NUMALLOC: Adaptive NUMA Memory Allocator

ANONYMOUS AUTHOR(S)

The NUMA architecture was proposed to accommodate the hardware trend with the increasing number of CPU cores. However, existing memory allocators have multiple issues for support the NUMA architecture: most of them were not designed for the NUMA architecture; NUMA-aware allocators still miss some performance opportunities promised by the hardware.

This paper proposes a novel memory allocator–`numalloc`—that is designed for the NUMA architecture. Different from existing allocators, `numalloc` integrates the task assignment into the allocator, where each thread is bound to a specific core in an interleaved way. Based on the location of a task, `numalloc` designs a node-aware memory allocation. `numalloc` also utilizes huge page support to improve the performance, and designs its interleaved and block-wise memory allocation to accommodate shared objects. Based on its extensive evaluation, `numalloc` achieves up to 5× performance speedup comparing to the default Linu x allocator.

1 INTRODUCTION

The Non-Uniform Memory Access (NUMA) architecture is an appealing solution for the scalability of multi-core era. Compared to Uniform Memory Access (NMUA) architecture, the NUMA architecture avoids the bottleneck of a central memory controller: each processor (also known as a domain or node) consists of multiple cores, while each node can access its own memory controller. Different nodes are connected via high-speed inter-connection (such as Quick Path Interconnect (QPI) [20] or HyperTransport bus [8]) to form a cache-coherent system that presents an abstraction of a single globally addressable memory.

However, NUMA systems may have serious performance issues if programs have one of the following issues, such as large amount of remote accesses, load imbalance, contention for interconnection and last level cache [4, 10], as discussed more in Section 2. There exists multiple types of approaches that devoting to improve the performance of NUMA systems.

First, some research may rely on programmers to manage memory allocations and task assignments explicitly [21, 27, 34**?**]. Although they could improve the performance greatly, they typically require significant human effort to rewrite the programs, where legacy systems cannot benefit from.

Second, some approaches migrate tasks or physical pages reactively based on memory access patterns or other hardware characteristics [4, 9, 10, 26]. These reactive approaches could improve the performance automatically without human involvement. However, they cannot solve all issues. For instance, if multiple objects located in the same page are concurrently accessed by multiple threads, either task migration or page movement may not offer much benefit [16].

Third, some researchers focus on the memory management by providing a new memory manager for existing software []. However, none of these systems could achieve the performance promised by the hardware. Given a simple example, if one page is fulled with objects that are utilized by different threads running on different nodes, then it is not able to achieve the best performance no matter where this page is located. Similarly, the pemissive tracking of memory ownership of an object cannot completely avoid the ownership drifting problem, where the .

XXXX-XXXX/2020/4-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

In this paper, we propose a new memory management method that employs the proactive memory management policy. There are multiple approaches: first, it combines the affinity of memory management and task management. Tasks were bound to memory nodes initially, and memory management will be based on the placement of each thread. Second, it proactively bound the memory to different nodes, so that it could reduce the . Third, it takes a different method to deal with allocations from main thread, Fourth, it takes the advantage of hardware progress automatically, e.g., huge pages. One application achieves 10% performance speedup with this consideration.

There are other implementation issues that may also contribute to the performance improvement. For instance, the metadata will be always allocated locally. The big objects will be re-utilized by small objects automatically, in order to reduce the potential cache misses.

We have designed a memory allocator based on the methods described above, and have evaluated it on a range of benchmarks and real applications on two different NUMA architecture. The evaluation showed that `numalloc` could significant improve the performance by up to 2X, with the average of 10% performance. The evaluation also show that `numalloc` has a bigger performance improvement on

How to design the new allocator? This is the new one.

We will try to achieve the following target.

(1) The performance will be as efficient as possible. (2) It is still For information-computable, we will achieve the following targets: We should still use the address to infer the following information, such as the size of the object and the shadow memory placement.

We don't support many bags for the same size class, just one big bag for each size class of each heap. Why it is necessary?

We would like to reduce the memory blowup as much as possible, where the memory will be returned back to the current thread's heap.

In order to support that, maybe we could have a big heap for the same size class, but different threads will start from different placement.

Is it good for the NUMA support in the future? For NUMA support, it is great if we can always return the memory back to its original

Virtual address will be divided into multiple nodes.

Then inside each step, we will divide a sub-heap into multiple mini-heaps, where each mini-heap will support one thread.

Virtual address will be divided into multiple nodes.

Then inside each per-node heap, we will further divide it into multiple mini-heaps, where each mini-heap will support one thread in order to reduce the possible contention.

For each per-thread heap, we will have two free-lists for each size class. The first one will be used for the current thread, without the use of lock at all. The second one will be utilized for returning memory from other threads.

RTDSCP is a way to know where a thread is executed on.

[[Should we put the metadata into the corresponding node? We will minimize the lock uses for each node, since that will invoke unnecessary remote accesses as well.]]

What is the uniqueness of NUMA architecture?

Remote accesses and imbalance? What are the big issues that could cause the performance issues on the NUMA architecture.

Programs also have some inherent imbalance. For instance, main thread typically will prepare the data for all children threads. Therefore, many existing tools discover that the block-wised allocation is one way to reduce the imbalance [28**?**]. Also, many applications are typically utilizes a producer-consumer architecture, where the objects allocated in one thread will be deallocated by another thread. This issue, if not handle correctly, will cause the ownership thrifting issues, which will

cause that it is impossible to know the placement of an object after a while. Very quickly, it may cause a lot of unnecessary remote accesses, when using the existing NUMA allocator [].

