NUMAlloc: A Faster NUMA Memory Allocator

Hanmei Yang
University of Massachusetts Amherst
USA
hanmeiyang@umass.edu

Xin Zhao
University of Massachusetts Amherst
USA
zhao@umass.edu

Jin Zhou
University of Massachusetts Amherst
USA
jinzhou@umass.edu

Wei Wang
University of Texas at San Antonio
USA
wei.wang@utsa.edu

Sandip Kundu
University of Massachusetts Amherst
USA
kundu@umass.edu

Bo Wu
Colorado School of Mines
USA
bwu@mines.edu

Hui Guan
University of Massachusetts Amherst
USA
huiguan@umass.edu

Tongping Liu
University of Massachusetts Amherst
USA
tongping@umass.edu

Abstract

The NUMA architecture accommodates the hardware trend of an increasing number of CPU cores. It requires the cooperation of memory allocators to achieve good performance for multithreaded applications. Unfortunately, existing allocators do not support NUMA architecture well. This paper presents a novel memory allocator – `NUMAlloc`, that is designed for the NUMA architecture. `NUMAlloc` is centered on a binding-based memory management. On top of it, `NUMAlloc` proposes an "origin-aware memory management" to ensure the locality of memory allocations and deallocations, as well as a method called "incremental sharing" to balance the performance benefits and memory overhead of using transparent huge pages. According to our extensive evaluation, `NUMAlloc` has the best performance among all evaluated allocators, running 15.7% faster than the second-best allocator (mimalloc), and 20.9% faster than the default Linux allocator with reasonable memory overhead. `NUMAlloc` is also scalable to 128 threads and is ready for deployment.

CCS Concepts: • **Software and its engineering → Memory management**.

Keywords: Memory Allocation, NUMA Architecture

ACM Reference Format:
Hanmei Yang, Xin Zhao, Jin Zhou, Wei Wang, Sandip Kundu, Bo Wu, Hui Guan, and Tongping Liu. 2023. NUMAlloc: A Faster NUMA Memory Allocator. In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3591195.3595276

1 Introduction

Non-Uniform Memory Access (NUMA) is the de-facto design for modern many-core machines to address the scalability issues of increasing hardware cores. In NUMA architecture, each processor (or called node/socket interchangeably) has its own memory, allowing threads running on different nodes to access their own memory concurrently. Unfortunately, it is challenging to achieve the expected scalability. One notorious performance issue is caused by remote accesses that a task accesses the memory of a remote node (called remote memory), since a remote access has much higher latency than accessing the memory from the local node (or a local access) [10]. Although many profiling tools are proposed to identify NUMA issues of applications [29, 37, 43, 49, 58, 63, 64], they typically focus on issues inside a memory object, while still requiring memory allocators to ensure the locality of memory allocations/deallocations.

In order to reduce remote accesses, some NUMA-aware allocators [31, 35, 62] have been proposed in the past. Kaminski built the first NUMA-aware memory allocator on top of TCMalloc in 2008 [31], called TCMalloc-NUMA in the remainder of this paper, which has been integrated into the modern TCMalloc [34]. TCMalloc-NUMA adds a freelist and page-span for each NUMA node, and binds the physical memory to a physical node explicitly. TCMalloc-NUMA improves the locality by allocating objects from the per-node list/page-span that a thread is running on. mimalloc [41] proposes per-page (e.g., 64K) freelists that are typically allocated/used by a single thread, which increases the locality.

However, existing designs are not sufficient to eliminate remote accesses: (1) although in theory an allocator can always check a thread's physical node (e.g., via the system call) so that a thread only allocates the memory from its local node, that is unfortunately too expensive due to the high overhead of the system call (around 10,000 cycles for getting the node of the memory); (2) Some common practice of existing allocators also weakens the locality guarantee: each thread typically tracks objects deallocated by itself, and prefers the available memory in its local cache upon memory allocations. However, there is no guarantee that an object deallocated by a thread is originally from the thread's local physical node; (3) Threads can be migrated by the underlying operating system from one node to another, where the migration will turn all of a thread's local accesses to remote ones. (4) Existing allocators do not take advantage of the location relationship between threads and memory when handling memory allocations/deallocations, initializing metadata, and sharing huge pages.

To address these issues, we propose a novel **binding-based allocator**, called `NUMAlloc`. `NUMAlloc` performs thread-binding and memory-binding inside the allocator at the same time. The thread-binding provides the following benefits: (1) it enables `NUMAlloc` to obtain the origin (or the physical node) of threads with few instructions without using expensive system calls, as threads are always staying at the same node; (2) Thread binding eliminates remote accesses caused by the above-mentioned thread migrations. Note that *`NUMAlloc`'s thread-binding binds a thread to a specific node, instead of a particular core*. It does not exclude OS-based scheduling, where the OS could still schedule based on the changed resource. By default, `NUMAlloc` takes the node-interleaved binding that binds continuous threads to different nodes in an interleaved way so that every node will have a similar number of threads. Given such a node balanced binding, we argue that `NUMAlloc` can be also employed in the server environment with thread-pool design. Further, `NUMAlloc` could also support node-saturate binding or any explicit binding provided by users, allowing users to provide more control. The combination of thread-binding and memory-binding inside the allocator provides a clear relationship between threads and memory. Therefore, it enables more advanced memory management discussed as follows, addressing the other two above-mentioned issues.

Based on memory and thread binding, `NUMAlloc` ensures the full locality of memory allocations with its **origin-aware memory management**. `NUMAlloc` guarantees that each thread will always allocate the memory from the same physical node that the thread is running on. Since `NUMAlloc` binds the virtual memory to physical nodes explicitly and maintains the mapping relationship internally, it could always infer the origin of each object. During deallocations, `NUMAlloc` guarantees that a freed object will be always placed into a freelist with the same origin as the allocating thread. In particular, an object is returned to the deallocating thread *only if* the object is originated from the same node as the current thread; otherwise, it will be returned to its original node (e.g., per-node freelist). In addition, `NUMAlloc` ensures that all heap metadata exists on the same local node, such as the metadata of tracking the size of objects.

Based on thread-binding, `NUMAlloc` proposes a new **incremental sharing** mechanism to take advantage of the Transparent Huge Pages (THP) of modern hardware [13]. Huge pages are expected to significantly reduce Translation Lookaside Buffer (TLB) misses, as each page table entry covers a larger range of virtual addresses (e.g., 2MB instead of 4KB). However, most existing allocators [3, 8, 40] do not support huge pages, or even require to disable huge pages [4]. Allocators supporting huge pages have their own shortcomings: LLAMA [45] allocates objects from huge pages based on the liveness of objects, but requires expensive analysis and profiling; TEMERAIRE [28] does not share huge pages between different threads, since mistakenly letting two remote threads share the same huge page may impose some performance degradation. In contrast, `NUMAlloc`'s thread binding makes it possible to share the huge pages between threads running on the same node. We call it "incremental sharing" as each thread only fetches the amount of memory it needs from the current page at a time, instead of the entire page. Therefore, `NUMAlloc` combines the best of both worlds that it takes the performance advantage of huge pages but does not deteriorate the memory consumption.

We have performed an extensive evaluation on synthetic and real applications, with 25 applications in total. We compared `NUMAlloc` with popular allocators, such as the default Linux allocator, TCMalloc [24], jemalloc [20], Intel TBB [53], Scalloc [3], and mimalloc [41]. `NUMAlloc` is running around 16% faster than the second-best allocator (mimalloc), achieving around 21% speedup comparing to the default Linux allocator. For the best case, `NUMAlloc` runs up to 5.3× and 4.6× faster than the default allocator and mimalloc. At the same time, `NUMAlloc`'s memory consumption is comparable to industrial-level allocators, such as TCMalloc, jemalloc, and mimalloc. `NUMAlloc` is much more scalable than all other allocators based on our evaluation. `NUMAlloc` shows promising potential for production deployment, due to its high performance and good scalability. Overall, this paper makes the following contributions:

- It proposes the first **binding-based memory management** to support the NUMA architecture.
- It proposes an **origin-aware memory management** to ensure the full locality of memory allocations.
- It proposes an **incremental sharing** to achieve a better balance between the performance and memory consumption for huge pages.
- Experimental results show that applications with `NUMAlloc` achieves better performance and scalability than all widely-used commercial allocators.

