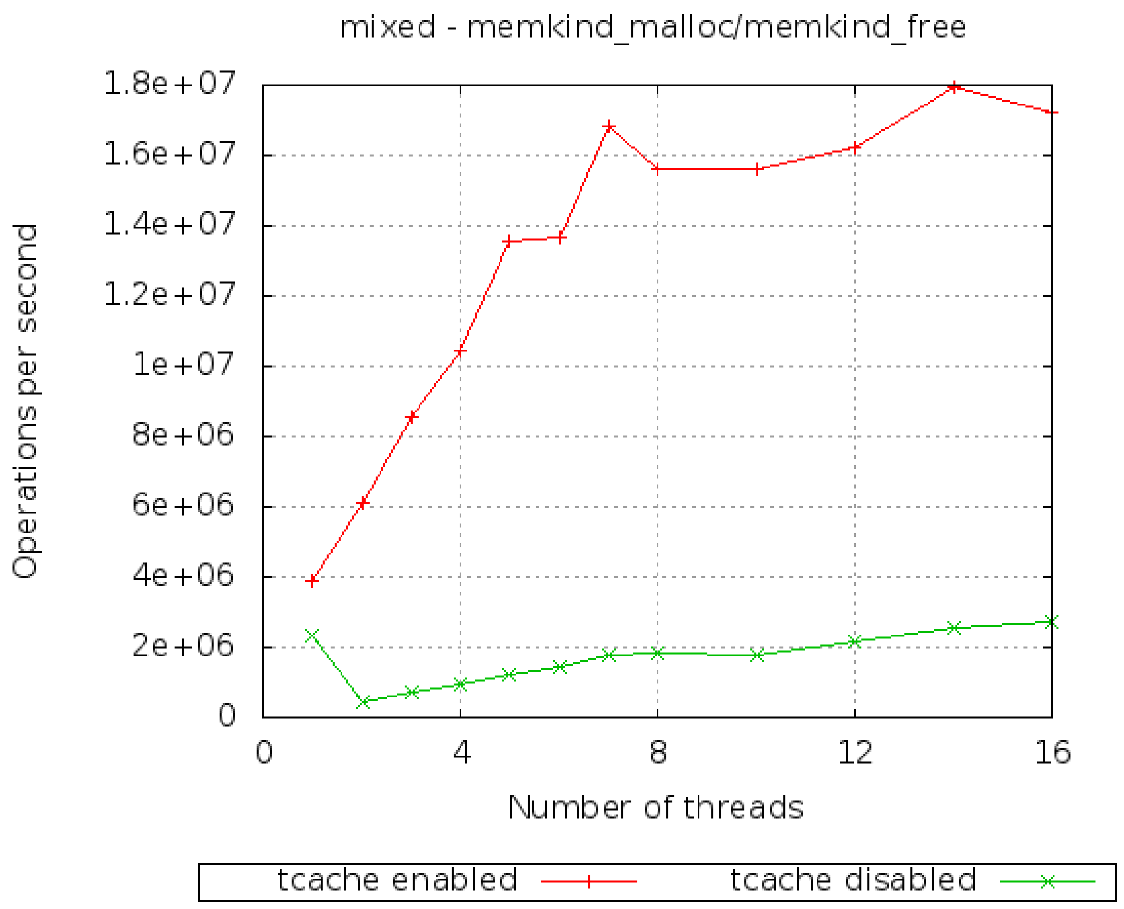

Figure 2: Comparison between memkind using jemalloc with enabled and disabled thread-cache support for extended arenas. (Mixed malloc/free scenario, allocation size - 8KB).

5.4 Results

Eventually, with the modifications described above, the average *memkind* performance is close to *jemalloc*, with the remark that the situation is slightly different for small and huge allocations.