Therefore, a good NUMA allocator should deal with these inherent imbalance of programs. Also, it should deal with the normal problems of NUMA hardware. It should

1.1 Novelty

- We will design a node-aware memory allocator that could actually identify the memory placement by using the virtual address.
- We will minimize the synchronization, since that will impose unnecessary remote accesses.
- We will minimize remote accesses by putting the metadata into multiple nodes.
- We may support multiple types of allocation, such as block-wise false sharing memory accesses, or node-balanced memory accesses, relying on the indication from user space.
- We will balance the memory consumption over all nodes, in order to avoid the contention of memory controller [31]. At least, we could balance the memory consumption among all allocators. Ideally, it is better to balance the memory accesses.
- We may track the relationship of all objects. Then those objects will be allocated correspondingly. For instance, if the objects are allocated in the main thread, we assume that they will be accessed as a shared mode. Then we would like to allocate in the node-balanced areana initially.

1.2 TODO list

(1) Dividing each node's memory into two parts, one for small objects, and one for big objects.
(2) Making the main thread to use a dedicate memory region. Memory will be allocated in different nodes in an interleaved way. However, this method has some benefits and shortcomings. If the memory is deallocated immediately, then it is time-consuming to put it remotely, and will hurt the performance. Therefore, it is necessary to identify such cases, and do not allocate such objects in the interleaved way.
(3) Fixing the merge and split for big objects
(4) Thinking about the integration of Ding Chen's theory. (Let Xin to do this part)
(5) Thinking about using the inside of the object for holding the pointers for link list, which will be faster and remove some memory overhead unnecessarily. (Let Xin to do this part)
(6) For producer-consumer threads, we will also track which objects are allocated and freed in different threads. For those objects, maybe they could utilize the same mechanism as the main thread. But we will do this in the future.
(7) Intercepting mmap and use the mainthread's interleaved idea.
(8) We will use huge pages by default in order to improve the performance. But we will utilize small bags for small size classes, which we will learn from TcMalloc. That is, we don't want to use 1 Megabyte information. Instead, we may utilize the 4 pages as an unit.

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 have its own 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 [4], including *Cache Contention*, *Node Imbalance*, *Interconnect Congestion*, and *Remote Accesses*. Based on the study [4], node imbalance and interconnect congestion may have a larger performance impact than cache contention and remote accesses.

Cache Contention: the NUMA architecture is prone to cache contention issue, where multiple tasks may compete for the shared cache.

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 the memory access, 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.

Overall, the NUMA architecture has multiple potential performance issues. However, these performance issues cannot be solved by the hardware automatically. Software support is required to control the placement of tasks, physical pages, and objects in order 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 [13, 25]. 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. 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 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. However, they all require programmers to specify the policy explicitly, which cannot achieve the promised performance by hardware automatically. `numalloc` relies on these existing system calls to support NUMA-aware memory allocations, but will achieve a better performance without changing explicit specification.

3 DESIGN AND IMPLEMENTATION

`numalloc` is designed as a drop-in replacement for the default memory allocator. It intercepts all memory allocation/deallocation APIs via the preloading mechanism. Therefore, there is no need to change the source code of applications to employ `numalloc`, and there is no need to use the custom OS or hardware.

Different from existing work, `numalloc` aims to reduce remote accesses, and balance the workload among different hardware nodes. It also utilizes huge page support to improve the performance, and designs its interleaved and block-wise memory allocation to accommodate shared objects. Multiple components that differentiate it from existing allocators are discussed in the remainder of this section.

`numalloc` also borrows many known mechanisms of existing allocators. First, it utilizes the size class to manage objects. Instead of allocating the exact size, `numalloc` will round the size of an allocation to its closest size class. Similar to TcMalloc [17], `numalloc` also utilizes fine-grained size classes for small objects, such as 16 bytes apart for objects less than 128 bytes, and 32 bytes apart for objects between 128 bytes and 256 bytes, then power-of-2 sizes afterwards. Second, it utilizes the "**Bi**g-**B**ag-**o**f-**P**ages" mechanism that all objects in the same bag will have the same size class, and separates the metadata from actual objects. `numalloc` only tracks the size information of each page, which helps reduce its memory overhead for the metadata. Third, `numalloc` utilizes freelist to manage freed objects. Every freed object will be added into a corresponding freelist, and objects in the freelist will be allocated first in order to reduce possible cache misses. Further, `numalloc` utilizes the first word of freed objects to link different objects, which is similar to Linux and TcMalloc [17]. This mechanism helps to reduce the memory overhead, but is prone to memory vulnerabilities, such as buffer overflows and double-frees [35, 40].

3.1 Topology Aware Task Assignment

Existing allocators typically do not schedule tasks explicitly, but relying on the default OS scheduler. The OS scheduler performs very well for the UMA architecture, since the latency of accessing the memory controller is the same for every core. However, the NUMA architecture imposes additional challenge [33]. If a task is moved to a new node, it has to access all of its memory through the interconnect, resulting in a higher memory latency. The scheduling may lead to significant performance difference for memory-intensive applications. Therefore, typically tasks are bound to a specific core/processor for the NUMA architecture [33, 42, 43].