2 Background

This section discusses the NUMA architecture, existing OS support and common design of memory allocators.

2.1 NUMA Architecture and OS Support

The Non-Uniform Memory Access (NUMA) architecture is designed to solve the scalability issue, due to its decentralized nature. Instead of making all cores wait 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 a node). Due to multiple memory controllers, the contention for the memory controller is largely reduced and therefore the scalability can be improved correspondingly. However, applications running on the NUMA architecture may suffer from remote accesses [10], where a thread accesses the memory in a remote node. Further, when multiple threads are accessing the memory in the same node, it may cause interconnect congestion and node imbalance [10].

Operating Systems already support the NUMA architecture, especially on task scheduling and physical memory allocation. For task scheduling support, the OS provides some system calls that allow users to bind a task to a specific node. For memory allocation, the OS provides system calls (e.g., `mbind`) to change memory allocation policy for a range of memory or for the whole process [18, 38]. However, those system calls still require programmers to specify the policy explicitly, which cannot, therefore, automatically provide the performance improvements. `NUMAlloc` relies on these existing system calls to manage memory allocations, as further described in Section 3, but without the need of changing user programs explicitly. Similar to `NUMAlloc`, the NUMA programming library `libnuma` [6] also utilizes these system calls to offer a user-friendly interface for data placement and thread binding policies. `NUMAlloc` differs from it in pioneering adoption of these ideas inside the allocator and we will see the benefits of it in Section 4. Linux also supports Automatic NUMA balancing (AutoNUMA) [26], which characterizes the memory accesses of each thread and migrates the threads or memory pages to improve the memory access locality. However, AutoNUMA may degrade the performance due to its unmap of the memory frequently [7], which cannot replace the NUMA-aware memory allocator.

2.2 Common Designs of Memory Allocators

Memory allocators share some common designs. First, most allocators manage objects differently based on the size of objects. For big objects, allocators may request objects from the OS directly and return them to the OS directly upon deallocation [8]. On the other hand, small objects will be tracked in freelists based on size classes. Managing small objects can be further classified into multiple categories, such as sequential, BiBOP and region-based, where region-based allocators do not belong to general-purpose allocators [23, 50]. For sequential allocators, subsequent memory allocations are satisfied in a continuous memory block [5]. BiBOP-style allocators, which stands for "**Bi**g-**B**ag-**o**f-**P**ages" [27], utilize one or multiple continuous pages as a "bag" that holds objects of the same size class. Currently many performance-oriented allocators [3, 20, 24], and most secure allocators [21, 50, 55, 56], belong to this category. Second, to support multithreaded applications, modern allocators (e.g., TCMalloc) often implement the "per-thread heap" that tracks deallocated objects from the current thread [31], where allocating objects from per-thread heaps do not need to acquire locks. Therefore, the per-thread heap is expected to reduce the contention between threads. When the number of objects or the size of freed objects of a per-thread heap is larger than a threshold, it will return objects to a common heap shared by multiple threads. `NUMAlloc` borrows these common designs in its implementation.

3 Design and Implementation

`NUMAlloc` is designed as a replacement memory allocator. It intercepts all memory allocation/deallocation invocations via the preloading mechanism, and redirects them to `NUMAlloc`'s implementation. Therefore, there is no need to change the source code of applications, and there is no need to use a custom OS or hardware. In the following, we first discuss `NUMAlloc`'s basic heap layout, and then discuss multiple components that separate it from existing allocators.

3.1 Basic Heap Layout

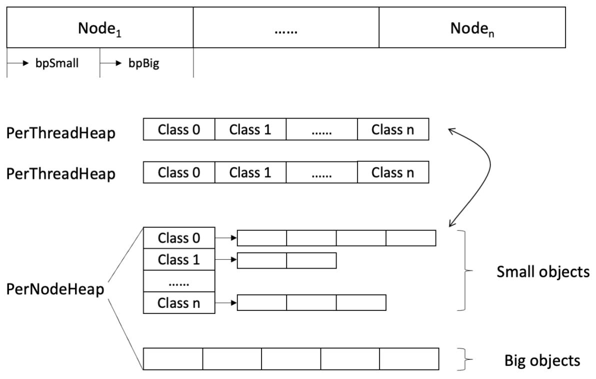


Figure 1. Overview of `NUMAlloc`'s heap layout.

`NUMAlloc`'s heap layout is designed as Figure 1. Initially, `NUMAlloc` requests a large and continuous block of memory from the underlying OS, and then divides it evenly into multiple regions based on the number of hardware nodes. Each region is bound to a different physical node via `mbind` system call. In particular, the first region is bound to the first node, the second one is bound to the second node, and so on. This design enables us to compute the physical node quickly from a memory address: we could compute the index of the physical node by dividing the heap offset by the region size. Each node's memory region will be further divided into two sub-regions, one for small objects, and the other one for big objects. The `bpSmall` pointer is utilized to track never-allocated memory for small objects, and the `bpBig` pointer tracks the position of big objects. Similar to existing allocators, `NUMAlloc` manages small and big objects differently. For small objects (< 512KB), each request will be satisfied from a particular size class and `NUMAlloc` utilizes the well-known BiBOP style that all objects in the same bag (32KB by default) will have the same size class. Big object allocation will be satisfied in a sequential manner and their sizes are aligned to the size of one bag.

To support the NUMA architecture, a per-node heap (PerNodeHeap in Figure 1) is proposed that has one freelist for each size class and one common freelist for all big objects from the current node. To reduce the contention, `NUMAlloc` adopts a per-thread heap (PerThreadHeap in Figure 1) that maintains a freelist for each size class, which requires no lock protection since each thread has its own per-thread heap. However, this may introduce memory blowup [8] that freed objects of a per-thread heap cannot be utilized for future allocations from other threads, which will be addressed in Section 3.5. `NUMAlloc` tracks the small objects' size information in a separate area called PerBagSizeInfo, while the big objects utilize a linked list called PerBigObjectSizeInfo to store the size and availability information, which allows coalescing multiple continuous big objects into a bigger one upon deallocations.

Overall, `NUMAlloc` includes a novel layout that can quickly compute the physical node (with the memory binding) and a per-node heap to support node-aware allocations. This design allows it to perform origin-aware memory management and incremental sharing efficiently, as discussed in the following sections.

3.2 Binding-Based Memory Management

As described in Section 1, thread migration will cause multiple performance issues for the NUMA architecture. Therefore, `NUMAlloc` binds each thread to a node specifically to avoid thread migration across different nodes. `NUMAlloc` currently supports two types of binding, node-interleaved binding and node-saturate binding. Node-interleaved binding binds continuous threads to different nodes in an interleaved way so that every node will have a similar number of threads. That is, the first thread will be bound to the node that it is scheduled to run by the OS, and the second thread will be bound to its next node, and so on. Instead, the node-saturate binding will bind sufficient threads to a node first before binding to a different node. For node-saturate binding, the threads to be assigned will be the same as the number of hardware cores. `NUMAlloc` uses node-interleaved binding by default but users can switch to the node-saturate binding by controlling the environment variable. As evaluated in Section 4, even a simple binding policy like node-interleaved binding can provide significant performance improvement for most applications. Further, users can provide their customized binding via a configuration file to fit the workload.

Note that `NUMAlloc` only binds a thread to a node, instead of a core, which still allows the scheduling initiated by the OS. To perform the binding correctly, `NUMAlloc` obtains the hardware topology in the initialization phase via the `numa_node_to_cpus` API, which tells the relationship between each CPU core and each node. Then it intercepts all thread creations in order to bind a newly-created thread to a specific node.

