NUMAlloc: Fine-grained NUMA Memory Allocator

ANONYMOUS AUTHOR(S)

The NUMA architecture was proposed to accommodate the hardware trend of the increasing number of CPU cores. However, none of the existing memory allocators can accommodate the heterogeneity of both modern hardware and applications, and therefore they cannot perform well on the NUMA architecture. This paper proposes a novel memory allocator–NUMAlloc, with fine-grained memory management that is designed for the NUMA architecture. NUMAlloc further proposes binding-based memory management and an interleaved heap to ensure locality and load balance. NUMAlloc's memory management is based on the size, the allocation pattern, the shared pattern, the origin of objects, and the location of a thread. According to our extensive evaluation, NUMAlloc has the best performance among all evaluated allocators. It is significantly faster than the second-best one, around 11% on average on an 8-node machine. For the best case, NUMAlloc achieves up to 5.8× performance speedup comparing to the default Linux allocator, and 4.7× faster than the second-best one (TcMalloc). NUMAlloc is also scalable to 128 threads, and is ready for the deployment.

1 INTRODUCTION

The Non-Uniform Memory Access (NUMA) architecture is an appealing hardware solution for the scalability of multi-core era. Compared to Uniform Memory Access (UMA) architecture, the NUMA architecture avoids the bottleneck of using one memory controller: each processor (also known as a domain or node) consists of multiple cores, while all cores of each node can access its own memory controller. Different nodes are connected via high-speed inter-connection, such as Quick Path Interconnect (Intel 2009) or HyperTransport bus (Consortium 2019), in order to form a cache-coherent system that presents an abstraction of a single globally addressable memory. However, NUMA applications may have serious performance issues, if programs have one of the following issues, such as large amount of remote accesses, load imbalance between different memory controllers, and interconnect congestion (Blagodurov et al. 2011, Dashti et al. 2013), as discussed more in Section 2.

The NUMA architecture imposes additional challenges for memory management due to its heterogeneity. General-purpose memory allocators, such as dlmalloc (Lea 1988), Hoard (Berger et al. 2000), TcMalloc (Ghemawat and Menage 2007), jemalloc (Evans 2011), Scalloc (Aigner et al. 2015), were designed for symmetric multiprocessing machines, without considering the difference between different nodes. Thus, these allocators may cause a large number of remote accesses and load balance issues for multithreaded applications running on the NUMA architecture (Kaminski 2012, Yang et al. 2019), which is also confirmed by our evaluations.

There exist NUMA-aware memory allocators (Kaminski 2012, Kim 2013, Yang et al. 2019). Based on our knowledge, Kaminski designs the first NUNA-aware memory allocator (Kaminski 2012), called TcMalloc-NUMA in the remainder of this paper. TcMalloc-NUMA satisfies an allocation from a block of memory that is explicitly bound to a thread's local node. It also introduces per-node freelists to track freed objects for each node: on the one hand, if a per-thread cache has

XXXX-XXXX/2020/5-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

too many freed objects, some objects will be migrated to a per-node freelist that the thread is running on; On the other hand, if a thread is running out of the memory, it will obtain objects from its per-node freelist before obtaining from other freelists. TcMalloc-NUMA improves the performance of some applications by reducing the number of remote accesses (Kaminski 2012). The other two papers also utilize a similar approach (Kim 2013, Yang et al. 2019), but with different focuses.

However, there are still multiple issues in these NUMA-aware allocators. In these allocators, the relationship between each thread and each node is not determined beforehand, which is completely relying on the OS scheduler. Although this design may benefit from better resource optimization provided by the OS scheduler, it adds additional overhead by frequently checking the placement of every thread. More importantly, *this design may break the locality guarantee when a thread is migrated to a new node*. First, this thread is forced to access its stack located remotely in the previous node, and reload all data that may already exist in cache lines there. Even worse, after the migration, this thread may place objects that are originated from the previous node to the freelist of the new node, causing remote accesses unnecessarily afterward. TcMalloc-NUMA has another serious issue that a freed object is always placed into the cache of the current thread, a design originated from TcMalloc (Ghemawat and Menage 2007). The per-thread cache was designed to reduce the lock contention faced by multithreaded applications since objects from per-thread cache can be allocated without acquiring any lock. However, remote accesses will be introduced, if an object is freed by a thread that is running on a different node from its allocation thread. Since such an object is placed to the per-thread cache of the deallocation thread, it will be accessed by this thread that is running on the remote node afterward, causing unnecessary remote accesses.

In summary, existing allocators cannot accommodate the heterogeneity of modern hardware and inside applications. On the one hand, the NUMA hardware introduces the heterogeneity between different nodes, with different access latency. On the other hand, objects inside the same application also have heterogeneity in the shared pattern, the allocation pattern, the size, and the ownership. To achieve good performance, memory management requires to perform more fine-grained memory management. This paper proposes such a novel NUMA-aware memory allocator – `NUMAlloc`. Different from existing memory allocators, `NUMAlloc` *not only specifies the node relationship for each thread, but also manages objects based on their classifications*. Overall, `NUMAlloc` improves both load balance and locality with the following designs.

First, `NUMAlloc` proposes a **binding-based memory management** that ensures both load balance and locality. As described above, since cross-node thread migration will bring the performance issue, `NUMAlloc` binds every thread to a node to eliminate the cross-node migration. Although the integration of task scheduling with memory management (mostly in OS level) has been proposed before (Diener 2015, Wagle et al. 2015, Yang et al. 2008), *`NUMAlloc` is the first memory allocator that is designed based on a special thread binding*. `NUMAlloc` only binds a thread to a specific node, but not to a core, which still allows the migration inside the node when necessary. `NUMAlloc`'s binding also considers load balance by binding threads to different NUMA nodes in an interleaved way. Because of this interleaved binding policy, each node will have a similar number of threads, ensuring load balance between different nodes (and memory controllers). For memory management, `NUMAlloc` further specifies physical page allocations for different ranges of virtual memory and ensures local allocations based on the binding of each thread. In `NUMAlloc`, each thread is guaranteed to have local allocations by controlling both every allocation and every deallocation. An allocation is satisfied either from the per-thread cache, or from the freelist or the heap of the current node. `NUMAlloc` checks every deallocation to ensure the locality. A freed object is only placed into per-thread cache only if it is originated from the same node. Otherwise, it is returned to its origin node. Due to the binding, `NUMAlloc` also ensures that the metadata of each thread and each node is also allocated from the local node, which cannot be done without the thread-binding.

Second, NUMAlloc further proposes **an interleaved heap** for potentially-shared objects, helping reduce the congestion of one memory controller (Blagodurov et al. 2011). Physical pages of the interleaved heap are allocated interleavedly across all physical nodes. But allocating private objects from the interleaved heap may cause unnecessary remote accesses, although providing a better balance. NUMAlloc checks the shared pattern of objects upon allocations, and only allocates potentially-shared objects from the interleaved heap. NUMAlloc further proposes a heuristics method to determine shared objects, as described in Section 3.2.

Third, NUMAlloc further employs **huge page support** to reduce TLB and cache misses, which further introduces the heterogeneity of hardware. Note that this is different from the default support of transparent huge page (THP), which is observed to hurt the performance for the NUMA architecture (Gaud et al. 2014, Panwar et al. 2019). The default THP may cause unnecessary memory waste and page-level false sharing. Differently, NUMAlloc classifies each allocation by its size and allocation pattern, limiting huge pages only for big objects and small objects that are allocated extensively. Due to this design, NUMAlloc's huge page support is never hurting the performance (Section 4.5.3).

In addition to these mechanisms, NUMAlloc is implemented very carefully to achieve good performance. It designs mechanisms to quickly locate the locality and size of each object upon deallocations. NUMAlloc designs an efficient mechanism to move objects between per-thread freelists and per-node freelists, without iterating objects in the freelists. It also reduces memory consumption for transparent huge page support by making multiple threads share the same bag.

We have performed extensive evaluation on synthetic and real applications, and compared NUMAlloc with popular allocators, such as the default Linux allocator, TcMalloc (Ghemawat and Menage 2007), jemalloc (Evans 2011), Intel TBB (Rogozhin 2014), and Scalloc (Aigner et al. 2015). For the performance on real applications, NUMAlloc achieves around 13% speedup on average comparing to the default Linux allocator, which is also 11% faster than the second-best one. For the best case, NUMAlloc runs up to 5.8× faster than the default allocator, and 4.7× faster than the second-best one (TcMalloc). For four synthetic applications, NUMAlloc is running around 2.2× faster on average than the second-best allocator. NUMAlloc is much more scalable than other allocators. NUMAlloc is ready for practical employment, due to its high performance, and good scalability.

Overall, this paper has the following contributions.

- NUMAlloc proposes fine-grained memory management to accommodate the heterogeneity of the hardware and applications.
- NUMAlloc proposes binding-based memory management and an interleaved heap that ensures both load balance and locality. It also proposes an allocation policy to take advantage of huge page support.
- NUMAlloc includes multiple careful designs to reduce the performance and memory overhead, such as fast metadata lookup, and efficient object migration.
- The extensive experiments confirm that NUMAlloc has a better performance and scalability than even widely-used industrial allocators.