Due to the importance of task assignment to memory locality, *`numalloc` embeds a topology-aware task assignment, which makes it different from existing allocators*. Every thread is bound to a specific node, where its memory allocations will be based on the location of its task, as described in Section 3.2. To balance the workload of each node, `numalloc` utilizes a round-robin manner on the nodes to assign the tasks , which is similar to TBB-NUMA [33]. Basically, a newly-created thread will be assigned to a node that is different from its preceding and subsequent sibling. This ensures that each processor/node will have a similar number of threads, and therefore a similar workload. An

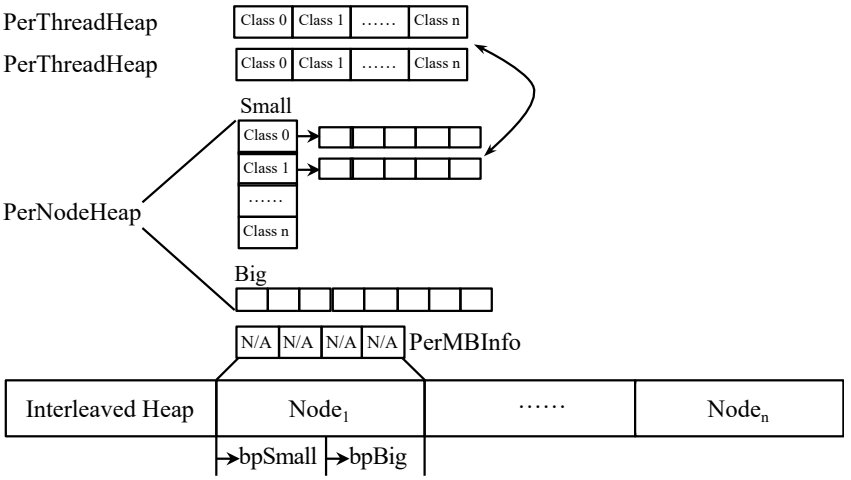

Fig. 1. Overview of `numalloc`.

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 approach 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.

3.2 Node-Aware Memory Management

`numalloc` designs a node-aware memory management based on its task assignment. All heap objects of a thread will be *physically* allocated from the node that the thread is currently pinned to, unless otherwise stated (Section 3.3). Since a memory allocator only deals with virtual memory, but not physical memory, *`numalloc` binds every range of its virtual memory to a particular node via the `mbind` system call*. Therefore, `numalloc` is able to identify the physical node information based on a virtual address, which is the same as existing work [22]. However, `numalloc` manages memory allocations and deallocations differently, which explains its significant performance advantage over the existing work [22] (see Section **??**).

`numalloc` handles deallocations of objects carefully, in order to ensure node-aware memory allocations. `numalloc` utilizes different mechanisms to handle large objects and small objects. In `numalloc`, an object with the size larger than 512KB will be treated as a big object. Otherwise, it is belonging to a small object. As described above, `numalloc` maintains multiple size classes for small objects. Upon each deallocation, `numalloc` determines its physical node (based on its virtual address) and the size information in order to place it correspondingly. For small objects, if an object is allocated from a different node, it will be placed into the common freelist of that node. Otherwise, it will be placed into the freelist of the deallocation thread. Utilizing a per-thread list avoids

the synchronization overhead, since only one thread is allowed to access its per-thread list. Big objects are handled differently due to two reasons. First, an application typically has relatively fewer number of big objects. Therefore, it is expensive to maintain size classes for big objects. Second, every big object has a larger impact on the memory consumption, which should be re-utilized as soon as possible. Therefore, each node keeps a per-node freelist to track big objects in the current node. Different from existing allocators, big objects are not returned to the OS immediately upon deallocations [2, 17], which reduces cache loading operations and memory management overhead of the OS. Instead, they will be tracked in the per-node freelist, and will be utilized to satisfy future allocation requests of both big objects and small objects. Upon the deallocation of a big object, `numalloc` tries to coalesce with its previous and next neighbor, if its neighbor is also freed.

`numalloc` manages memory allocations as follows. Basically, an allocation will be satisfied from the corresponding freelist at first, if some freed objects exist. Otherwise, a new/never-used object will be allocated. Objects in the freelist will be allocated using the last-in-first-out order, which helps reduce possible cache misses since last-freed objects are more likely to be hot. A big object will be allocated from the per-node freelist, while a small object will be allocated from the thread's per-thread list if possible and then the per-node freelist. `numalloc` moves small objects between per-thread lists and per-node lists: if freed objects in a per-thread list is over a pre-defined threshold, a percentage of freed objects will be moved to the thread's per-node list; Whenever the per-thread list is empty, some objects will be moved from the corresponding per-node freelist. Both movement will take an adaptive mechanism. For instance, if a per-thread list keeps migrating objects from its per-node list successfully, then the percentage of moving to per-node list will be reduced. Similarly, if a per-node list do not have objects to move to the per-thread list, `numalloc` will move less frequently.

In its implementation, `numalloc` reduces the overhead of frequent system calls, and borrows the information-computable design from existing allocators in order to quickly locate the physical node information [39, 40]. The basic idea of its memory layout is illustrated in Fig. 1. Basically, `numalloc` takes the advantage of the huge address space of 64-bits machine to achieve the quick lookup. Instead of frequently allocating the memory from the OS, `numalloc` obtains a big chunk of memory (few terabytes) from the underlying OS initially, and then divides it to multiple chunks with the same size, where each chunk will be bound to a specific memory node as illustrated in Fig. 1. Therefore, the physical node information could be computed directly from the virtual address, by checking the distance from the starting address of the heap. Comparing the method of using a hash table to track the node information [22], this mechanism is much faster, since it avoids the hashing and lookup overhead.

Each chunk belonging to one node is further divided into two parts, one for small objects (managed with the `bpSmall` bump pointer) and one for big objects (managed using the `bpBig` pointer). There are two reasons to separate them. First, we plan to utilize huge page support for big objects, but not small objects, which is further described in Section 3.4. Second, placing big objects together helps to coalesce two continuous objects into one bigger object. For small objects, `numalloc` employs the "**Bi**g-**B**ag-**Of**-**P**ages" (BiBOP) style that each bag will have multiple continuous pages, and each bag will have objects with the same size class. `numalloc` currently sets its bag size of small objects to be one megabytes. An object with the size larger than 512 kilobytes is considered to be a big object, which will be always aligned to megabytes. Therefore, `numalloc` only requires to record the size information for each megabytes, shown as `PerMBInfo` in Fig. 1. `numalloc` embeds the availability information of each chunk to the `PerMBInfo` structure, using the lowest significant bit of the size. Each node has a per-node heap, which includes a per-node freelist to track big objects, and multiple freelists to track small objects with different size classes. As described above, each thread also has its freelists for different size classes. Small objects can be moved between a per-thread freelist and a per-node list as described above.