In addition to thread binding, `NUMAlloc` also includes memory binding, which binds memory regions to each NUMA node, as discussed in Section 3.1. With both bindings, `NUMAlloc` can quickly obtain the physical node where a heap object is located, as well as the node where the thread is running, without the need for expensive system calls, which makes more advanced memory management possible. In summary, `NUMAlloc`'s memory management is based on bindings and we are the first to show how much performance improvement can be obtained if bindings are considered within the allocator.

3.3 Origin-Aware Memory Management

As described in Section 3.1, `NUMAlloc` includes an origin-computable design that could quickly determine the origin of each heap object via the computation. On top of it, `NUMAlloc` proposes an origin-aware deallocation that will always return a freed object to a freelist with the same origin. In particular, if a freed object originated from a different node, it is returned to its original node's freelist. Otherwise, a small object is returned to the current thread's freelist and a big object is returned to the current node's freelist. Compared to node-based freelists, there is no need to acquire a lock when operating on the per-thread freelist. Different from all existing work, `NUMAlloc` may return a freed object into the per-thread list or its original node's freelist, instead of simply putting it into the per-thread list. That is, `NUMAlloc` considers the origination of objects for deallocations.

`NUMAlloc` also ensures node-local memory allocations. For small objects, the allocation follows this order: (1) The per-thread's freelist will be checked first, since there is no need to acquire any lock and objects may be still hit in the cache (as they are just accessed by the thread). (2) If the per-thread freelist does not have available objects, `NUMAlloc` tries to allocate from the current node's freelist. As mentioned above, a node's freelist holds objects originating from this node. (3) If the previous two steps fail, we will allocate the memory from the current node's un-allocated region, as shown by *bpSmall* in Figure 1. For big objects, allocation will be satisfied from per-node freelists or un-allocated region (pointed by *bpBig* pointer in Figure 1) of the current node, indicating the allocation locality.

3.4 Incremental Sharing

When Transparent Huge Page (THP) is enabled, the OS prefers to allocate huge pages if a program touches a continuous memory region with a size larger than a huge page (e.g., 2MB). Since `NUMAlloc` allocates a large region initially (as shown in Figure 1), huge pages will be employed by the OS correspondingly. However, it is important to reduce memory fragmentation, as one allocation from a memory block will be assigned to a huge page. `NUMAlloc` makes multiple threads (from the same node) share the same huge page, instead of having a separate superblock for each thread as Scalloc [3] and TEMERAIRE [28]. That is, when a thread is running out of memory, it obtains only multiple objects at a time (currently 32KB for one bag) from the corresponding memory block, instead of getting few megabytes for each per-thread heap. For small objects larger than 32KB (but less than 512KB), each thread will get only one object at a time, by aligning to 32KB as well. This is why it is called "incremental sharing". `NUMAlloc` allows objects with different size classes to share the same huge page, to further reduce the memory fragmentation.

In the evaluation, we observe that `NUMAlloc` actually will utilize huge pages for metadata, which may introduce unnecessary memory overhead since it only needs 8 bytes for "PerBagSizeInfo" used internally by `NUMAlloc`. To get rid of this overhead, we leverage the `madvise` system call to make the metadata memory allocate from normal pages. These are the basic reasons that `NUMAlloc` has much less memory consumption than Scalloc, as evaluated in Section 4.2.

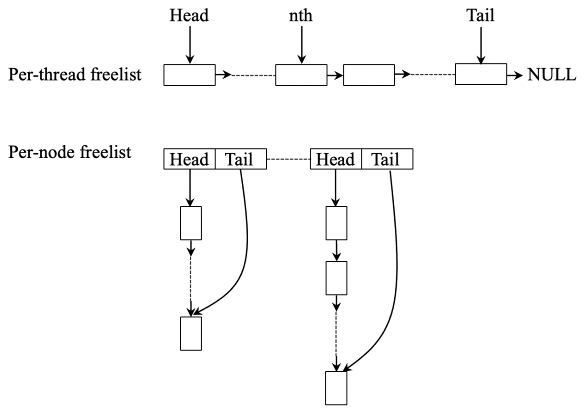


Figure 2. Per-thread freelist and per-node freelist design to achieve efficient object movement.

3.5 Efficient Object Movement

`NUMAlloc` requires moving freed objects between per-thread and per-node freelists frequently to reduce memory blowup. On the one hand, when a per-thread freelist has too many freed objects, some of them should be moved to the per-node freelist so that other threads could re-utilize these freed objects. On the other hand, each per-thread freelist needs to obtain freed objects from its per-node heap, when a thread is running out of memory. Therefore, an efficient mechanism is required to support frequent movement. Existing allocators, such as TCMalloc [24], traverse the freelist to collect a specified number of objects, and then move all of them at a time, which unfortunately has the following issues: (1) traversing a freelist will bring some unnecessary data to the cache when an allocator is reusing freed objects from the freelist. This loading wastes the cache resources and evicts cache lines with useful content in the future. (2) Existing allocators typically move recently-freed objects (and hot in cache) to other freelist, which is not good for performance. (3) The traversal of the shared list (per-node heap) may introduce significant lock contention, if multiple threads are waiting to fetch objects from the shared list.

`NUMAlloc` proposes an efficient mechanism with the following data structures to avoid these issues. First, each per-thread freelist maintains two pointers that point to the least recently used objects (shown as the `Tail` pointer) and the nth pointer (counted from the tail, shown as the `nth` pointer) in the upper part of Figure 2. This structure avoids the traverse of freelist during the movement, and allows the movement of the least recently used objects (between $(n{+}1)th$ and $Tail$) to the per-node freelist. After the movement, the `Tail` pointer will be set to the original nth object. Second, `NUMAlloc` also proposes a circular array shown in the bottom part of Figure 2 that helps move objects from the shared per-node freelist to per-thread freelists. Each per-node freelist actually consists of many sub-lists, where a `Head` pointer and a `Tail` pointer point to the header and the tail of each sub-list. When a thread is moving multiple objects from the per-node freelist, it will move all objects in a sub-list (pointed by a pair of `Head` and `Tail` pointers) at a time. Therefore, there is no need to traverse the whole list to obtain these objects for the movement, which reduces the contention.

4 Experimental Evaluation

This section aims to answer the following questions:

- **Performance:** How is `NUMAlloc`'s performance, comparing to existing general allocators and NUMA-aware allocators? (Section 4.1)
- **Memory Consumption:** What is the memory consumption of `NUMAlloc`? (Section 4.2)
- **Scalability:** How is the scalability of `NUMAlloc`? (Section 4.3)
- **Impact of Design Choices:** What is the impact of each design choice on the performance of `NUMAlloc`? (Section 4.4)

Experimental Setup: NUMAlloc was evaluated on an Intel Xeon(R) Platinum 8153 machine with 8 nodes, where each node has 16 cores. 8 nodes are divided into two groups, where the four nodes of each group are fully connected, and there are four links between the two groups. Any two nodes are less than or equal to 2 hops, where the latency of one hop and two hops is 2.1× and 3.1× of local accesses, respectively. The machine is installed with 512GB memory. The underlying OS is Linux Debian 10 and the compiler is GCC-8.3.0. In the evaluation, transparent huge page, AutoNUMA and hyperthreading are enabled unless otherwise specified.

Compared Allocators: We compare NUMAlloc with the default Linux allocator (Glibc-2.28) [44], TCMalloc [34], TCMalloc-NUMA [31], jemalloc-5.2.1 [20], Intel TBB-2021.5 [32], Scalloc-1.0.0 [3], and mimalloc-1.6.7 [41]. Note that we are comparing against TCMalloc's and TBB's NUMA awareness version. The evaluated TCMalloc already includes TEMERAIRE [28]'s huge page support. We do not include Hoard [8] as it is not the state-of-art anymore [3, 41].

Evaluated Applications: We evaluated the PARSEC applications [9], five OpenMP/MPI applications AMG, LAMMPS, Nekbone, QMCPACK, and Quicksilver from CORAL-2 Benchmarks [1], and real applications including Aget, Apache httpd-2.4.35, Memcached-1.4.25, MySQL-5.7.15, Pbzip2, Pfscan and SQLite-3.12.0. PARSEC applications are using native inputs [9]. We utilize 128 threads, which is the same as the number of cores of the evaluated machine. For Apache, we use the ab script to send 1,000,000 requests [22]. For MySQL, we use sysbench with 128 threads separately, each issuing 100,000 requests. For Memcached, the python-memcached is used to evaluate it with 3000 loops to get the sufficient runtime [52]. 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 [16]. For OpenMP/MPI applications, we use a hybrid MPI + OpenMP mode with one MPI process per node and 16 OpenMP threads per process.