The remainder of this paper is organized as follows. Section 2 introduces the NUMA architecture and the corresponding OS support. Section 3 focuses on the design and implementation of NUMAlloc. After that, Section 4 describes its experimental evaluation, and Section 5 discusses the limitation of NUMAlloc. In the end, Section6 discusses some relevant related work, and then Section 7 concludes.

2 BACKGROUND

This section introduces some background that is necessary for NUMAlloc. It starts with the description of the NUMA architecture and some potential performance issues. Then it further discusses existing OS support for the NUMA architecture.

2.1 NUMA Architecture

Traditional computers are using the Uniform Memory Access (UMA) model that all CPU cores are sharing a single memory controller, where any core can access the memory with the same latency (uniformly). However, the UMA architecture cannot accommodate the hardware trend with the increasing number of cores, since all of them may compete for the same memory controller. Therefore, the performance bottleneck is the memory controller in many-core machines, since a task cannot proceed without getting its necessary data from the memory.

The Non-Uniform Memory Access (NUMA) architecture is proposed to solve the scalability issue, due to its decentralized nature. Instead of making all cores waiting for the same memory controller, the NUMA architecture typically is installed with multiple memory controllers, where a group of CPU cores has its memory controller (called as a node). Due to multiple memory controllers, the contention for the memory controller could be largely reduced and therefore the scalability could be improved correspondingly. However, the NUMA architecture also has multiple sources of performance degradations (Blagodurov et al. 2011), including *Cache Contention*, *Node Imbalance*, *Interconnect Congestion*, and *Remote Accesses*.

Cache Contention: the NUMA architecture is prone to cache contention that multiple tasks may compete for the shared cache. Cache contention will introduce more serious performance issue if the data has to be loaded from a remote node.

Node Imbalance: When some memory controllers have much more memory accesses than others, it may cause the node imbalance issue. Therefore, some tasks may wait more time for memory accesses, thwarting the whole progress of a multithreaded application.

Interconnect Congestion: Interconnect congestion occurs if some tasks are placed in remote nodes that may use the inter-node interconnection to access their memory.

Remote Accesses: In NUMA architecture, local nodes can be accessed with less latency than remote accesses. Therefore, it is important to reduce remote accesses to improve the performance.

Based on the study (Blagodurov et al. 2011), node imbalance and interconnect congestion may have a larger performance impact than cache contention and remote accesses. These performance issues cannot be solved by the hardware automatically. Software support is required to control the placement of tasks, physical pages, and objects to achieve the optimal performance for multithreaded applications.

2.2 NUMA Support Inside OS

Currently, the Operating System already provides some support for the NUMA architecture, especially on task scheduling, or physical memory allocation.

For the task scheduling support, the OS provides system calls that allow users to place a task to a specific node. One example of such system calls is `pthread_attr_setaffinity_np` that sets the CPU affinity mask attribute for a thread. Therefore, users may employ these system calls to assign tasks on a specific memory node or even a specific core. However, programmers should specify the task assignment explicitly.

For memory allocation, the OS provides multiple methods to support NUMA related memory management. First, the OS, such as Linux, supports the first-touch or interleaved policy (Diener et al. 2015, Lameter 2013a). By default, the OS will allocate a physical page from the same node as a task that first accesses the corresponding virtual page, also called first-touch policy. First-touch policy maximizes local accesses over remote accesses, but it cannot eliminate remote accesses for shared objects (Yang et al. 2019). The interleaved policy helps to achieve a balanced workload on interconnection and memory controller, avoiding the load imbalance issue described above. However, users may require to invoke a specific system call explicitly to change the allocation policy to be the interleaved policy. Second, the OS also provides some system calls that allow users to specify the physical memory node explicitly, via system calls like `mbind`. On top of these system calls, `libnuma` provides stable APIs for controlling the scheduling and memory allocation policies, and `numactl` allows a process to control the scheduling or memory placement policy inside the user space.

Overall, existing OS or runtime systems already provide some interfaces that allow users to control the scheduling and memory policy for NUMA architecture inside the user space (Yang et al. 2019). However, they all require programmers to specify the policy explicitly, which cannot achieve the performance speedup automatically. `NUMAlloc` relies on these existing system calls to support NUMA-aware memory allocations explicitly, as further described in Section 3.

3 DESIGN AND IMPLEMENTATION

`NUMAlloc` is designed as a drop-in replacement for the default memory allocator. It intercepts all memory allocation/deallocation invocations via the preloading mechanism. Therefore, there is no need to change the source code of applications, and there is no need to use custom OS or hardware. Multiple components that differentiate it from existing allocators are further discussed in the remainder of this section.

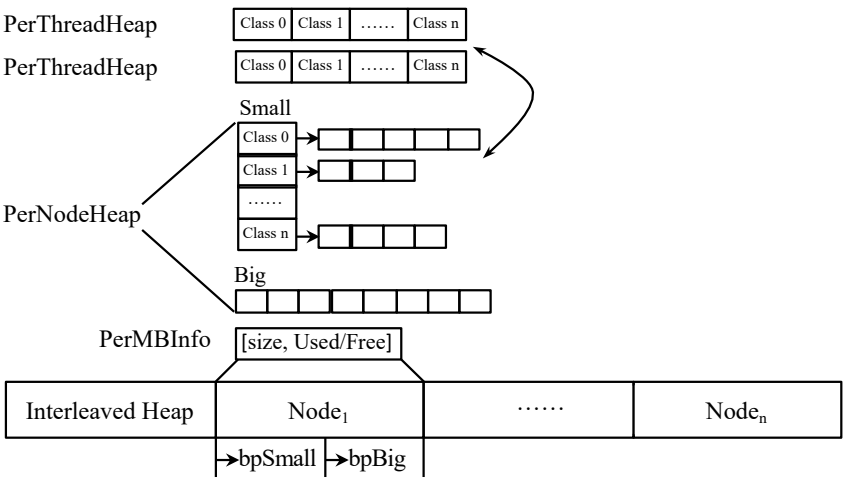

Fig. 1. Overview of `NUMAlloc`'s Heap Layout.

3.1 Binding-Based Memory Management

Existing allocators typically do not schedule threads explicitly, but relying on the default OS scheduler. The OS scheduler performs very well for the UMA architecture that the latency of accessing

the memory controller is the same for every core. However, the NUMA architecture imposes additional challenges (Majo and Gross 2015). If a thread is migrated to a new node, it can break the locality guarantee. It has to access its own stack remotely and reload all data that is already in cache lines of the old node, resulting in a higher memory latency. Also, it may complicate the memory management of objects in its per-thread cache, as further explained in Section 1.

Due to the importance of avoiding cross-node migration, `NUMAlloc` embeds a topology-aware task assignment that also considers load balance. To balance the workload of each node, `NUMAlloc` utilizes a round-robin manner to assign the tasks to different nodes. Basically, a newly-created thread will be assigned to a node that is different from its predecessor. This binding policy ensures that each processor/node will have a similar number of threads, and therefore a similar workload. An alternative approach is to assign the same number of threads as cores to one node, and then assign subsequent threads to the next node. However, this policy has two issues. First, it may cause significant performance issue for applications with the pipeline model due to remote accesses, if threads of different stages are assigned to different memory nodes. Second, it may not fully utilize all memory controllers, if the number of threads is less than the number of cores in total. In the implementation, `NUMAlloc` recognizes the hardware topology via the `numa_node_to_cpus` API, which tells the relationship between each CPU core and each memory node. It intercepts all thread creations in order to bind a newly-created thread to a specific node. `NUMAlloc` employs `pthread_attr_setaffinity_np` to set the attributes of a thread, and passes the attribute to its thread creation function. Therefore, every thread is scheduled to the specified node upon the creation time. Note that a thread is pinned to a node in `NUMAlloc`, instead of a core, which still allows the OS scheduler to perform the load balance when necessarily.

Based on its thread assignment, `NUMAlloc` designs a node-aware memory management. Since a user-space memory allocator only deals with virtual memory, but not physical memory, *`NUMAlloc` binds a range of virtual memory to a particular node via the `mbind` system call*. As shown in Fig.1, each node is mapped to four terabytes (TB) initially, and the interleaved heap is placed in a separate place (Section 3.2). Therefore, `NUMAlloc` is able to compute the physical node information based on a virtual address. More specifically, it could divide the offset (from the heap beginning) by the size of each per-node heap to compute its node index. This design takes the advantage of the huge address space of 64-bits machine to achieve the quick lookup.