Cache Warmup Mechanism. For small objects, `numalloc` also borrows the cache warmup mechanism of TcMalloc [17]. TcMalloc utilizes the `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 adds all objects of this block into the 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 adding operation. 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 adding 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 freelist. A bump pointer is used to track the position of never-used objects, whose updates do not require to be exactly one page each time. By comparison, TcMalloc may waste some memory in the end of each block, if the block size is not equal to multiple times of the class size.

3.3 Interleaved and Block-wise Memory Allocation for Shared Objects

`numalloc` proposed a new interleaved and block-wise heap for shared objects, as illustrated in Fig. 1. Based on our observation, most NUMA performances issues identified by existing NUMA profilers are related to shared objects [24**?**]. Shared objects are typically allocated in the main thread, but are accessed concurrently by multiple 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. 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 the 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.

`numalloc` allocates the memory in a block-wise way for shared objects, when the size of an allocation is larger than the twice of the multiplication of nodes and the page size. The basic idea of block-wise allocation is illustrated in Fig. 2. The key reason to use the block-wise allocation is that children threads may access the memory continuously [**?**]. In addition to that, the block-wise allocation will still maintain the balance between different memory nodes, which won't cause unnecessary performance issue.

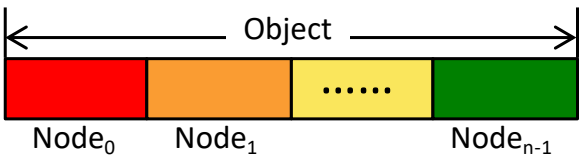

Fig. 2. Block-wise Memory Allocation

During the implementation, `numalloc` only allocates 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, especially for applications with the pipeline model, the overhead of tracking shared objects is typically larger than the performance benefit based on our evaluation.

It is very important to identify shared objects correctly. Otherwise, it will introduce performance issue by allocating private objects in a different node, causing remote accesses unnecessarily. `numalloc` utilizes the allocation/deallocation pattern to identify shared objects: an object is treated as a shared one initially, and is allocated from the interleaved heap; If a newly-allocated object is deallocated before creating children threads, all objects from the corresponding callsite are considered to be private objects. Therefore, `numalloc` utilizes the callsite to differentiate objects, which is determined by the program logic.

However, it is very challenging to determine the allocation callsite *uniquely* and *efficiently* due to multiple reasons. First, function pointers are removed in most applications if they were compiled with an optimization, which makes it impossible to obtain return addresses (and callsite) via built-in functions. Second, applications have a wrapper for memory allocations, which requires multiple levels of call stacks to uniquely identify one callsite. Although there exist mechanisms to encode calling context explicitly [41, 46], they require the recompilation of the applications. The `backtrace` function can obtain the callsite, but it is too slow to be used in production environment.

`numalloc` 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 when there exists the allocation wrapper, where multiple allocations inside the same function will be treated as the same callsite. A shared callsite can be treated as a private callsite, if an object from these callsites invokes the deallocation before creating children threads. For the performance reason, `numalloc` obtains the return address quickly via a constant offset, where the offset is uniquely determined after the compilation of `numalloc`.

`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 key is not existing in the hash table or 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. When an allocation is satisfied in the callsite and the callsite is new, the corresponding allocation will be tracked in the second hash table. The second hash table will be checked upon deallocations, where the corresponding allocation callsite will be marked as private if an object is allocated in the same epoch.

3.4 Automatic Huge Page Support

`numalloc` employs the existing huge page support to further improve the performance. Modern hardware typically installs with huge page support. A huge page is a page that its page size is much larger than 4 kilobytes. Currently, the Linux system has two page sizes, 4 kilobytes and 2 megabytes. Based on the existing study [19], 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. 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. We observed around 10% performance improvement for some applications, when we are using huge page support.

However, the default huge page support is not good for the performance [16, 37]. 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 [29], if a partial range of a huge page is not accessed.

`numalloc` utilizes huge pages differently. Ideally, if huge pages are utilized for private objects, then there are no hot page effect and page-level false sharing. Also, if huge pages are only utilized for big objects or all small objects in a huge page will be allocated, then the memory footprint can be minimized. `numalloc` takes these consideration into account. Each per-node heap is further divided into two parts as illustrated in Fig. 1: small objects will be allocated from the first half and will be allocated using small pages, while big objects will be allocated from the second half with

huge pages (2MB). When a big object is utilized for small objects, only frequently-allocated small objects can utilize such an object. We believe that our design balances the performance and memory consumption.

3.5 Efficient Object Migration

`numalloc` maintains per-thread heap and per-node heap in order to reduce the synchronization overhead, which indicates the necessity of moving objects between different heaps. All threads bounded to the same node will share the per-node heap. When a per-thread list has too many objects, some of them should be moved to the per-node list 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. These frequent migrations require an efficient mechanism. Currently, `numalloc` utilizes singly link lists to manage freed objects, imposing an additional challenge of migrating objects efficiently.

One straightforward method is to traverse the whole list to collect a batch of objects, and then moves them at a time. Although the idea of using the batch reduces the contention, but this still has multiple issues. First, traversing a list will actually pollute the cache of the current thread, especially when a thread is moving out objects from its per-thread heap. That is, the thread does not need to access these objects any more, where traversing these objects will bring these objects to the cache. Second, it is very slow to traverse a list, which may create thread contention when multiple threads are migrating freed objects from the per-node heap concurrently. We observed a 20% slowdown for some applications due to this straightforward mechanism. `numalloc` further proposes multiple mechanisms to migrate objects efficiently.