4.1 Performance Evaluation

Figure 3 shows the performance of PARSEC, OpenMP/MPI and real applications with different allocators (separated by one empty column). The runtime of each allocator is normalized to that of Linux's default one. Overall, NUMAlloc has the best performance among these allocators. In particular, NUMAlloc is 15.7% faster than the second-best allocator (mimalloc) and 20.9% faster than the default Linux allocator. For the best case (e.g., fluidanimate), NUMAlloc is running up to 5.3× faster than the default Linux allocator, and 4.6× faster than mimalloc. On average, NUMAlloc is 18.4%, 18.2%, and 20.7% faster than TCMalloc [34], TCMalloc-NUMA [31], Intel TBB [33], all of which are NUMA-aware allocators. Considering only real applications, NUMAlloc outperforms both the default Linux allocator and mimalloc by an average of 8.6% and 7.4%, respectively. We further evaluated a larger scale version of memcached and NUMAlloc still outperforms other allocators.

As shown in Figure 3, NUMAlloc has a significant performance improvement (over 25%) in the following applications, including dedup, fluidanimate, streamcluster, swaptions, pbzip2, AMG, LAMMPS, and Quicksilver. We further examine the number of remote accesses and TLB misses to confirm whether NUMAlloc significantly reduces them for these applications. The results are shown in Figure 4, which includes the runtime performance (black line), remote accesses (blue line), and TLB misses plus remote accesses (red line) together for a better comparison. Overall, NUMAlloc significantly reduces the number of remote accesses for the evaluated applications. In particular, NUMAlloc has 9× fewer remote accesses than the default Linux allocator and 8× fewer than the second-best allocator (mimalloc) on average. For TLB misses, NUMAlloc reduces it by 18× compared to the default Linux allocator. We also notice that, as expected, TCMalloc performs best among allocators other than NUMAlloc with a low number of TLB misses (about 1.7× of NUMAlloc) due to its support for huge pages. As can be seen from Figure 4, NUMAlloc greatly reduces the number of remote accesses for five applications, fluidanimate, Pbzip2, streamcluster, LAMMPS, and AMG. Let us utilize fluidanimate as an example, where NUMAlloc is running 4.8× faster than TBB and 5.3× faster than the default Linux allocator. Figure 4 shows that TBB and the default Linux allocator have 5.9× and 6.2× more remote accesses than NUMAlloc, which explains why NUMAlloc is the fastest on this application. NUMAlloc's big reduction of remote accesses can be attributed to the following factors: its thread binding avoids unnecessary remote accesses; its metadata is placed on the local node, based on the binding design; its origin-aware memory allocation ensures locality of memory allocations.

However, for some applications, there is not much difference in the number of remote accesses compared to some allocators. Based on our investigation, NUMAlloc is running faster than others due to the reduction of TLB misses instead. Taking dedup as an example, compared to TCMalloc-NUMA, NUMAlloc generates 1.8% more remote accesses, but it has more than 21× fewer TLB misses, resulting in better performance. This is also true for swaptions compared with Scalloc and mimalloc. From Figure 4, we notice that mimalloc has a surprisingly low number of remote accesses on swaptions compared to all other allocators. We currently do not know the exact reason for this. Nevertheless, NUMAlloc ended up performing slightly better due to 1.57× fewer TLB misses. For Quicksilver, although NUMAlloc reduces the remote accesses and TLB misses by 1.62× and 1.13× compared to the second-best allocator (mimalloc), the performance is slightly worse. This is because other factors (e.g. cache misses) dominate the performance. But this is the only exception across all evaluations. In most cases, fewer remote

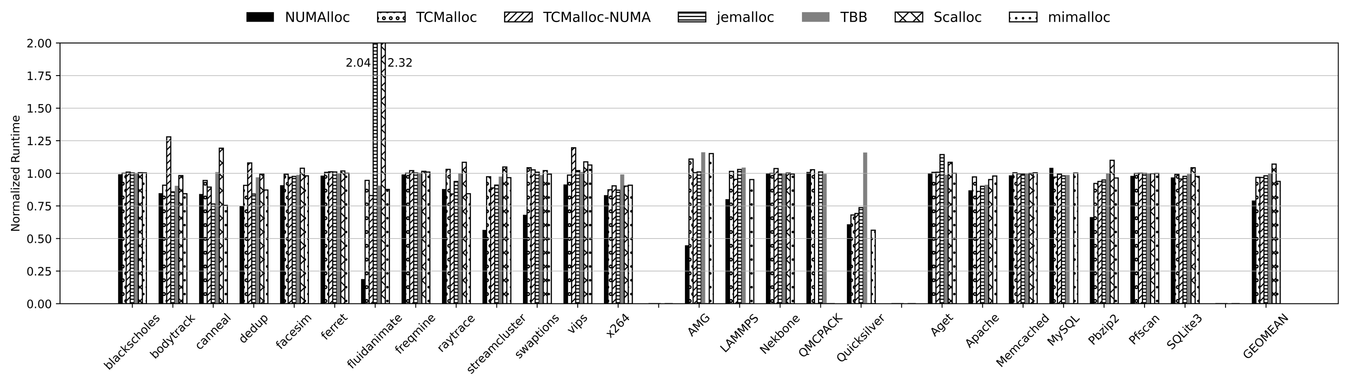


Figure 3. Performance of different allocators on PARSEC, OpenMP/MPI and real applications, where all data are normalized to that of the default Linux allocator. Here, a lower bar indicates a better performance. Note that some applications failed to run with some allocators (mainly due to OOM).