Each per-node heap is further divided into two parts, one for small objects (managed with the `bpSmall` bump pointer) and one for big objects (with the `bpBig` pointer). An object with the size larger than 512 KB will be treated as a big object, and others are small objects. The space for big objects are separated from that for small objects so that we could utilize huge pages for big objects, as discussed in Section 4.5.3. Another difference between small objects and big objects is that small objects are managed by size classes, while big objects of each node are tracked in a global free list. For small objects, `NUMAlloc` utilizes the "**Bi**g-**B**ag-**o**f-**P**ages" (BiBOP) mechanism that all objects in the same bag will have the same size class, and separates the metadata from actual objects. In order to track the size information of objects, `NUMAlloc` utilizes one megabytes (MB) as a basic unit, where the information is tracked in a PerMBInfo data structure (as shown in Fig. 1. A big object in `NUMAlloc` is always aligned to megabytes, and small objects in the same MB (called as a bag) will has the same size. For a big object that is larger than 1MB, all corresponding bytes in the PerMBInfo will be set to the specific size upon the allocation. `NUMAlloc` embeds the availability information of each chunk to the `PerMBInfo` structure, using the lowest significant bit of the size. `NUMAlloc` is able to coalesce multiple continuous big objects into a bigger object upon deallocations.

For each size class of small objects, `NUMAlloc` maintains a freelist for each node and for each thread. Every size class has 16-byte difference when the size is less than 128 bytes, and then 32 bytes apart for objects less than 256 bytes. When an object is larger than 256 bytes, its size class

will be always power of two. Except those ones stated in Section 3.2, NUMAlloc ensures node-aware memory allocations that an allocation is guaranteed to be allocated from the local node physically. *For each deallocation, NUMAlloc ensures that an object is always placed into a freelist based on the origin of this object*. More specifically, for small objects, it will be placed into the current thread's freelist **only if** it is originated from the same node that the current thread belongs to. Otherwise, it will be placed to the per-node freelist of its original owner. A big object is always placed into the freelist of its owner. *NUMAlloc also handles allocations carefully to ensure the locality*. For small objects, an allocation will be satisfied from the freelist of the current thread, where all freed objets is originated from the local node, if there exists some freed objects. Otherwise, the thread will migrate some freed object from the per-node freelist. If its per-node freelist is empty, then it will get a batch of never-used objects from the bag of a size class. Allocations of big objects is always satisfied from per-node freelist first, before getting a new object from the per-node heap.

Cache Warmup Mechanism: For small objects, NUMAlloc also borrows the cache warmup mechanism of TcMalloc (Ghemawat and Menage 2007). TcMalloc utilizes a mmap system call to obtain multiple pages (depending on the class size) from the OS each time, when it is running out of the memory for one size class. For such a memory block, TcMalloc inserts all objects of this block into its central freelist at one time. Since TcMalloc utilizes the first word of each object as the pointer for the freelist, this mechanism warms up the cache by referencing the first word of each object during the insertion. According to our observation, this warmup mechanism improves the performance of one application (raytrace) by 10%. Based on our understanding, the performance improvement is caused by data prefetches, since inserting objects to the freelist has a simple and predictable pattern. NUMAlloc employs a similar mechanism for small objects with the size less than 256 bytes, and adds all objects inside a page to the per-thread freelist.

3.2 Interleaved Heap

NUMAlloc proposed an interleaved heap for shared objects. Based on our observation, most NUMA performances issues identified by existing NUMA profilers are related to shared objects (Lachaize et al. 2012, Liu and Mellor-Crummey 2014). These shared objects are typically allocated in the main thread, but are passed to children threads later. Allocating the physical memory for shared objects in one node may introduce load imbalance issue, and therefore causing significant number of remote accesses and potential interconnect congestion. Therefore, NUMAlloc reserves a range of memory for shared objects, called as "Interleaved Heap" in Fig. 1. NUMAlloc utilizes the mbind system call to specify that physical pages of this range will be allocated from all nodes interleavedly. This design helps balance the volume of memory accesses of all memory controllers, reducing interconnect congestion and load imbalance.

During the implementation, NUMAlloc only allocates potentially-shared objects from the main thread in the interleaved heap. Although it may benefit the performance, if shared objects allocated in children threads are also using the interleaved heap. However, the overhead of tracking shared objects is typically larger than the performance benefit, based on our evaluation (skipped in the paper).

It is very important to identify shared objects correctly and efficiently. NUMAlloc utilizes the allocation/deallocation site to identify shared objects: every allocation of the main thread is treated as a shared one initially, and is allocated from the interleaved heap. The callsite-based method is aligned with the intuition, since the shared behavior of a callsite is determined by the program logic. When a newly-allocated object is deallocated before creating children threads, all objects from the corresponding callsite are considered to be private objects, and then are only allocated from the per-node heap afterward. Overall, NUMAlloc monitors memory allocation/deallocation pattern to identify whether a corresponding callsite is shared or not.

NUMAlloc requires to track the shared pattern for each callsite. For the performance reason, NUMAlloc further proposes to utilize the sum of the stack position and the return address of the allocation to identify a callsite, called "*callsite key*". When memory allocations are invoked in different functions, their stack positions are likely to be different. The return address (of the application) tells the invocation placement inside the same function. However, this design cannot completely avoid mis-identification issue, where multiple allocations inside the allocation wrapper will be treated as the same callsite. However, the mis-identification will not cause any correctness issue, but only performance issue. During the implementation, we have thought about two other mechanisms. One is to obtain the callsite correctly with the `backtrace` function, but it is too slow to be used in production environment. The other mechanism will require the recompilation to encode calling context explicitly (Sumner et al. 2010, Zeng et al. 2014).

NUMAlloc utilizes a hash table to track the status of every callsite. Upon every allocation of the main thread, NUMAlloc checks the status of the allocation callsite, with the callsite key as described above. If the callsite is identified as a shared callsite, the current allocation will be satisfied from the interleaved heap. Otherwise, it will be allocated from the per-node heap. Upon deallocations, NUMAlloc marks an allocation callsite as private, if an object from this callsite has been deallocated before creating other threads.

3.3 Explicit Huge Page Support

NUMAlloc employs explicit huge page support to further improve the performance. The size of a huge page is 2 megabytes (MB) in the current OS. Based on the existing study (Gorman 2010), huge pages can reduce Translation Look-aside Buffer (TLB) misses that may significantly affect the performance, since a huge page covers a larger range of memory than a normal page (4 KB). Reducing TLB misses helps reduce the interferences on the cache utilization caused by TLB misses, and therefore reduces cache misses. Huge pages could also reduces the contention in the OS memory manager, since it imposes less page faults and requires much less time on page table management. We observed around 10% performance improvement for some applications, when we are using huge page support.

However, existing studies showed that the transparent huge page support of the OS is not good for the performance (Gaud et al. 2014, Panwar et al. 2019). In fact, it may have some harmful impact on the performance of NUMA systems. First, it can cause the *hot page effect* when multiple frequently-accessed objects were mapped to the same physical page, causing the overloading of the corresponding memory node. Second, huge pages are more prone to *page-level false sharing*, when multiple threads are accessing different data inside the same page. Besides that, the huge page may increase the memory footprint (Maas et al. 2020), if only a partial range of a huge page is accessed.

NUMAlloc utilizes huge pages explicitly to avoid these issues. Ideally, if huge pages are only utilized for private objects, then there will be no hot page effect and page-level false sharing. Also, if huge pages are only utilized for big objects that are larger than the page size, or if all small objects in a huge page will be allocated, then there is no need to worry about the memory consumption. NUMAlloc takes these consideration into account. NUMAlloc only employs huge pages for large objects with the size larger than a huge page, and for small objects that are predicted to be used a lot. For the latter one, NUMAlloc employs the history information to predict, and only uses huge pages for a size class that has allocated at least one bag of objects.

3.4 Efficient Object Migration

NUMAlloc maintains per-thread heaps and per-node heaps in order to reduce the synchronization overhead. This design also requires to move freed objects between different freelists. When a per-thread freelist has too many objects, some of them should be moved to the thread's per-node freelist

so that other threads could re-utilize these freed objects, reducing the memory consumption. Similarly, each per-thread list may need to obtain freed objects from its per-node heap, when a thread is running out the memory. Frequent migrations require an efficient mechanism. `NUMAlloc` utilizes singly link lists to manage freed objects, imposing an additional challenge of migrating objects efficiently.

One straightforward method is to traverse the freelist from the head in order to collect a specified number objects, and then moves them at a time. Actually both TcMalloc and TcMalloc-NUMA utilize such a mechanism, which actually has multiple issues. First, traversing a list will actually pollute the cache of the current thread unnecessarily, especially when a thread is moving out objects. `NUMAlloc` utilizes the first word of each freed object as the pointers for the linked list, where traversing these objects will bring these objects to the cache that the current thread do not need. Second, it may introduce thread contention, when multiple threads are migrating freed objects from the per-node freelist concurrently. We observed 20% slowdown for some applications, due to this straightforward mechanism. `NUMAlloc` further proposes an efficient mechanism to migrate objects efficiently.

In order to avoid the pollution on the per-thread cache, each per-thread freelist maintains two pointers that pointing to the least recent object (shown as the `Tail` object) and the nth object separately, as shown in Fig. 2. Maintaining the pointer to the nth object requires only a forward traverse to obtain the pointer of $n-1th$ object, and then a thread can migrate n objects (between $n-1th$ and the `Tail` object) easily. After the migration, the `Tail` pointer can be set to the original nth object. However, this mechanism alone cannot reduce the lock contention when multiple threads are concurrently obtaining objects.