In order to avoid the pollution on the per-thread cache in moving out objects, 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 Figure 3. Therefore, if a thread is going to migrate n objects (between $n-1th$ and the `Tail` object), it only requires a forward check to obtain the pointer of $n-1th$ object and then it could move out n objects at a time. After the migration, the `Tail` pointer can be set to the original *nth* object. But this mechanism alone cannot reduce the contention when multiple threads are concurrently obtaining objects.

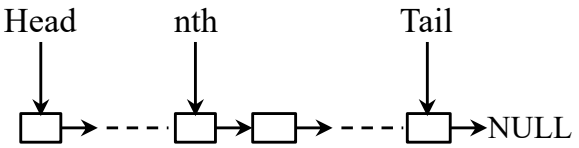

Fig. 3. Avoiding the traverse of per-thread freelist

In order to avoid the bottleneck of the per-node heap, `numalloc` introduces a circular array of freelists as shown in Figure 4, where the number of entries is a variable that can be changed by the compilation flag. This array has two pointers, one is `toGetIndex` and the other one is `toPutIndex`. When a thread migrates freed objects to the per-node heap, it will add the list of objects to the freelist pointed by `toPutIndex`,

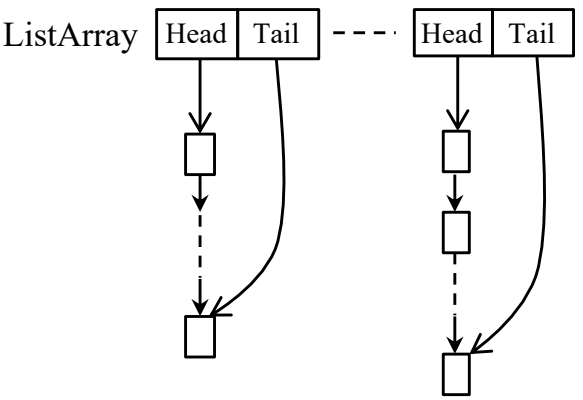

Fig. 4. An array of freelists for per-node heap

and update the index after the migration. Similarly, if a thread tries to obtain freed objects from the per-node heap, it will obtain all objects in the freelist pointed by the `toGetIndex`, and update the index afterwards. If the freelist pointed by the `toGetIndex` has no freed objects, indicating that there is no freed objects in the per-node heap for this size class, then the index is not updated. As described above, if a thread running on the other node just deallocates an object to the per-node heap, then this object will be added to the head of the freelist pointed by `toPutIndex`, but the index is not changed after the deallocation.

3.6 Other Mechanisms

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

Memory Reutilization: Some applications may create new threads after some threads have exited. `numalloc` re-utilizes the memory for these exited threads. Basically, `numalloc` introduces a thread index for each thread, which is utilized to index the corresponding per-thread heap. `numalloc` intercepts thread joins and cancels so that it can assign the indexes of exited threads for newly-created threads, and re-utilize their heaps correspondingly.

MiniBag Design to Reduce Memory Consumption: During the development, we noticed that it excessive memory consumption can be imposed when the OS utilizes huge pages by default. In order to avoid this issue, we propose the combination of per-node heap and per-thread cache. In order to reduce the contention, `numalloc` will obtain multiple objects at a time from the per-node heap.

4 EVALUATION

4.1 Performance Evaluation

We compare `numalloc` with multiple popular allocators, such as Linux's glibc allocator, TcMalloc [17], or NUMA Aware TcMalloc [22], jemalloc [14], Scalloc [1]. For the simplicity, NUMA aware TcMalloc is called as TcMalloc-NUMA in the remainder of this paper.

Performance evaluation will be performed on two different hardware as further described in the Table 1:

System	**Machine A**	**Machine B**
CPUs/Model	Xeon Gold 6138	
NUMA Nodes	2	8
Physical Cores	2×20	8×16
Node Latency	local: 1.0 1 hop: 2.1	
Interconnect Bandwidth	8GT/s	
Memory Bandwidth	19.87 GB/s	

Table 1. Machine Specifications.

~~*Application Statistics.* We also checked the corresponding details of~~

To make our evaluation results more convincing, we evaluated 23 applications, and showed both normalized values for every single of them and final average normalized values in the following sections. All the values are normalized based on the performance results of Linux's glibc allocator, so that a smaller value less than one means better performance and vice versa. Among the 23 applications, twelve are from the PARSEC suite [3] of applications, seven are real applications like Apache httpd-2.4.35, MySQL-5.7.15, Memcached-1.4.25, SQLite-3.12.0, Aget, Pfscan, and Pbzip2, and rest four are benchmarks from Hoard [2], including threadtest, larson, active-false sharing and passive-false sharing. All of these benchmarks are multi-threaded application and threads are distributed over nodes in our target machines sharing data with each other more or less, which made them more relevant toward gauging performance on modern NUMA machines than single-threaded benchmark suites, such as SPEC.