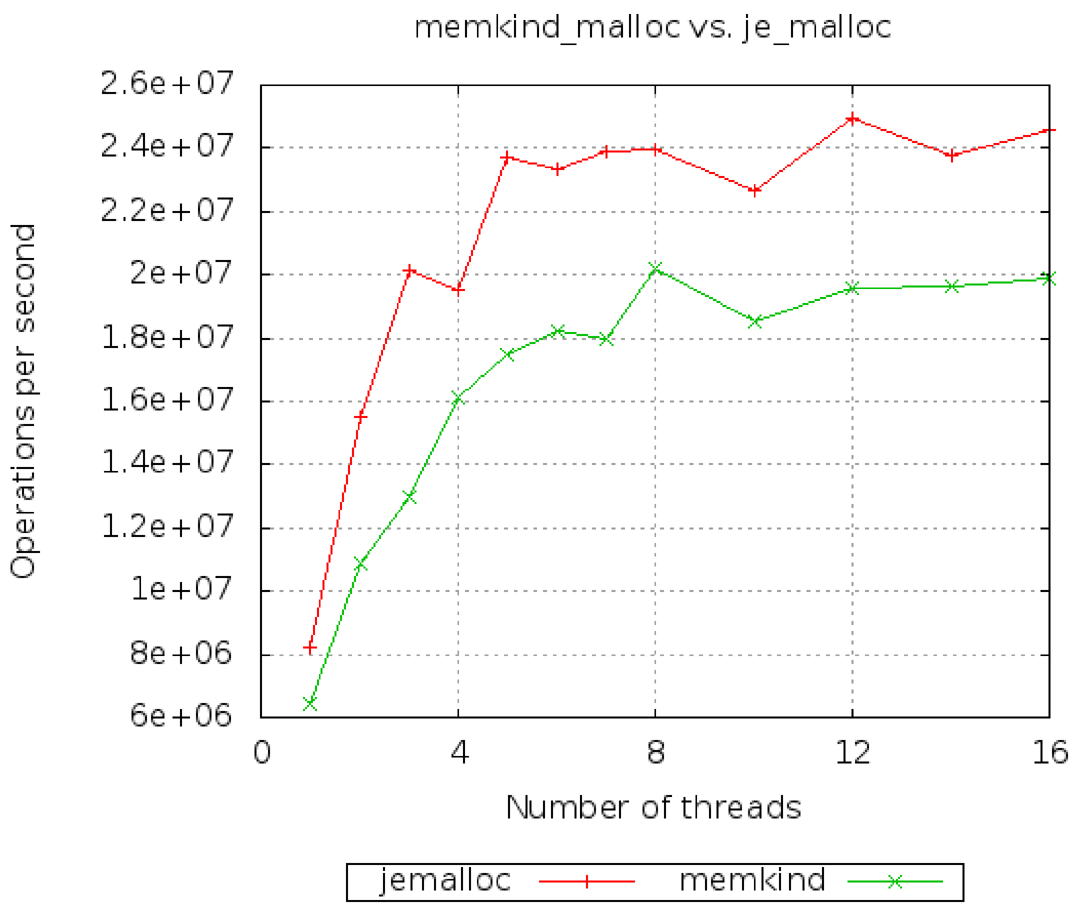

Figure 3: Performance of jemalloc and memkind for randomized allocations size [32B-512B].

For huge allocations, the performance of *memkind* is almost identical to *jemalloc*. On this path, the performance is limited mainly by the lock contention on two global mutexes, which protect a tree of extant huge allocations and a tree of recyclable chunks. Thread cache support and the arena selection algorithm have no impact on the performance, as those mechanisms are not used on huge allocation path (as for *jemalloc* 3.5.1). For small and large allocations [32B-8KB] benchmark results for *memkind* are still lower than for *jemalloc*. In *mixed* scenario (phase 2), the gap is about 15% (Figure 3), and because of the thread cache usage, the decline in performance is related mostly to the overhead introduced by `je_mallocx()` or the *memkind* itself. However, for *malloc* scenario (phase 1), the average performance drop is higher (about 20-35%), which may be related to the additional `memkind_partition_mmap()` callback, used to create a memory mapping with some kind-specific flags or NUMA memory policy.

6 Conclusions

We have described a memory management interface that enables a range of new memory technologies which are coming to market and solves the existing problem of tracking memory policies within the memory management framework. The presented solution is implemented entirely in user space. However, the interface exposes the array of features available through Linux system calls to a higher level in the software stack. We have designed an implementation that limits the total number of required system calls and thus improves the overall performance when compared to a solution without buffer reuse. By effectively segregating the reused buffers the solution enables client specific requirements to remain isolated. Through the decorator interface we enable tracking, accounting and profiling of allocations made by all clients. The high performance computing community relies heavily on middleware solutions to simplify scientific applications. It is our hope that these middleware solutions will adopt the *memkind* interface for memory allocation which will enable user applications to effectively use new memory technologies and advanced policies without modifying the application software nor the operating system.