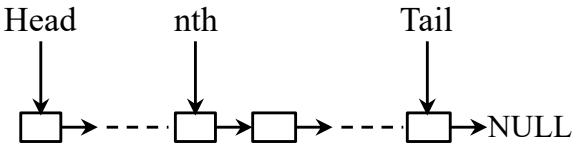

Fig. 2. Avoiding the traverse of per-thread freelist

In order to avoid the bottleneck of the per-node heap, `NUMAlloc` introduces a circular array of freelists as shown in Fig. 3, where the number of entries is a variable that can be changed by the compilation flag. Each entry has two pointers, head and tail separately. In order to support the put and get operations to the freelist, this array has two pointers, `toGetIndex` and `toPutIndex`. If a thread tries to obtain freed objects from the per-node heap, it will obtain all objects pointed by the `toGetIndex`, and increment the index afterward. If the freelist pointed by the `toGetIndex` has no freed objects, there is no freed objects in the per-node freelist for this size class. The put operation will utilize the pointer `toPutIndex`. There are two scenarios for the put operation. First, a thread may put an freed object directly to the per-node freelist, if this object is originated from a different node that the thread does not belong to. In this case, the object will be placed into the freelist pointed by the `toPutIndex`, but the index is not updated after the deallocation. Second, when freed objects in a per-thread freelist is

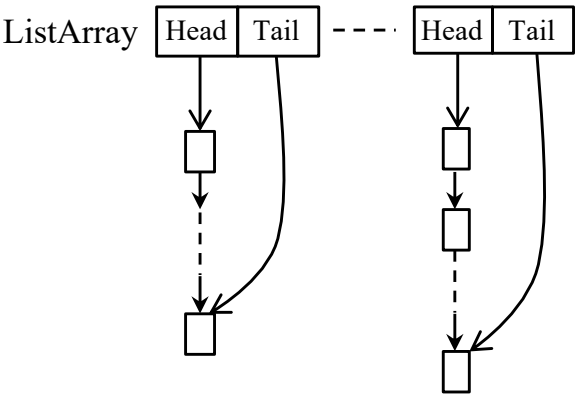

Fig. 3. An array of freelists for per-node heap

above the predefined watermark, the thread will migrate a batch of objects to the freelist pointed by the `toPutIndex`. After this migration, the current freelist is considered to be full, and will update the index to the next entry in the circular array.

3.5 Other Mechanisms

Node-Local Metadata: `NUMAlloc` guarantees that all of the metadata is always allocated in the same node, based on its thread binding as described in Section 3.1. Such metadata includes per-node and per-thread freelists for different size classes, and freelists for big objects. Similarly, `NUMAlloc` utilizes the `mbind` system call to bind the memory to a specific node.

Encouraging Memory Reutilization: In `NUMAlloc`, every thread has its own freelists for each size class so that there is no need to acquire the lock when an allocation can be satisfied from its per-thread heap, similar to TcMalloc. That is, two threads will not share the same per-thread heap. However, some applications may create new threads after some threads have exited. `NUMAlloc` re-utilizes the memory for these exited threads. Basically, `NUMAlloc` intercepts thread joins and cancels so that it can assign heaps of exited threads for newly-created threads, and re-utilize their heaps correspondingly.

Transparent Huge Page Support: During the development, we noticed that excessive memory consumption can be imposed when the OS enables transparent huge pages by default. In order to reduce memory consumption, `NUMAlloc` makes multiple threads share the same bag (for the same size class), instead of having a separate bag for each thread. If each thread is running out of the memory, it obtains multiple objects at a time from the corresponding bag. Currently, if a class size is less than one page, then we will at most get objects with the total size of one normal page. Otherwise, it will get 4 objects (with the size less than 64 KB) or 2 objects afterward. Based on our evaluation, this mechanism actually reduces the memory consumption for multiple times for a machine with 128 cores and 8 nodes, with the transparent huge page support by default.

4 EXPERIMENTAL EVALUATION

This section aims to answer the following research questions:

- **Performance:** How is `NUMAlloc`'s performance, comparing to popular allocators and NUMA-aware allocators? (Section 4.2)
- **Memory Consumption:** What is the memory consumption of `NUMAlloc`? (Section 4.3)
- **Scalability:** How is the scalability of `NUMAlloc`? (Section 4.4)
- **Design Decisions:** How important every design choice can actually impact performance? (Section 4.5)

4.1 Experimental Setup

`NUMAlloc` was evaluated on two different machines as specified in Table 1. Typically, machine A has 2 nodes, with 40 cores in total, while machine B has 8 nodes with 128 cores. For machine B, any two nodes are less than or equal to 3 hops. For the evaluation, the hyperthreading was turned off on both machines. The performance data shown in this paper is the average of 10 runs, in order to avoid any bias caused by unexpected events.

4.2 Performance Evaluation

Multithreaded applications are chosen to evaluate the performance. The number of threads is set to be the the total number of cores on two machines if possible, with 40 threads on machine A and 128 threads on machine B. For applications with multiple phases (e.g., `ferret` and `dedup`) or work only

Table 1. Machine specifications for evaluation

System	**Machine A**	**Machine B**
CPUs/Model	Xeon Gold 6138	Xeon(R) Platinum 8153
CPU Frequency	2.10GHz	2.00GHz
NUMA Nodes	2	8
Physical Cores	2×20	8×16
Node Latency	local: 1.0 1 hop: 2.1	local: 1.0 1 hop: 2.1 2 hops: 3.1
Interconnect Bandwidth	8GT/s	10.4GT/s
Linux	Ubuntu 18.04	Debian 10
Compiler	GCC-7.5.0	GCC-8.3.0

for power-of-two threads, we chose the maximum number of threads that is smaller than but is close to the total number of cores. We compare `NUMAlloc` with multiple popular allocators, such as the default Linux allocator, TcMalloc-2.7 (Ghemawat and Menage 2007), TcMalloc-NUMA (Kaminski 2012), jemalloc-jemalloc-5.2.1 (Evans 2011), Intel TBB–2020.1 (Rogozhin 2014), and Scalloc-1.0.0 (Aigner et al. 2015).

We evaluate all applications from the PARSEC suite (Bienia and Li 2009), and seven real applications like `Apache httpd-2.4.35`, `MySQL-5.7.15`, `Memcached-1.4.25`, `SQLite-3.12.0`, `Aget`, `Pfscan`, and `Pbzip2`. The inputs for these applications are listed as follows. PARSEC applications are using native inputs (Bienia and Li 2009). For MySQL, we use `sysbench` with 40 and 128 threads separately, each issuing 100,000 requests. The `python-memcached` script is used to exercise `Memcached`, with 3000 loops to get the sufficient runtime (Reifschneider 2013). The `ab` is used to test `Apache` server (Foundation 2020), by sending 1,000,000 requests in total. `Aget` is tested by downloading a 30-MB file, and `Pfscan` is tested by searching a keyword in a 500MB data. In terms of `Pbzip2`, we test it by compressing 10 files with 30MB each. Finally, SQLite is tested through a program called `threadtest3` (Developers. 2019).

The evaluation results can be seen in Fig. 4b and Fig. 4a separately, where the runtime of each allocator is normalized to that of the Linux's default allocator. That is, a lower bar indicates a better performance. In the remainder of this paper, all performance data are using the same format. By default, `NUMAlloc` will embed with the interleaved heap support. However, two applications, `canneal` and `raytrace`, have a much worse performance when the interleaved heap is enabled, as further discussed in Section 4.5.2.. These two figures show the best data for these two applications, without the support of the interleaved heap.

Overall, `NUMAlloc` has the best performance on both machines. Comparing to the default allocator, `NUMAlloc` is 6% faster on Machine A and 13% faster on Machine B. TcMalloc is the second-best one among all allocators, which is only 1% faster than that of the default allocator. For the best case (e.g., `fluidanimate`) on the 8-node machine, `NUMAlloc` is running up to 5.8× faster than the default Linux allocator, and it is 4.7× than the second-best one–TcMalloc. Comparing with the other NUMA-aware allocator – TcMalloc-NUMA (Kaminski 2012), `NUMAlloc` runs 6% faster on Machine A (2-node) and 23% faster on Machine B with 8 nodes. We also notice that `NUMAlloc` achieves a much better performance on the machine with more hardware cores and more NUMA nodes, which indicates `NUMAlloc`'s scalable design.