The number of threads of all benchmarks were adjusted according how many cores and nodes in the target machine to make threads could be properly distributed over the nodes and cores, making the number of threads as close as the number of cores. Mostly, thread number was 40 in the Machine A and 128 in the Machine B, and I will give the specific number below if it is not this default value. For specific, in PARSEC applications, native inputs [3] were used here.For MySQL, we used sysbench with 16 threads and 100,000 max requests. Besides, python-memcached [38] script was used for memcached with 3000 loops to get sufficient runtime and ab [15] is used to test Apache by sending 1,000,000 requests. For Aget, we tested it by dowloading a 30 M file. For Pfscan, we executed a keyword search in a 500M data. In terms of Pbzip2, we tested it by compressing 10 30M files. Finally, SQLite was tested through a program called "threadtest3.c" [11]. In the Hoard [2] benchmarks, we used 100 iterations and 1,280,000 64-byte objects for threadtest and also we run larson for 10 seconds with 1,000 7-2048 bytes object to cover all size classes in almost all allocators for 10,000 iterations.For false sharing , we used 100,000 inner-loop , 100,000 iterations with 8 bytes objects.

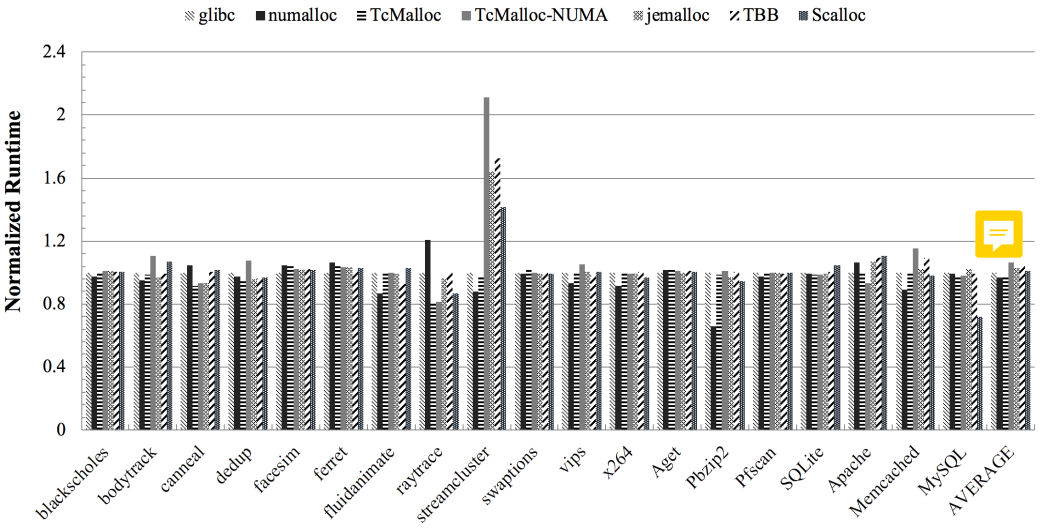

Fig. 5. Normalized runtime with different allocators for PARSEC and real applications in Machine A

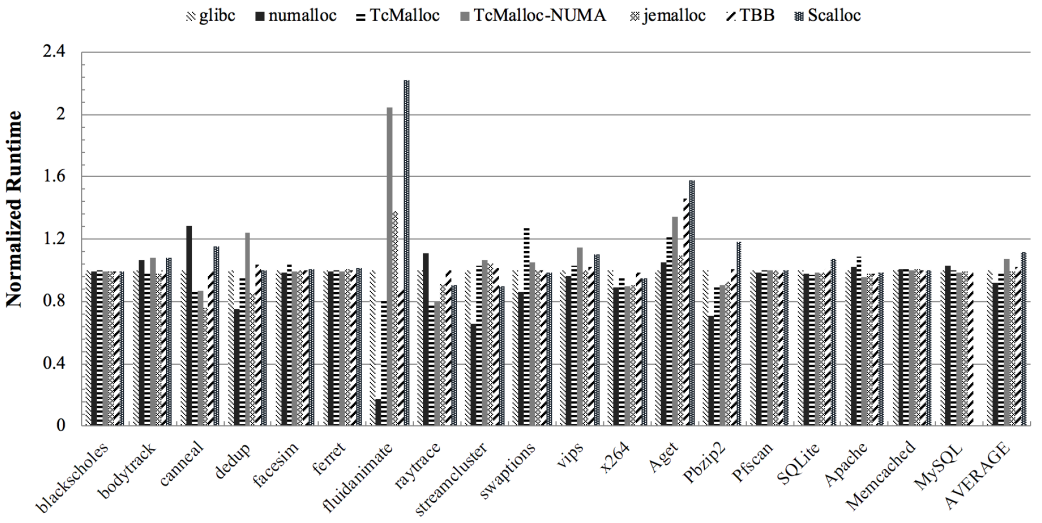

Fig. 6. Normalized runtime with different allocators for PARSEC and real applications in Machine B

Figure 5 shows the normalized runtime of different allocators in Machine A and figure 6 is for Machine B, compared with Linux allocator. We can see that the average value of `numalloc` is 0.97 in Machine A and 0.92 in Machine B and it is always the best among all other allocators. The reason that `numalloc` got better performance in Machine B is that there are more nodes and more cores in Machine B, which means `numalloc` could be very helpful to better to take use hardware resource of multi nodes and cores.In the figure 5, we could get that the average normalized performance of `numalloc` is 0.97 that is the best through these 7 allocators with 0.98 for TcMalloc, 1.00 for Scalloc. And `numalloc` is also the best for most of single applications here, especially for fluidanimate, streamcluster and x264 in which 10 percent of improvements or even more are gotten. Besides `numalloc`'s performance of ratrace and canneal is not very good here, which is caused by their few data sharing between threads, but we could get amazing improvement if we shutdown interleaved heap in `numalloc` and we will give the data in following sections.In the figure 6, we could see more exciting improvement from `numalloc`, with average normalized value of 0.92 that is not only the best but also far aware better than all the rest allocators that TcMalloc and jemalloc got 0.99, TcMalloc-NUMA and TBB got roughly 1.07 and 1.01 separately. And also, we can see that the performance of `numalloc` is the best for almost each single applications, especially it got 0.17 in fluidanimate and 0.66 in streamcluster which is far better than any of other allocators. As the same thing, the performance of ratrace and canneal is not good here, we will talk about it later after we shut down the interleaved heap.