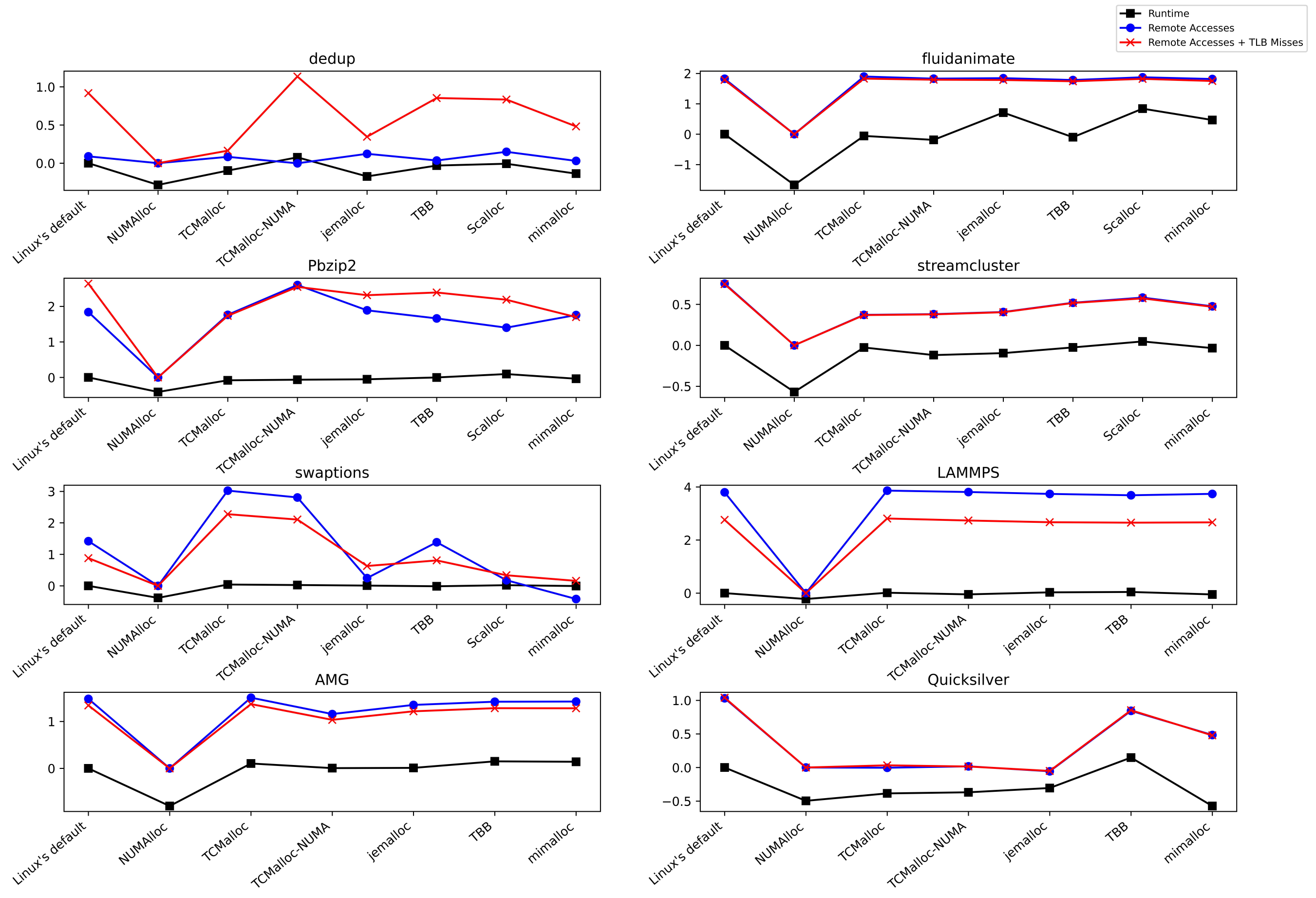


Figure 4. Normalized runtime, remote accesses, and TLB misses of different allocators, where the remote accesses and TLB misses data are normalized to that of `NUMAlloc` and runtime data is normalized to that of the default Linux allocator for better comparison of line trend. All data are finally log-transformed to limit the range of data distribution. Here, a lower number indicates a better performance.

accesses and TLB misses indicate better performance. If we look at remote accesses plus TLB misses data (red line) in

Figure 4, NUMAlloc always performs the best among all evaluated allocators. Meanwhile, we observe that the remote accesses plus TLB misses (red line) generally has a positive correlation with application performance (black line). To summarize, NUMAlloc has fewer remote accesses and fewer TLB misses, which are the reasons that it outperforms other allocators.

4.2 Memory Consumption

We also measure the memory consumption of different allocators on PARSEC benchmark [9] and real applications, as shown in Table 1. Overall, the default Linux allocator has the smallest memory consumption, and Intel TBB is the second-best one. NUMAlloc's total memory consumption is around 17.6% more than that of the default Linux allocator, but it is similar to TCMalloc and 1.63× better than the second fastest allocator (mimalloc).

The memory consumption of NUMAlloc is almost 4.5× lower than that of Scalloc with huge page support. Similar to NUMAlloc, Scalloc allocates a big region of virtual memory from the underlying OS initially, which will be backed by huge pages physically. While the huge pages can improve the performance of an application, they can also lead to a significant increase in memory usage. For example, an application will use 2MB of physical memory even if it only allocates a small object (e.g., 8 bytes). Compared to Scalloc, NUMAlloc makes threads (with different size classes) running on the same node share the same huge page, as described in Section 3.4, which effectively reduces its memory consumption. That is why the total memory consumption of NUMAlloc is far better than Scalloc. Interestingly, we observe that the memory consumption of NUMAlloc is similar to that of TCMalloc (TEMERAIRE [28]), which is a state-of-the-art allocator optimized specifically for huge pages. We expect that NUMAlloc's memory consumption can be further reduced by utilizing some complicated mechanisms proposed by TEMERAIRE.

We further confirmed the memory consumption when transparent huge page support is disabled, which can be seen as the "w/o THP" column in Table 1. In this case, NUMAlloc's total memory overhead is actually comparable to the default Linux allocator and TBB, where the total memory consumption is decreased from 15938 MB to 13622 MB. That is, NUMAlloc imposes a low memory overhead when not using huge pages.

4.3 Scalability

To validate the scalability of NUMAlloc, we use four synthetic applications from Hoard [8], including threadtest, larson [39], cache-scratch and cache-slash, which is also employed by existing work [3]. We do not use the PARSEC applications, as they are not scalable by design. For instance, raytrace has no performance difference when running with 16 threads or 40 threads. In the evaluation, we maximize the number of threads on each node for NUMAlloc. For instance, 32 threads will use 2 nodes, as each node has 16 cores. For other allocators, we only specify the number of threads, and it is up to the OS to determine the scheduling.

Figure 5 demonstrates how the performance speedup of various allocators changes as the number of threads increases. All data are normalized to the runtime of Linux's default allocator under one thread. Overall, NUMAlloc has the best performance when the number of threads is 128. Its average speedup is 88×, compared to Linux's allocator with one thread, while the second-best allocator – mimalloc – has a 75× speedup. In contrast, the default Linux allocator only has a speedup of 49×. That is, NUMAlloc has the best scalability compared to other allocators.

Among these applications, cache-scratch and cache-thrash test false sharing issues that can be introduced by allocators, where multiple threads access different objects in the same cache line. When false sharing occurs, threads accessing seemingly unrelated data will invalidate each other when performing writes, resulting in performance degradation. cache-scratch tests passive false sharing, which is introduced upon deallocations, where a freed object can be utilized by another thread. cache-thrash tests active false sharing, which in contrast is introduced during the initial allocations, where multiple continuous objects sharing the same cache line are allocated to different threads. Based on our understanding, NUMAlloc will not introduce active false sharing, since each thread will get a page of objects initially. Although NUMAlloc might introduce some passive false sharing due to its per-thread cache design, it avoids remote allocations across the node, where other allocators do not have such mechanisms. We believe that is the major reason for NUMAlloc's better performance.

In these four applications, NUMAlloc only performs worse than mimalloc for larson with 128 threads. larson simulates a multithreaded server that can respond to requests from different clients. In this application, each thread is given a set of objects, and they perform random deallocations and allocations on these objects within a round, and finally pass the objects to the next thread before terminating. Unlike other applications, larson runs for a fixed time and we use a throughput metric (the number of memory allocations per second) to measure the performance. Remote deallocations are quite common for this application, as local objects can be passed to other remote threads. Therefore, the performance of larson is sensitive to the memory recycling mechanisms of the allocator, as observed in the existing work [3, 54]. As discussed in Section 3.3, the remote object's deallocation is managed by the original node's freelist to ensure the locality. This freelist is shared among all threads running on this node, which can become a bottleneck when serving multiple deallocations at the same time. Nevertheless, NUMAlloc still performs better than most allocators on larson and can

Table 1. Memory consumption of different allocators.

Apps	Memory Usage (MB)								
	default Linux allocator	NUMAlloc		TCMalloc	TCMalloc-NUMA	jemalloc	TBB	Scalloc	mimalloc
		w/ THP	w/o THP						
blackscholes	615	779	615	635	622	633	615	631	622
bodytrack	35	95	46	129	43	566	35	2019	46
canneal	888	825	770	884	757	1286	889	9323	891
dedup	907	1200	1097	1111	1005	1397	910	10228	1592
facesim	2630	3428	2726	2973	2742	3540	2630	9102	3177
ferret	211	319	220	348	213	652	182	3424	647
fluidanimate	470	543	342	492	482	480	470	5371	476
freqmine	1877	1433	1342	1913	2658	1899	1859	1870	3084
raytrace	1287	1445	1627	1115	1414	1287	1288	9177	1392
streamcluster	112	132	111	160	123	127	113	195	138
swaptions	41	182	19	111	22	547	41	1817	14
vips	225	487	287	451	268	776	225	3666	983
x264	2856	3019	2721	3480	3064	3720	2860	5433	4094
Aget	8	53	5	38	47	94	42	122	44
Apache	4	4	4	10	19	10	4	42	4
Memcached	16	28	18	25	54	41	18	305	32
Mysql	282	429	243	327	609	854	282		975
Pbzip2	525	898	823	1088	859	1161	534	5413	7213
Pfscan	522	526	523	537	528	535	522	554	524
Sqlite3	45	113	83	119	75	143	46	686	109
Geomean	**1.00**	**1.56**	**0.99**	**1.57**	**1.40**	**2.30**	**1.09**	**7.47**	**1.72**
Total	**13556**	**15938**	**13622**	**15946**	**15604**	**19748**	**13565**	**69378**	**26057**

scale to 128 threads. All of these data indicate that NUMAlloc is scalable to 128 cores.