The default Linux allocator achieves a reasonable performance on the NUMA architecture due to its arena-based design. Based on our analysis, the Linux allocator will always return an object

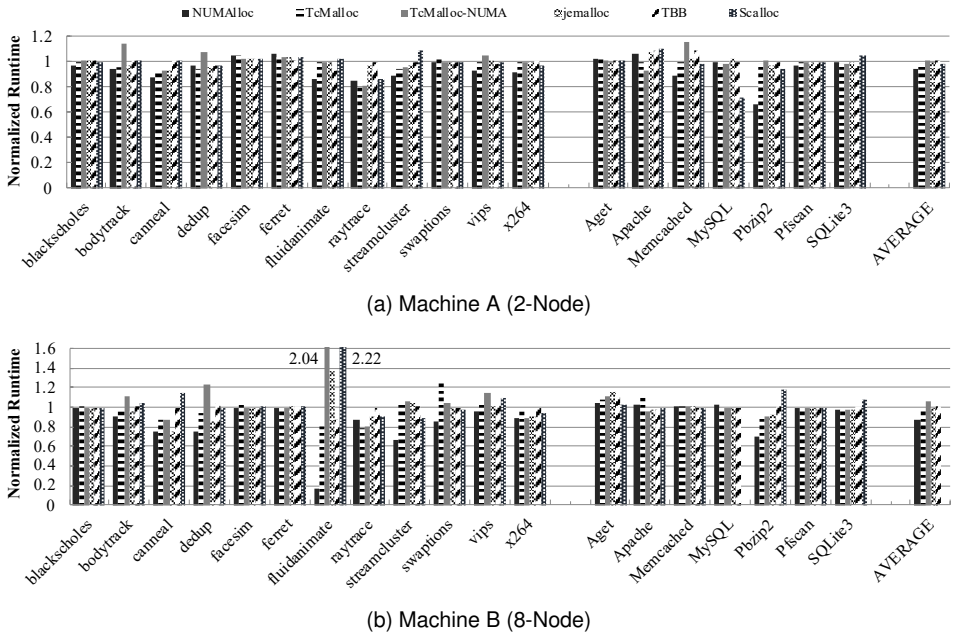

Fig. 4. Performance of different allocators, where all data are normalized the default Linux allocator.

back to its original arena, and then allocate such objects to the thread owning this arena afterward. This design is integrating well with Linux's first-touch allocation policy (Lameter 2013b), where a page is allocated in the same node as the thread that touches it first. Therefore, an object is typically allocated from the local node of its allocation thread. An object that is deallocated from a different thread will be always returned back to its original arena, and then will be re-utilized by its original allocation thread locally. In contrast, other allocators typically utilize a per-thread cache to store objects that are deallocated by the current thread, which may lead to remote accesses as described in Section 1.

TcMalloc-NUMA is the only available allocator that is claimed to support the NUMA architecture (Kaminski 2012). However, its performance is not worse than that of TcMalloc. TcMalloc-NUMA is based on TcMalloc-0.97 (released in 2008), which does not have new features of TcMalloc-2.7 (the version for our evaluation). TcMalloc-NUMA imposes the largest overhead `fluidanimate` (2.04×) on the Machine B (8-node), mostly due to no thread binding support. TcMalloc-NUMA does not implement following mechanisms of `NUMAlloc`, including binding-based assignment, the interleaved heap, explicit huge page support, and efficient object migration.

4.3 Memory Consumption

We also measure memory overhead of different allocators for these applications. For non-server applications, such as `Aget`, `Pbscanf`, `Pbzip2` and all PARSEC applications, we utilized the sum of the `maxresident` output from the `time` utility and the size of huge pages, since the `time` output does not include huge pages. To determine the size of huge pages, a script is used to periodically collect the number of huge pages by reading from `/proc/meminfo` file, and then the maximum value of huge pages are used. Memory assumption of server applications, such as `MySQL`, `SQLite`, and `Memcached`, `Apache`, is collected by the sum of both `VmHWM` and `HugetlbPages` fields from `/proc/PID/status` file, after the corresponding client exits.

Table 2. Memory consumption of Different Allocators in Machine B (8-node)

Applications	Memory Usage (MB)						
	Linux's Default	NUMAlloc	TcMalloc	TcMalloc-NUMA	jemalloc	TBB	Scalloc
blackscholes	615	509	621	623	633	615	630
bodytrack	37	161	45	46	570	37	1994
canneal	888	879	774	757	1294	888	36149
dedup	912	1236	983	1023	1389	912	8556
facesim	560	500	603	601	1133	547	3056
ferret	184	493	195	183	596	184	3377
fluidanimate	470	392	483	484	481	470	3437
raytrace	1288	1472	1092	1543	1287	1288	4398
streamcluster	113	105	123	121	127	113	193
swaptions	33	268	16	21	540	37	1817
vips	228	536	248	269	778	227	3681
x264	2859	2721	3047	3064	3719	2859	5402
Aget	8	74	11	10	93	8	80
Apache	8	34	10	9	10	4	42
Memcached	16	80	25	24	41	18	263
Mysql	277	732	314	315	500	276	N/A
Pbzip2	463	747	817	813	1121	454	4881
Pfscan	522	542	528	528	535	522	554
Sqlite3	45	284	60	75	139	44	681
Total	**9527**	**11763**	**9993**	**10510**	**14986**	**9502**	**79190**

Memory overhead of different allocators is listed in Table 2, which was evaluated on Machine B. Machine B was chosen because it has more nodes and physical cores. In total, NUMAlloc's memory consumption is around 23% more than that of the default Linux allocator. For applications with small footprint, NUMAlloc may utilize up to 9.3× more memory, such as Aget. Intel TBB allocator has the smallest memory consumption, and the default Linux allocator is the second-best one. Scalloc is the worst one in terms of memory consumption, which consumes around 8.3× more memory that that of TBB.

Based on our analysis, the following reasons may lead to NUMAlloc's more memory consumption. First, Machine B has transparent huge page support by default, which actually shows the worst case for NUMAlloc. The OS will utilize huge pages if a memory area is larger than the size of a huge page (2MB). Since NUMAlloc utilizes mmap to allocate a huge chunk of virtual memory, this makes all heap memory for real objects will be allocated from huge pages. Currently, NUMAlloc also utilizes 1MB as the superblock for each size class, making objects of a size class that will occupy at least 1MB even if it only uses an object inside. Therefore, an application with many size classes will waste more memory. NUMAlloc makes all threads share the same bag for each size class, as described in Section 3.5, which effectively reduces its memory consumption by multiple times.

Scalloc has excessive memory consumption, since its design does not support transparent huge pages very well. Similar to NUMAlloc, Scalloc utilizes a mmap system call to allocate a continuous huge region of virtual memory from the underlying OS. Since every thread will get a virtual span (2MB) for each size class in Scalloc, it will utilize 2MB physical memory even if only a word is touched. Differently, NUMAlloc avoids this issue as described in Section 3.5.

4.4 Scalability

We evaluated the scalability on the Machine B. Machine B is chosen since it has more cores and more nodes, making it better to evaluate the scalability. In order to understand the scalability of NUMAlloc when comparing to other allocators, we evaluate four synthetic applications from Hoard (Berger et al. 2000), including threadtest, larson (Larson and Krishnan 1998), cache-scratch and cache-slash, which is also employed by existing work (Aigner et al. 2015). The reason of utilizing synthetic applications is that they have a better scalability by design (Aigner et al. 2015). Since other allocators cannot specify the configuration, we only evaluate the scalability with different number of threads. For NUMAlloc, we maximize the number of threads on each node. For instance, the result of 32 threads will use 2 node, since each node has 16 cores. The corresponding data is shown as Fig. 5.

Among these applications, cache-scratch tests passive false sharing, and cache-thrash tests active false sharing. False sharing occurs when multiple threads are concurrently accessing different words in the same cache line. Passive false sharing is introduced upon deallocations, where a freed object can be utilized by another thread. In contrast, active false sharing is introduced during the initial allocations, where multiple continuous objects sharing the same cache line are allocated to different threads. For these false sharing tests, we use 100,000 inner-loop, and 100,000 iterations with 8-byte objects. NUMAlloc will not introduce active false sharing, since each thread will get a page of objects initially. Although NUMAlloc might introduce passive false sharing due to its per-thread cache design, it avoids remote allocations across the node. We believe that is the major reason for its better performance. Other allocators do not have such mechanisms. That is the reason why NUMAlloc is one of the best allocators for cache-scratch, and achieves much better speedup than all other allocators in cache-thrash (30% faster than the second-best one), as shown in Fig. 5. TcMalloc has serious both active and passive false sharing issue, which is the major reason that it does not perform well on these applications.

larson is to simulate a multithreaded server that could respond to requests from different clients. Each thread will receive a random number of objects in the beginning, perform a random number of allocation and deallocations to simulate the handler for processing requests, and then pass objects to the next thread. We test larson for 10 seconds with 1,000 objects for 10,000 iterations, where each allocation is between 7 bytes and 2048 bytes. As shown in Fig. 5, NUMAlloc is around 16% faster than the second-best allocator–TcMalloc.

threadtest is an application that performs a large number of allocations and deallocations within a specified number of threads. Also, it allows to specify how much work to be done between each allocation and deallocation. For threadtest, we use 100 iterations, 1,280,000 allocations, 0 work, and 64-byte objects (for the allocation). This benchmark will stressfully test the performance overhead of allocation and deallocation. For this application, NUMAlloc is 2.6× faster than the second-best one (jemalloc), when there are 128 threads.