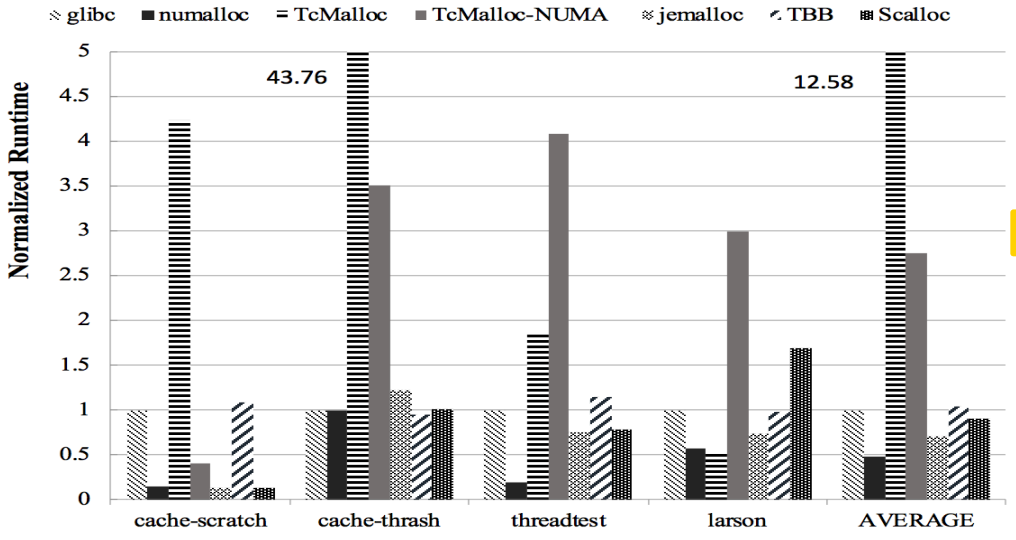

Fig. 7. Normalized runtime with different allocators for Hoard benchmarks in Machine A

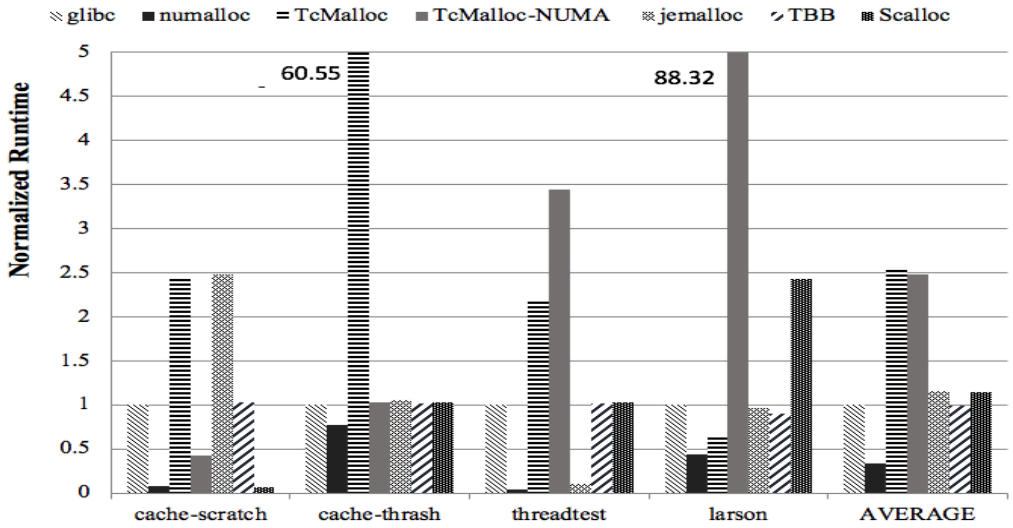

Fig. 8. Normalized runtime with different allocators for Hoard benchmarks in Machine B

In the figure 7 and figure 8, we show the normalized performance for Hoard benchmarks in Machine A and Machine B separately. We can see from figure 7 that the average value of `numalloc` is also the best, which is 0.47 that means 2 times faster than Linux's glibc, and jemalloc got 0.7 and Scalloc got 0.9. In the threadtest, the normalized value of `numalloc` is 0.19 , far better than any of others, which means there are few central free list competitions, mainly contributed by properly node management and low overheads operations. For false sharing, `numalloc`'s performance is also almost the best as same as Scalloc and jemalloc, which means they could handle false sharing issues

very properly. In the larson, numalloc and TcMalloc are the best, which mainly contributed by their low overheads for allocation and remote de-allocation, but due to our better node management, numalloc could be better in the Machine B which will be mentioned later. In the figure 8, we can also see that numalloc got lowest average normalized value:0.33, significantly smaller than any of others that TBB got 0.99, Scalloc and jemalloc got roughly 1.14. And also, numalloc and Scalloc could handle false sharing issue very well, and numalloc could extremely well reduce central free list competition in threadtest. In larson, numalloc is the best due to its properly multi-node management.