Acknowlegments and Disclaimers

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energys National Nuclear Security Administration under contract DE-AC04-94AL85000

References

[1] Is-tcmalloc-stable-enough-for-production-use, 2011.

[2] S. Balay, W. D. Gropp, L. C. McInnes, and B. F. Smith. Efficient management of parallelism in object oriented numerical software libraries. In E. Arge, A. M. Bruaset, and H. P. Langtangen, editors, *Modern Software Tools in Scientific Computing*, pages 163–202. Birkhäuser Press, 1997.

[3] R. Barriuso and A. Knies. SHMEM user's guide for C. Technical report, Technical report, Cray Research Inc, 1994.

[4] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *SIGPLAN Not.*, 35(11):117–128, Nov. 2000.

[5] E. D. Berger, B. G. Zorn, and K. S. McKinley. Composing high-performance memory allocators. *SIGPLAN Not.*, 36(5):114–124, May 2001.

[6] E. D. Berger, B. G. Zorn, and K. S. McKinley. Reconsidering custom memory allocation. *SIGPLAN Not.*, 37(11):1–12, Nov. 2002.

[7] W. W. Carlson, J. M. Draper, D. E. Culler, K. Yelick, E. Brooks, and K. Warren. *Introduction to UPC and language specification.* Center for Computing Sciences, Institute for Defense Analyses, 1999.

[8] B. Dawes, D. Abrahams, and R. Rivera. Boost c++ libraries. *URL http://www. boost. org*, 35:36, 2009.

[9] H. C. Edwards, C. R. Trott, and D. Sunderland. Kokkos: Enabling Manycore Performance Portability through Polymorphic Memory Access Patterns. *Journal of Parallel and Distributed Computing*, 74(12):3202 – 3216, 2014.

[10] J. Evans. A scalable concurrent malloc(3) implementation for freebsd, 2006.

[11] J. Evans. Scalable memory allocation using jemalloc, 2011.

[12] M. Folk, G. Heber, Q. Koziol, E. Pourmal, and D. Robinson. An overview of the hdf5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[13] E. Gabriel et al. Open mpi: Goals, concept, and design of a next generation mpi implementation. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 97–104. Springer, 2004.

[14] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc. *goog-perftools. sourceforge. net/doc/tcmalloc. html*, 2009.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the mpi message passing interface standard. *Parallel computing*, 22(6):789–828, 1996.

[16] M. A. Heroux et al. An overview of the trilinos project. *ACM Trans. Math. Softw.*, 31(3):397–423, 2005.

[17] P. Kaminski. Numa aware heap memory manager. *AMD Developer Central*, 2009.

[18] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[19] M. R. Meswani et al. Toward efficient programmer-managed two-level memory hierarchies in exascale computers. In *Proceedings of the 1st International Workshop on Hardware-Software Co-Design for High Performance Computing*, Co-HPC '14, pages 9–16, Piscataway, NJ, USA, 2014. IEEE Press.

[20] M. R. Meswani et al. Heterogeneous memory architectures: A hw/sw approach for mixing die-stacked and off-package memories. In *Proceedings of the International Symposium on High Performance Computer Architecture*, 2015.

[21] J. Nieplocha, R. J. Harrison, and R. J. Littlefield. Global Arrays: A portable "shared-memory" programming model for distributed memory computers. In *Proceedings of the 1994 ACM/IEEE Conference on Supercomputing*, Supercomputing '94, pages 340–349, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[22] R. W. Numrich and J. Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, Aug. 1998.

[23] R. Rew and G. Davis. Netcdf: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, 1990.

[24] R. Riesen et al. Designing and implementing lightweight kernels for capability computing. *Concurr. Comput. : Pract. Exper.*, 21(6):793–817, Apr. 2009.

[25] B. Schäling. *The Boost C++ Libraries.* Boris Schäling, 2011.

[26] A. Singh. *Mac OS X internals: a systems approach.* Addison-Wesley Professional, 2006.

[27] A. Welch et al. Extending the OpenSHMEM memory model to support user-defined spaces. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pages 11:1–11:10, New York, NY, USA, 2014. ACM.

[28] R. W. Wisniewski, T. Inglett, P. Keppel, R. Murty, and R. Riesen. mos: An architecture for extreme-scale operating systems. In *Proceedings of the 4th International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS '14, pages 2:1–2:8, New York, NY, USA, 2014. ACM.