Overall, NUMAlloc has the best overall performance when running on the 128-core machine. It is running 79% faster than the second-best one (Scalloc), and 2.2× faster than the default Linux allocator on Machine B with 8 nodes. Multiple reasons contribute to the good performance of NUMAlloc: NUMAlloc imposes very minimal system call overhead, and little synchronization overhead. Also,

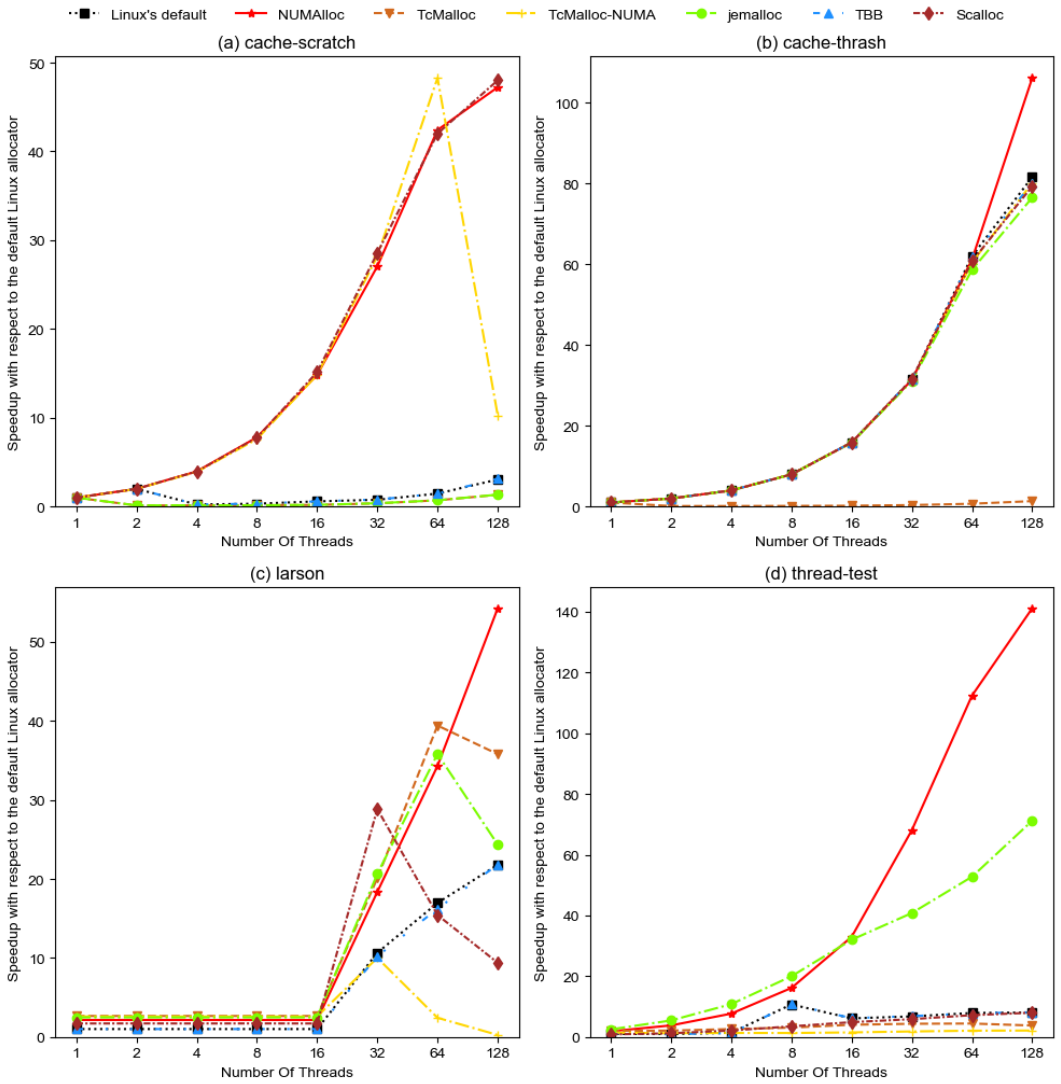

Fig. 5. Scalability of different allocators
all data is normalized to the runtime of the default Linux allocator with one thread.

it introduces less remote accesses than all other allocators, due to its NUMA-aware design. Other allocators have more or less false sharing issue, or incurs remote accesses.

4.5 Design Choices

This section further confirms `NUMAlloc`'s multiple design choices. Some mechanisms are very basic, such as node-aware memory allocation, which is mandatory for NUMA-aware memory allocators, and cannot be evaluated separately. Therefore, they are not evaluated here. Some design choices may only help on one or two applications, such as efficient object migration, which are skipped as well.

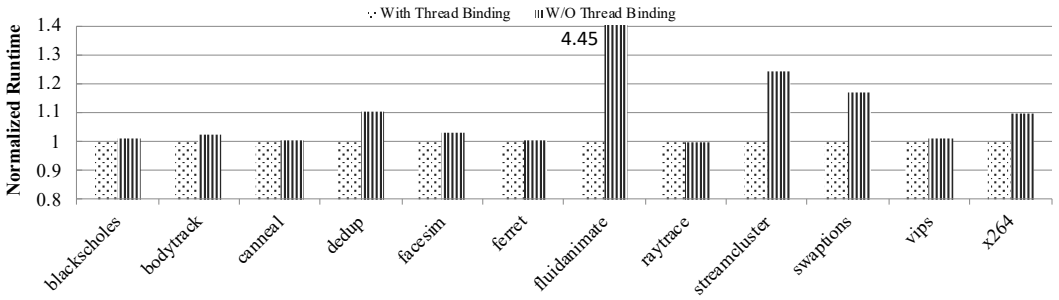

Fig. 6. Normalized runtime with and without thread binding on Machine B (8-Node) (using the Linux allocator)

4.5.1 Thread Binding. Fig. 6 shows the performance difference with and without thread binding. `NUMAlloc` relies on thread binding, which can not be disabled explicitly. For this experiment, we utilize the default Linux allocator, but designing an additional library to bind threads to different nodes in a round-bin way, similar to `NUMAlloc`'s thread binding policy. We observe that the thread binding improves the performance significantly for some applications. Among them, `fluidanimate` runs around 4.5× faster with the thread binding, while `streamcluster` runs 22% faster than the default one without the binding. This clearly indicates that the thread binding will benefit the performance. Based on our analysis, there are two reasons for this performance speedup. First, a thread will not be migrated to a different core, avoiding unnecessary remote accesses caused by cross-node migration, as further discussed in Section 1. Second, `NUMAlloc`'s thread binding balances the workload, thus reducing the congestion of interconnect or one memory controller.

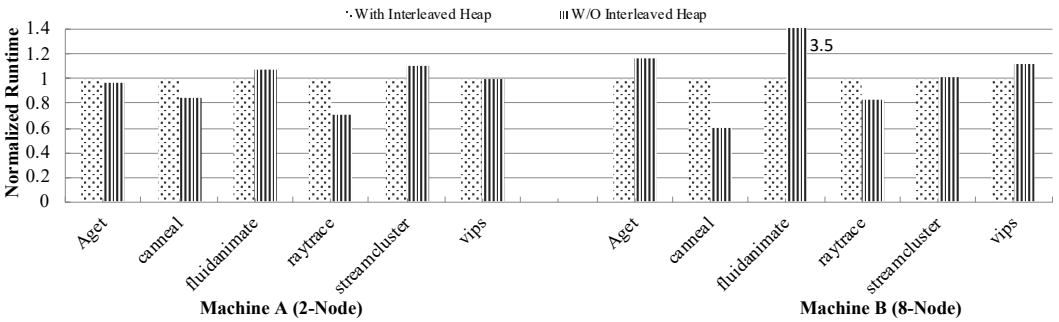

Fig. 7. Normalized runtime with and without interleaved heap on Machine B (8-Node).

4.5.2 Interleaved Heap. We also evaluate the performance difference when the support of interleaved heap is enabled or not. The performance data is shown in Fig. 7. Note that we have evaluated all real applications listed in Section 4.2. The applications that have no or little performance impact by the interleaved heap are omitted in this figure.

From Fig. 7, we have the following conclusion: the interleaved heap will benefit (or at least has no harmful impact on) the performance for most applications. Some applications, such as `fluidanimate`, will have the performance speedup of 3.5× with the interleaved heap. Therefore, the interleaved heap can be enabled by default, unless programmers know that it will not benefit the performance. A simple metric is to use the portion of the serial phase inside a multithreaded applications. Applications having a large portion of time spent in the serial phase, such as `canneal` and

raytrace, may not have good performance with the interleaved heap support. With the interleaved heap, NUMAlloc allocates the memory from all nodes interleavedly, instead of from the local node (based on the default first-touch policy). However, private objects that are allocated in a remote node may introduce unnecessary overhead due to remote accesses. That is, the interleaved heap will have a harmful impact on the serial execution, but may benefit the parallel execution because of its load balance. Therefore, programmers may turn off the interleaved heap for applications that are mostly running in serial phases. It is easy to turn on/off the interleved heap via a compilation flag.

4.5.3 Huge Page Support. Since Machine B utilizes transparent huge pages by default, we evaluate the performance impact of huge page support on Machine A, the 2-node machine. We only utilize PARSEC applications for this evaluation.