4.4 Design Choices

This section further confirms NUMAlloc's multiple design choices.

4.4.1 Choices of Thread Binding.

NUMAlloc's memory management is based on binding, including thread binding and memory binding. We believe such bindings benefit the performance and open up other design opportunities, such as origin-aware memory management, metadata allocation and incremental sharing. The combination of all design choices makes NUMAlloc a faster and more efficient allocator. Therefore, we cannot evaluate the impact of thread binding by directly disabling it on NUMAlloc since other designs depend on it. To overcome this problem, we implement a thread binding library that allows other allocators to enable binding. Figure 6(a) shows the impact of thread binding on two allocators, default Linux allocator and TCMalloc. The results are normalized to the data with thread binding of each allocator, respectively, so we omit the ones with thread binding. Thus, this figure can be considered to show how much slower it would run without thread binding. Here we use the node-interleaved binding. As shown in Figure 6(a), the thread binding improves the performance significantly for some applications. For instance, `fluidanimate` runs around 4.45× faster on default Linux allocator and 3.66× faster on TCMalloc with the node-interleaved thread binding. Similarly, `streamcluster` runs around 20% and 30% faster than the corresponding one without the binding. We further use `perf` [2] to analyze the reasons for the significant performance improvement of these two applications. The results confirm that remote accesses are significantly reduced with thread binding, mainly due to the elimination of thread migration. Interestingly, the cache miss rate also decreases with thread binding. Overall, thread binding will benefit the performance of most applications without hurting others, which should be included in the memory allocator by default.

We also compare the performance of two types of thread binding: node-interleaved and node-saturate thread binding. In node-saturate binding, we bind the maximum possible number of threads (same as the number of cores) to a node and then switch to the next node. As shown in Figure 6(b), the node-interleaved thread binding is almost always better than node-saturate thread binding, except for `vips`. On average, node-interleaved binding is around 19% faster than node-saturate one for these evaluated applications. This indicates that people should use node-interleaved binding, if they would like to employ all hardware cores. However, if they only want to use partial cores, then the node-saturate binding could be a better choice. Furthermore, NUMAlloc allows users to adjust the binding option according to their requirements.

4.4.2 Impact of Origin-aware Deallocation.

NUMAlloc adopts an origin-aware memory management to ensure the

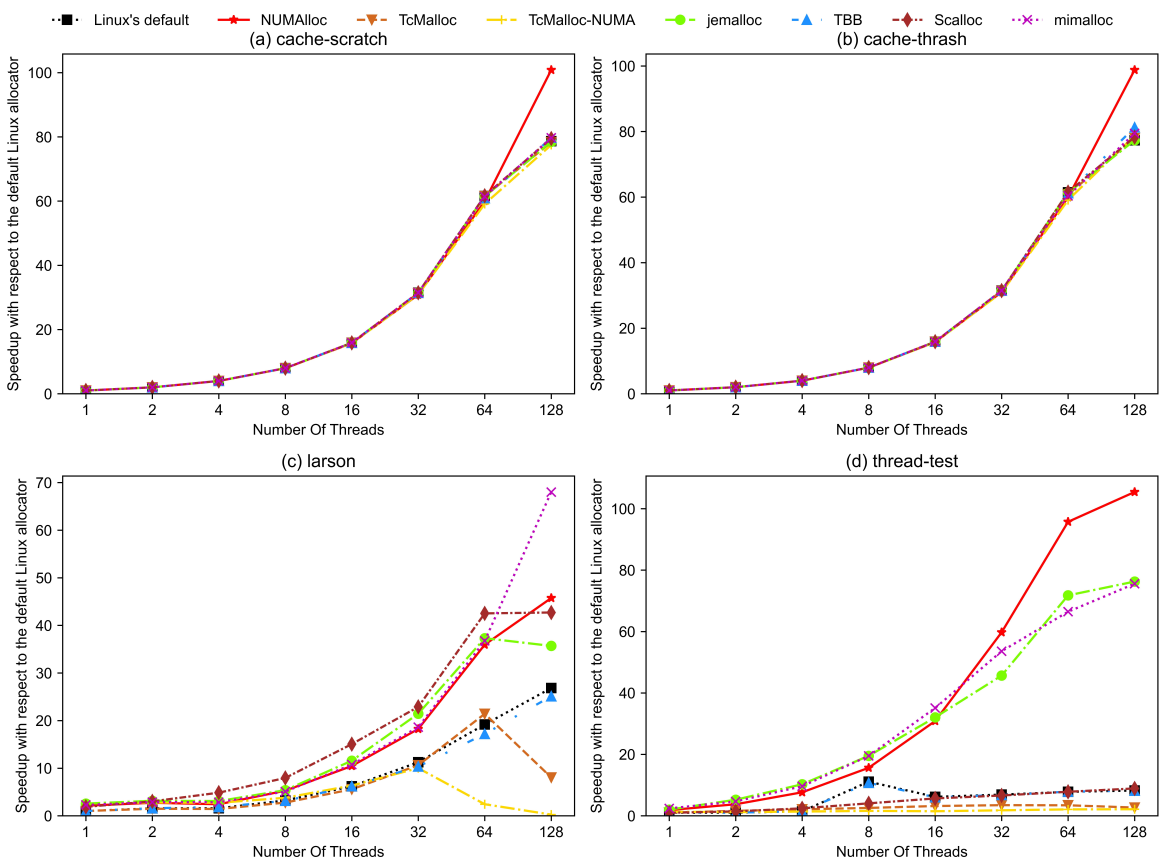


Figure 5. Scalability evaluation of different allocators.
All data are normalized to the runtime of the default Linux allocator with one thread.

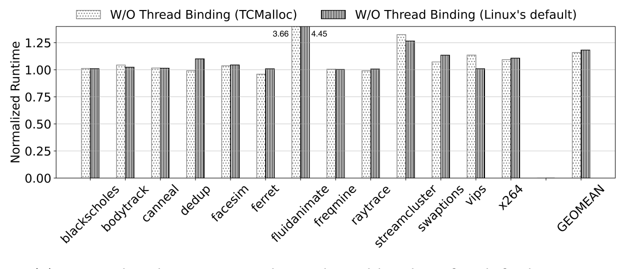
locality of memory allocations and deallocations, as discussed in Section 3.3. Some NUMA-aware allocators take locality into consideration during allocation, but neglect to handle the remote deallocation, resulting in remote accesses when reusing the memory. Instead, `NUMAlloc` proposes origin-aware deallocation which guarantees that a freed object will always return to its original node's heap. We further verified the effect of this design and the results are shown in Figure 7, where the data is normalized to the runtime with origin-aware deallocation. According to the Figure 7, all evaluated applications benefit from origin-aware deallocation and applications that have more remote deallocations, such as canneal, streamcluster and vips, achieve significant performance improvements. Overall, `NUMAlloc` runs 3.8% slower if we do not consider the origin of freed objects.