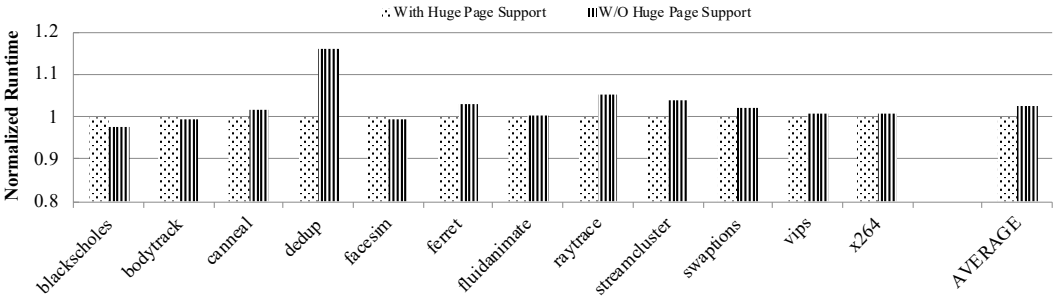

Fig. 8. Normalized runtime with and without huge page support on Machine A (2-node).

The results are shown in Fig. 8. When integrating with huge page support, NUMAlloc achieves a significantly better performance for dedup, where the performance difference is around 15%. On average, the huge page support improves the performance of all evaluated applications about 2.5%, and will not hurt the performance. This clearly indicates that it is beneficial to have explicit huge page support of NUMAlloc.

5 LIMITATION

This section describes some limitations of NUMAlloc.

The first limitation of NUMAlloc is that it consumes more memory than some popular allocators. Based on our analysis, the memory consumption is caused by NUMAlloc's bag mechanism. Currently, NUMAlloc employs one MB as a bag, which indicates that objects of each size will occupy at least of 1 MB, when transparent huge page support is enabled. NUMAlloc's design achieves a fast lookup on the metadata, but will utilize more memory unfortunately. We will investigate whether reducing the size of a bag could help reduce the memory consumption in the future.

The second limitation is that NUMAlloc will crash less often by preallocating a huge chunk of memory from the OS in the beginning. If an invalid reference is landed within a pre-allocated range, a program will not crash, different from other allocators. Instead, NUMAlloc aims to achieve the high performance over the reliability. Therefore, we believe that this limitation is acceptable.

6 RELATED WORK

This section discusses some related work with NUMAlloc.

General Purpose Allocators: There exists a large number of allocators (Aigner et al. 2015, Berger et al. 2000, Evans 2011, Ghemawat and Menage 2007, Lea 1988), but they are not designed for

NUMA architecture. Based on the management of small objects, allocators can be further classified into multiple types, such as sequential, BiBOP, and region-based allocators (Gay and Aiken 1998, Novark and Berger 2010). Region-based allocators are suitable for special situations where all allocated objects within the same region can be deallocated at once (Gay and Aiken 1998). For sequential allocators, subsequent memory allocations are satisfied in the continuous memory area, such as the Linux allocator originated from dlmalloc (Lea 1988) and Windows allocator (Novark and Berger 2010). That is, objects with different sizes can be placed continuously. For BiBOP-style allocators, one or multiple continuous pages are treated as a "bag", holding objects with the same size class. Many performance-oriented allocators, such as TCMalloc (Ghemawat and Menage 2007), `jemalloc` (Evans 2011), Hord (Berger et al. 2000), Scalloc (Aigner et al. 2015), and most secure allocators, such as OpenBSD (Foundation 2012) and DieHarder (Novark and Berger 2010), belong to this type. `NUMAlloc` also belongs to BiBOP-style allocators, and proposes multiple special designs for the NUMA architecture.

NUMA-aware Allocators: TcMalloc-NUMA adds additional node-based freelists and free spans to store freed objects and pages belonging to the same node (Kaminski 2012), which is similar to `NUMAlloc`. It also invokes the `mbind` system call to bind physical memory allocations to the node that the current thread is running on, which is similar to Jarena (Yang et al. 2019). However, both of them does not support huge pages and interleaved heap, invokes too many `mbind` system calls, and does not handle the metadata's locality. nMART proposes a NUMA-aware memory allocation for soft real-time system (Kim 2013). It proposes node-oriented allocation policy to minimize the access latency, and ensures temporal and spatial guarantee for real-time systems. nMART requires the change of the underlying OS, which is different from `NUMAlloc`. nMART also has a different target as `NUMAlloc` that tries to meet the time requirement of real-time systems, and `NUMAlloc` focuses more on the performance.

NUMA-Aware Java Heap Management: Some approaches that focus on improving the performance for Java applications, but they are not general purpose memory allocators. Ogasawara et al. focus on finding the preferred node location for JAVA objects during the garbage collection and memory allocations (Ogasawara 2009), via thread stack, synchronization information, and object reference graph. Tikir et al. propose to employ hardware performance counters to collect the runtime information of Java applications, and then migrate an object to the closet node with most accesses (Tikir and Hollingsworth 2005). NumaGiC reduces remote accesses in garbage collection phases with a mostly-distributed design so that each GC thread will mostly collect memory references locally, and utilize a work-stealing mode only when no local references are available (Gidra et al. 2015).

Combination of Task Scheduling and Memory Management: Redline integrates task scheduling and memory management inside the OS level (Yang et al. 2008), in order to support interactive applications. Majo et al. proposes to consider both data locality and cache contention to achieve better performance for the NUMA applications (Majo and Gross 2011). Wagle observed that dynamic memory allocations, thread placement and scheduling, memory placement policies, OS configurations may help improve the query performance of in-memory databases (Wagle et al. 2015). Majo et al. proposes to set task-to-thread affinity, and pin threads to specific cores to achieve a better performance (Majo and Gross 2015). Diener proposes a new kernel framework to combine the task management and memory management together to achieve the better performance (Diener 2015). They inspire `NUMAlloc`'s binding-based memory management. But `NUMAlloc` is the first work that combine both together inside a memory allocator.

NUMA Libraries: Cantalupo et al. proposes multiple APIs that allow users to manage their memory in fine granularity by combining with multiple existing system calls (Cantalupo et al. 2015). However, they are not targeting for a general purpose allocator, since it requires programmers to manage the memory explicitly. Majo et al. proposes multiple source-code based algorithmic changes in order to improve data sharing and memory access patterns for NUMA architectures (Majo and Gross 2013). Williams et al. propose to group data structures that can be migrated together with arenas (Williams et al. 2018). Shoal also proposes a set of APIs that allow user to specify memory access patterns (Kaestle et al. 2015). Then it could automatically replicate or distribute the memory to different NUMA nodes automatically. But both of them need significant manual effort to employ this.

Reactive Systems for NUMA Architecture: Some systems migrate tasks or physical pages reactively based on memory access patterns or other hardware characteristics (Blagodurov et al. 2011, Corbet 2012, Dashti et al. 2013, Lepers et al. 2015). They improve the performance automatically without human involvement. However, these systems impose additional overhead (sometimes unnecessarily) by migrating tasks or physical pages reactively. Further, a reactive approach cannot reduce remote accesses by the migration, if objects of the same page are concurrently accessed by multiple threads in multiple nodes (Gaud et al. 2014). `NUMAlloc` belongs to a proactive approach that does not require explicit and page migration, which is complementary to reactive systems. It is possible to combine these two together to achieve a better performance.

7 CONCLUSION

`NUMAlloc` is a drop-in memory allocator replacement that is specially designed for the NUMA architecture. Applications can be linked to `NUMAlloc` directly, without the change of code and the recompilation. `NUMAlloc` is different from existing memory allocators, since it proposes fine-grained memory management to accommodate the heterogeneity of both modern hardware and applications. It further proposes binding-based memory management and an interleaved heap to ensure locality and load balance. Based on our extensive evaluation, `NUMAlloc` achieves a significantly better performance than other popular allocators on the NUMA architecture, up to 5.8× faster than the default allocator. `NUMAlloc` is available at https://github.com/availableuponacceptance.

REFERENCES

Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2015. Fast, Multicore-scalable, Low-fragmentation Memory Allocation Through Large Virtual Memory and Global Data Structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, USA) *(OOPSLA 2015)*. ACM, New York, NY, USA, 451–469. https://doi.org/10.1145/2814270.2814294

Emery D. Berger, Kathryn S. McKinley, Robert D. Blumofe, and Paul R. Wilson. 2000. Hoard: a scalable memory allocator for multithreaded applications. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems* (Cambridge, Massachusetts, United States). ACM Press, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

Christian Bienia and Kai Li. 2009. PARSEC 2.0: A New Benchmark Suite for Chip-Multiprocessors. In *Proceedings of the 5th Annual Workshop on Modeling, Benchmarking and Simulation*.

Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. 2011. A Case for NUMA-aware Contention Management on Multicore Systems. In *Proceedings of the*

2011 USENIX Conference on USENIX Annual Technical Conference (Portland, OR) *(USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=2002181.2002182

Christopher Cantalupo, Vishwanath Venkatesan, Jeff Hammond, Krzysztof Czurlyo, and Simon David Hammond. 2015. *memkind: An Extensible Heap Memory Manager for Heterogeneous Memory Platforms and Mixed Memory Policies.* Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States).