4.4.3 Impact of Incremental Sharing.

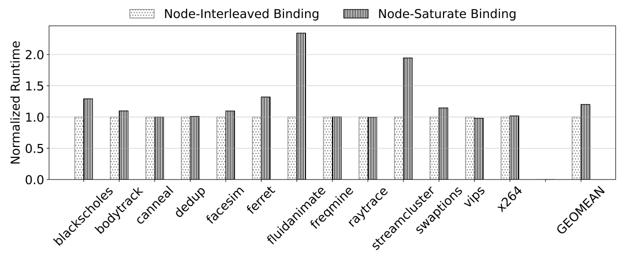
As discussed in Section 3.4, it is beneficial to embrace the transparent huge page support in modern systems. We evaluate the performance impact of transparent huge pages. The results are shown in Figure 8. When integrating with transparent huge pages, `NUMAlloc` achieves significantly better performance for `vips`, where it is running 16% faster. On average, transparent huge pages improve the performance by about 2.62%. There are no applications that run slower with huge pages. This clearly indicates that it is beneficial to enable transparent huge pages for the NUMA architecture, especially when `NUMAlloc` is used. Although using huge pages may increase the memory overhead, our incremental sharing mechanism helps to reduce the memory fragmentation. In our experiments with the PARSEC benchmark, we observed an average savings of 10.1% in memory overhead when incremental sharing is enabled. As shown in Table 1, the memory overhead of `NUMAlloc` is still acceptable, given the comparison of other mainstream allocators and the hardware trend of increasing memory capacity.

5 Discussion

This section describes some limitations of `NUMAlloc`. First, `NUMAlloc` may consume more memory than some popular allocators, especially when transparent huge pages are enabled. `NUMAlloc` currently allocates a big chunk (larger than a huge



(a) Normalized runtime without thread binding for default Linux allocator and TCMalloc, where the lower is the better.



(b) Normalized runtime with node-interleaved and node-saturate binding for `NUMAlloc`, where the lower is the better.

Figure 6. Performance impact of thread binding.

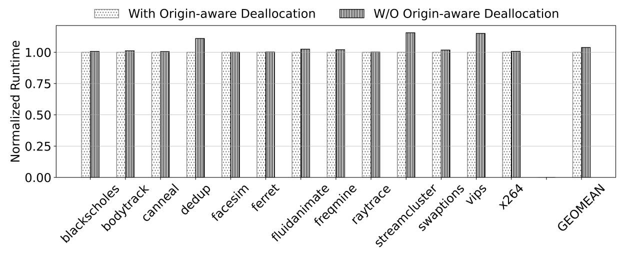


Figure 7. Normalized runtime with and without origin-aware deallocation for `NUMAlloc`.

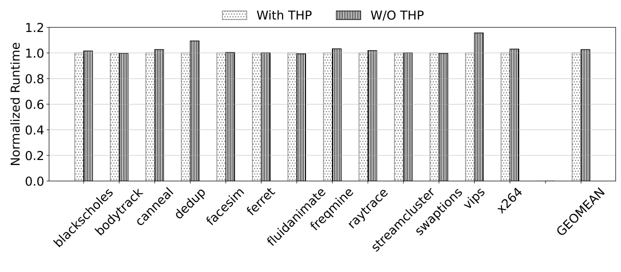


Figure 8. Normalized runtime with and without THP for `NUMAlloc`.

page) from the OS, then the OS will satisfy the memory allocations with huge pages when transparent huge pages are enabled. Although this method reduces the possible system call overhead and enjoys the performance benefits caused by reducing TLB misses, it does introduce more memory consumption. That is, the whole huge page will be wasted even if applications only use a small portion of huge pages. However, we believe that the memory overhead can be further reduced by more fine-grained management, such as TEMERAIRE's mechanism. We leave this implementation to our future work.

Second, NUMAlloc is designed with explicit thread binding, where people may be concerned that it conflicts with the OS scheduler. In fact, based on our understanding, this should not be a big issue due to the following reasons. (1) `NUMAlloc`'s thread binding does not exclude OS-based scheduling, as it only binds a thread to a node rather than a core. (2) `NUMAlloc` allows users to adjust the binding flexibly via a configuration file to meet the needs of different workloads. (3) Thread binding is even suitable for server applications with thousands of threads, as `NUMAlloc`'s binding balances the workload among different physical nodes.

6 Related Work

This section discusses some related work of `NUMAlloc`.

General Purpose Allocators. There exists a large number of allocators [3, 8, 20, 24, 40], but they are not designed for the 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 [23, 50]. Region-based allocators are suitable for special situations where all allocated objects within the same region can be deallocated at once [23]. For sequential allocators, subsequent memory allocations are satisfied in the continuous memory area, such as the Linux allocator [40] and Windows allocator [50]. That is, objects of 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. `NUMAlloc` also belongs to BiBOP-style allocators, as do many other high-performance and security-focused allocators [3, 8, 20, 24, 50]. But `NUMAlloc` 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 [31], 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 [62]. But JArena requires the co-design of applications, runtime system and the underlying OS, which is not transparent to users [62]. Also, both of them invoke too many `mbind` system calls, and do not handle the metadata's locality. nMART proposes a NUMA-aware memory allocation for soft real-time systems [35]. It proposes a node-oriented allocation policy to minimize the access latency, and ensures temporal and spatial guarantees 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. mimalloc also supports NUMA memory management [41]. It records the associated NUMA node for each segment, and tries to obtain a segment from the same node when reusing segments between threads. mimalloc proposes a page-based freelist that could only serve a thread at a time [41], where all objects will be returned to the same page-based freelist upon deallocations. By allocating the physical memory of each page locally, mimalloc has achieved some level of locality. However, mimalloc cannot ensure local allocations when a thread is migrated. NUMAlloc overcomes these issues, and further balances the memory accesses from different nodes via its node-interleaved thread binding.

NUMA-Aware Java Heap Management. Some approaches focus on improving the performance of 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 [51], 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 [57]. 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 [25].

Combination of Task Scheduling and Memory Management. Redline integrates task scheduling and memory management inside the OS level [61], to support interactive applications. Majo et al. propose to consider both data locality and cache contention to achieve better performance for the NUMA applications [46]. 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 [59]. Majo et al. propose to set task-to-thread affinity, and pin threads to specific cores to achieve a better performance [48]. Diener proposes a new kernel framework to combine task management and memory management together to achieve better performance [17]. Debes et al. propose the combination of enhanced work-pushing and deferred allocation together to improve the performance for data-parallel tasks, but focus on special programming models [19]. They inspire NUMAlloc's binding-based memory management. But NUMAlloc is the first work that exploits the benefits of binding inside a memory allocator.

Huge Page Support of Memory Allocators. SuperMalloc [36] is possibly the first allocator that supports huge pages. To reduce memory waste, it only utilizes huge pages for large objects. LLAMA [45] allocates memory objects with a similar expiration time to the same huge pages, and utilizes machine learning to identify the lifetime of memory objects from each callsite. That is, LLAMA requires the profiling to adjust its memory allocations for each application, which could be expensive or inconvenient to do so. TEMERAIRE [28], which is the default setting of TCMalloc, maximizes the usage of huge pages. It allocates big objects from huge pages, and also allocates small objects from partially-filled huge pages. However, based on our investigation, TEMERAIRE does not allow different threads to share the same huge page, possibly caused by the reason that TCMalloc is not a binding-based allocator, then making different threads share the same huge page could potentially introduce too many remote accesses. Instead, NUMAlloc allows the sharing of huge pages from different threads, which helps reduce the internal memory fragmentation. It is worth noting that NUMAlloc did not utilize some sophisticated mechanisms (such as TEMERAIRE or LLAMA) to manage huge pages, but achieved a similar memory overhead with TCMalloc. NUMAlloc could be further improved by borrowing some sophisticated mechanisms of LLAMA and TEMERAIRE in the future.

NUMA Libraries. Cantalupo et al. propose multiple APIs that allow users to manage their memory in fine granularity by combining with multiple existing system calls [12]. However, they are not targeting a general-purpose allocator, since it requires programmers to manage the memory explicitly. Majo et al. propose multiple source-code based algorithmic changes in order to improve data sharing and memory access patterns for NUMA architectures [47]. Williams et al. propose to group data structures that can be migrated together with arenas [60]. Shoal also proposes a set of APIs that allow the user to specify memory access patterns [30]. 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 [10, 11, 14, 15, 42]. NUMAlloc belongs to a proactive approach that does not require explicit and page migration, which is complementary to these reactive systems.

7 Conclusion

NUMAlloc is a memory allocator that is specially designed for the NUMA architecture. Applications can be linked to NUMAlloc directly, without the change of code and recompilation. NUMAlloc is different from existing memory allocators, as it is the first binding-based allocator. On top of it, it further proposes origin-aware memory management and incremental sharing to improve the locality and exploit huge pages. Based on our extensive evaluation, NUMAlloc achieves a significantly better performance than other popular allocators on the NUMA architecture, which is running 15.7% faster (and up to 4.6× faster) than the second-best allocator.

References

[1] 2017. CORAL-2 Benchmarks. https://asc.llnl.gov/coral-2-benchmarks.

[2] 2020. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page.

[3] 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)*. 451–469. https://doi.org/10.1145/2814270.2814294

[4] Martin Aigner, Christoph M. Kirsch, Michael Lippautz, and Ana Sokolova. 2016. scalloc. https://github.com/cksystemsgroup/scalloc.

[5] Periklis Akritidis. 2010. Cling: A Memory Allocator to Mitigate Dangling Pointers. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. 177–192. http://www.usenix.org/events/sec10/tech/full_papers/Akritidis.pdf

[6] Andreas Kleen at SUSE LINUX. 2012. "A NUMA API for LINUX". http://developer.amd.com/wordpress/media/2012/10/LibNUMA-WP-fv1.pdf.

[7] Avi Kivity . 2016. Automatic NUMA balancing may reduce performance. https://github.com/scylladb/scylla/issues/1120.

[8] 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). 117–128. https://doi.org/10.1145/378993.379232

[9] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. 72–81. https://doi.org/10.1145/1454115.1454128

[10] 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). 1–1. http://dl.acm.org/citation.cfm?id=2002181.2002182

[11] W. Bolosky, R. Fitzgerald, and M. Scott. 1989. Simple but Effective Techniques for NUMA Memory Management. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles (SOSP '89)*. Association for Computing Machinery, 19–31. https://doi.org/10.1145/74850.74854

[12] 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. (No. SAND2015-1862C)*. Technical Report. Sandia National Lab.(SNL-NM), Albuquerque, NM.

[13] William Cohen. 2014. Examining Huge Pages or Transparent Huge Pages performance. https://developers.redhat.com/blog/2014/03/10/examining-huge-pages-or-transparent-huge-pages-performance.

[14] Jonathan Corbet. 2012. AutoNUMA: The Other Approach to NUMA Scheduling. https://lwn.net/Articles/488709/.

[15] 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)*. 381–394. https://doi.org/10.1145/2451116.2451157

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

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

[18] 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.

[19] Andi Drebes, Antoniu Pop, Karine Heydemann, Albert Cohen, and Nathalie Drach. 2016. Scalable Task Parallelism for NUMA: A Uniform Abstraction for Coordinated Scheduling and Memory Management. In *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation, PACT 2016, Haifa, Israel, September 11-15, 2016*, Ayal Zaks, Bilha Mendelson, Lawrence Rauchwerger, and Wen-mei W. Hwu (Eds.). ACM, 125–137. https://doi.org/10.1145/2967938.2967946

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

[21] OpenBSD Foundation. 2012. "OpenBSD". "https://www.openbsd.org".

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

[23] 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

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

[25] 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

[26] Mel Gorman. 2012. Foundation for automatic NUMA balancing. "https://lwn.net/Articles/523065/".

[27] David R Hanson. 1980. A portable storage management system for the Icon programming language. *Software: Practice and Experience* 10, 6 (1980), 489–500.

[28] A.H. Hunter, Chris Kennelly, Paul Turner, Darryl Gove, Tipp Moseley, and Parthasarathy Ranganathan. 2021. Beyond malloc efficiency to fleet efficiency: a hugepage-aware memory allocator. In *15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21)*. USENIX Association, 257–273. https://www.usenix.org/conference/osdi21/presentation/hunter

[29] Intel Corporation. [n. d.]. Intel VTune Performance Analyzer. http://www.intel.com/software/products/vtune.

[30] 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

[31] 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.

[32] Alex Katranov and Anton Potapov. 2021. oneAPI Threading Building Blocks. https://github.com/oneapi-src/oneTBB.

[33] Alex Katranov and Michael Voss. 2020. Optimize Intel oneAPI Threading Building Blocks for NUMA Architectures. https://www.intel.com/content/www/us/en/developer/videos/onetbb-optimizing-for-numa-architectures.html.

[34] Chris Kennelly and Paul Burton. 2021. TCMalloc: Implement NUMA awareness. https://github.com/google/tcmalloc/commit/ef7a3f8d794c42705bf4327ca79fa17186904801.

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

[36] Bradley C Kuszmaul. 2015. SuperMalloc: a super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 International Symposium on Memory Management.* 41–55.

[37] 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

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

[39] Per-Åke Larson and Murali Krishnan. 1998. Memory Allocation for Long-Running Server Applications. *SIGPLAN Not.* 34, 3 (Oct. 1998), 176–185. https://doi.org/10.1145/301589.286880

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

[41] Daan Leijen. 2020. mimalloc. https://github.com/microsoft/mimalloc.

[42] 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

[43] 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 '14).* ACM, New York, NY, USA, 259–272. https://doi.org/10.1145/2555243.2555271

[44] Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram, and Ulrich Drepper. 2019. The GNU C Library Reference Manual. https://www.gnu.org/software/libc/manual/2.28/pdf/libc.pdf.

[45] 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.* 541–556. https://doi.org/10.1145/3373376.3378525

[46] 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

[47] Zoltan Majo and Thomas 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

[48] 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

[49] C. McCurdy and J. Vetter. 2010. Memphis: Finding and fixing NUMA-related performance problems on multi-core platforms. In *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS).* 87–96. https://doi.org/10.1109/ISPASS.2010.5452060

[50] 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

[51] 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

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

[53] Kirill Rogozhin. 2014. Controlling memory consumption with Intel® Threading Building Blocks (Intel® 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".

[54] Scott Schneider, Christos D. Antonopoulos, and Dimitrios S. Nikolopoulos. 2006. Scalable Locality-Conscious Multithreaded Memory Allocation. In *Proceedings of the 5th International Symposium on Memory Management* (Ottawa, Ontario, Canada) *(ISMM '06).* Association for Computing Machinery, New York, NY, USA, 84–94. https://doi.org/10.1145/1133956.1133968

[55] Sam Silvestro, Hongyu Liu, Corey Crosser, Zhiqiang Lin, and Tongping Liu. 2017. FreeGuard: A Faster Secure Heap Allocator. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017.* 2389–2403. https://doi.org/10.1145/3133956.3133957

[56] Sam Silvestro, Hongyu Liu, Tianyi Liu, Zhiqiang Lin, and Tongping Liu. 2018. Guarder: An Efficient Heap Allocator with Strongest and Tunable Security. In *Proceedings of The 27th USENIX Security Symposium (Security'18).*

[57] 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

[58] François Trahay, Manuel Selva, Lionel Morel, and Kevin Marquet. 2018. NumaMMA: NUMA MeMory Analyzer. In *Proceedings of the 47th International Conference on Parallel Processing* (Eugene, OR, USA) *(ICPP 2018).* Association for Computing Machinery, New York, NY, USA, Article 19, 10 pages. https://doi.org/10.1145/3225058.3225094

[59] 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.

[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

[61] 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

[62] 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).

[63] Xin Zhao, Jin Zhou, Hui Guan, Wei Wang, Xu Liu, and Tongping Liu. 2021. NumaPerf: Predictive NUMA Profiling. In *Proceedings of the ACM International Conference on Supercomputing* (Virtual Event, USA) *(ICS '21).* ACM, 52–62. https://doi.org/10.1145/3447818.3460361

[64] L. Zhu, H. Jin, and X. Liao. 2016. A Tool to Detect Performance Problems of Multi-threaded Programs on NUMA Systems. In *2016 IEEE Trustcom/BigDataSE/ISPA.* 1145–1152. https://doi.org/10.1109/TrustCom.2016.0187

Received 2023-03-03; accepted 2023-04-24