HyperTransport Consortium. 2019. HyperTransport. https://en.wikipedia.org/wiki/HyperTransport.

Jonathan Corbet. 2012. AutoNUMA: the other approach to NUMA scheduling. "https://lwn.net/Articles/488709/".

Mohammad Dashti, Alexandra Fedorova, Justin Funston, Fabien Gaud, Renaud Lachaize, Baptiste Lepers, Vivien Quema, and Mark Roth. 2013. Traffic Management: A Holistic Approach to Memory Placement on NUMA Systems. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems* (Houston, Texas, USA) *(ASPLOS '13)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

SQL Developers. 2019. How SQLite Is Tested. "https://www.sqlite.org/testing.html".

Matthias Diener. 2015. Automatic task and data mapping in shared memory architectures. (2015).

Matthias Diener, Eduardo HM Cruz, and Philippe OA Navaux. 2015. Locality vs. Balance: Exploring data mapping policies on NUMA systems. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 9–16.

Jason Evans. 2011. Scalable memory allocation using jemalloc. "https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/".

OpenBSD Foundation. 2012. "OpenBSD". "https://www.openbsd.org".

The Apache Software Foundation. 2020. ab - Apache HTTP server benchmarking tool. "https://httpd.apache.org/docs/2.4/programs/ab.html".

Fabien Gaud, Baptiste Lepers, Jeremie Decouchant, Justin Funston, Alexandra Fedorova, and Vivien Quéma. 2014. Large Pages May Be Harmful on NUMA Systems. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference* (Philadelphia, PA) *(USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 231–242. http://dl.acm.org/citation.cfm?id=2643634.2643659

David Gay and Alexander Aiken. 1998. Memory Management with Explicit Regions. In *Proceedings of the ACM SIGPLAN '98 Conference on Programming Language Design and Implementation (PLDI), Montreal, Canada, June 17-19, 1998*. 313–323. https://doi.org/10.1145/277650.277748

Sanjay Ghemawat and Paul Menage. 2007. "TCMalloc : Thread-Caching Malloc". "http://goog-perftools.sourceforge.net/doc/tcmalloc.html".

Lokesh Gidra, Gaël Thomas, Julien Sopena, Marc Shapiro, and Nhan Nguyen. 2015. NumaGiC: A Garbage Collector for Big Data on Big NUMA Machines. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems* (Istanbul, Turkey) *(ASPLOS '15)*. ACM, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

Mel Gorman. 2010. Huge pages. "https://lwn.net/Articles/374424/".

Intel. 2009. An Introduction to the Intel QuickPath Interconnect. http://www.intel.com/content/dam/doc/white-paper/quick-path-interconnect-introduction-paper.pdf.

Stefan Kaestle, Reto Achermann, Timothy Roscoe, and Tim Harris. 2015. Shoal: Smart Allocation and Replication of Memory for Parallel Programs. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 263–276. http://dl.acm.org/citation.cfm?id=2813767.2813787

Patryk Kaminski. 2012. "NUMA aware heap memory manager". http://amd-dev.wpengine.netdna-cdn.com/wordpress/media/2012/10/NUMA_aware_heap_memory_manager_article_final.pdf.

Seyeon Kim. 2013. *Node-oriented dynamic memory management for real-time systems on ccNUMA architecture systems*. Ph.D. Dissertation. University of York.

Renaud Lachaize, Baptiste Lepers, and Vivien Quéma. 2012. MemProf: A Memory Profiler for NUMA Multicore Systems. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference* (Boston, MA) *(USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=2342821.2342826

Christoph Lameter. 2013a. Numa (non-uniform memory access): An overview. *Queue* 11, 7 (2013), 40–51.

Christoph Lameter. 2013b. NUMA (Non-Uniform Memory Access): An Overview. *Queue* 11, 7, Article 40 (July 2013), 12 pages. https://doi.org/10.1145/2508834.2513149

Per-Åke Larson and Murali Krishnan. 1998. Memory Allocation for Long-Running Server Applications. *SIGPLAN Not.* 34, 3 (Oct. 1998), 176âĂŞ185. https://doi.org/10.1145/301589.286880

Doug Lea. 1988. The GNU C Library. "http://www.gnu.org/software/libc/libc.html".

Baptiste Lepers, Vivien Quéma, and Alexandra Fedorova. 2015. Thread and Memory Placement on NUMA Systems: Asymmetry Matters. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference* (Santa Clara, CA) *(USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 277–289. http://dl.acm.org/citation.cfm?id=2813767.2813788

Xu Liu and John Mellor-Crummey. 2014. A Tool to Analyze the Performance of Multithreaded Programs on NUMA Architectures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Orlando, Florida, USA) *(PPoPP âĂŹ14)*. Association for Computing Machinery, New York, NY, USA, 259âĂŞ272. https://doi.org/10.1145/2555243.2555271

Martin Maas, David G. Andersen, Michael Isard, Mohammad Mahdi Javanmard, Kathryn S. McKinley, and Colin Raffel. 2020. Learning-based Memory Allocation for C++ Server Workloads. In *ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]*. 541–556. https://doi.org/10.1145/3373376.3378525

Zoltan Majo and Thomas R. Gross. 2011. Memory Management in NUMA Multicore Systems: Trapped Between Cache Contention and Interconnect Overhead. In *Proceedings of the International Symposium on Memory Management* (San Jose, California, USA) *(ISMM '11)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1993478.1993481

Z. Majo and T. R. Gross. 2013. (Mis)understanding the NUMA memory system performance of multithreaded workloads. In *2013 IEEE International Symposium on Workload Characterization (IISWC)*. 11–22. https://doi.org/10.1109/IISWC.2013.6704666

Zoltan Majo and Thomas R. Gross. 2015. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. In *Proceedings of the 20th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (San Francisco, CA, USA) *(PPoPP 2015)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/2688500.2688509

Gene Novark and Emery D. Berger. 2010. DieHarder: securing the heap. In *Proceedings of the 17th ACM conference on Computer and communications security* (Chicago, Illinois, USA) *(CCS '10)*. ACM, New York, NY, USA, 573–584. https://doi.org/10.1145/1866307.1866371

Takeshi Ogasawara. 2009. NUMA-aware Memory Manager with Dominant-thread-based Copying GC. In *Proceedings of the 24th ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications* (Orlando, Florida, USA) *(OOPSLA '09)*. ACM, New York, NY, USA, 377–390. https://doi.org/10.1145/1640089.1640117

Ashish Panwar, Sorav Bansal, and K. Gopinath. 2019. HawkEye: Efficient Fine-grained OS Support for Huge Pages. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*. 347–360. https://doi.org/10.1145/3297858.3304064

Sean Reifschneider. 2013. "Pure python memcached client". "https://pypi.python.org/pypi/python-memcached".

Kirill Rogozhin. 2014. Controlling memory consumption with IntelÂő Threading Building Blocks (IntelÂő TBB) scalable allocator. "https://software.intel.com/content/www/us/en/develop/articles/controlling-memory-consumption-with-intel-threading-building-blocks-intel-tbb-scalable.html".

William N. Sumner, Yunhui Zheng, Dasarath Weeratunge, and Xiangyu Zhang. 2010. Precise calling context encoding. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*. 525–534. https://doi.org/10.1145/1806799.1806875

M. M. Tikir and J. K. Hollingsworth. 2005. NUMA-Aware Java Heaps for Server Applications. In *19th IEEE International Parallel and Distributed Processing Symposium*. 108b–108b. https://doi.org/10.1109/IPDPS.2005.299

Mehul Wagle, Daniel Booss, Ivan Schreter, and Daniel Egenolf. 2015. NUMA-aware memory management with in-memory databases. In *Technology Conference on Performance Evaluation and Benchmarking*. Springer, 45–60.

Sean Williams, Latchesar Ionkov, Michael Lang, and Jason Lee. 2018. Heterogeneous Memory and Arena-Based Heap Allocation. In *Proceedings of the Workshop on Memory Centric High Performance Computing, MCHPC@SC 2018, Dallas, TX, USA, November 11, 2018*. 67–71. https://doi.org/10.1145/3286475.3286568

Ting Yang, Tongping Liu, Emery D. Berger, Scott F. Kaplan, and J. Eliot B. Moss. 2008. Redline: first class support for interactivity in commodity operating systems. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation* (San Diego, California) *(OSDI'08)*. USENIX Association, Berkeley, CA, USA, 73–86. http://dl.acm.org/citation.cfm?id=1855741.1855747

Zhang Yang, Aiqing Zhang, and Zeyao Mo. 2019. JArena: Partitioned Shared Memory for NUMA-awareness in Multi-threaded Scientific Applications. *arXiv preprint arXiv:1902.07590* (2019).

Qiang Zeng, Junghwan Rhee, Hui Zhang, Nipun Arora, Guofei Jiang, and Peng Liu. 2014. DeltaPath: Precise and Scalable Calling Context Encoding. In *12th Annual IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2014, Orlando, FL, USA, February 15-19, 2014*. 109. https://dl.acm.org/citation.cfm?id=2544150