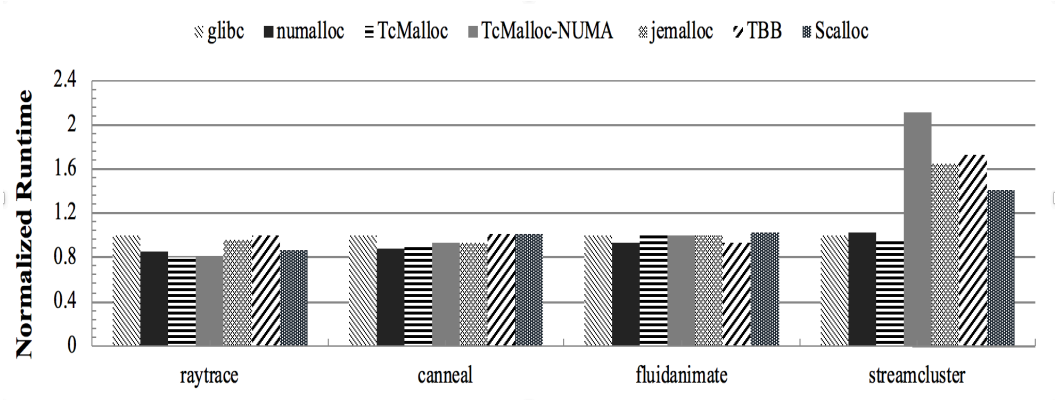

Fig. 9. Normalized runtime with different allocators for PARSEC benchmarks in Machine A after shut down interleaved heap

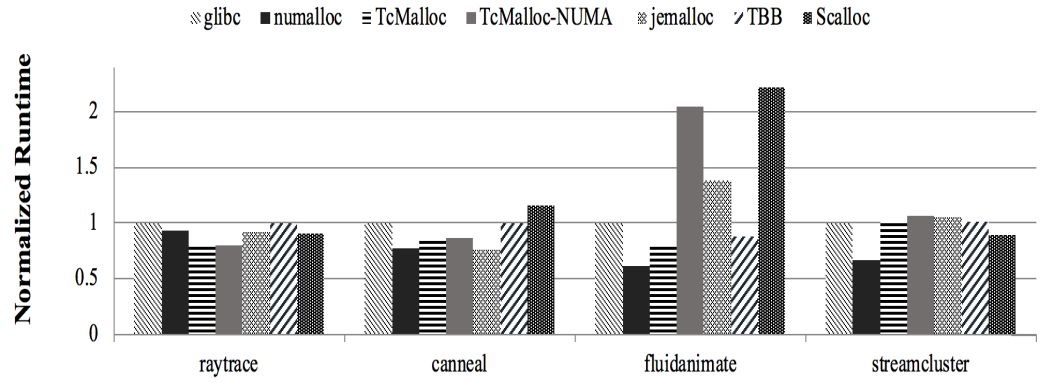

Fig. 10. Normalized runtime with different allocators for PARSEC benchmarks in Machine B after shut down interleaved heap

Finally, in figure 9 and figure 10, we show some performance results of some applications that got significant different values after we shut down interleaved heap for numalloc. We can see that for some applicatios with less data sharing between threads like ratrace and canneal, numalloc could got significant improvements due to its low overheads and proper memory management. But for some other applications with intensive memory operations and sharing like fluidanimate, shutting

down interleaved heap could hurt performance, since interleaved heap could help to distributed resource contention evenly over multi-nodes and then got low overheads.

4.2 Memory Evaluation

We also measured memory overheads of different allocators for same applications above. For parsec and Hoard benchmarks, the sum of maxresident output from time utility and the size of huge page usage is used,and huge page size could be gotten by periodically collecting /proc/meminfo file. Memory assumption of server applications like MySQL, apache and Memcached is collected by the sum of the VmHWM field and HugetlbPages field from /proc/PID/status file before we shutdownt the server.We always reboot server applications for each single test.

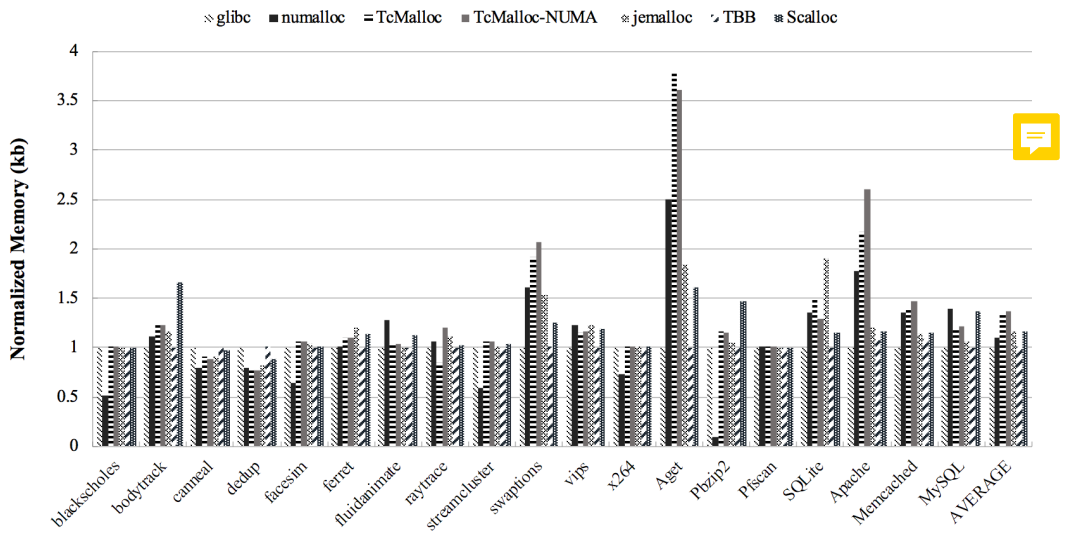

Fig. 11. Normalized memory overheads with different allocators for PARSEC benchmarks in Machine A

In figure 11, we give the normalized memory overheads also compared with Linux's glibc in Machine A. We can see that the average value of `numalloc` is 1.09, which means `numalloc` costs as same memory as Linux's glibc in average and `numalloc` is the best compared with others, although others also do not cost too much. In addition, for most of single application, `numalloc` costs as same memory as Linux's glibc or even less like 0.51 for blackscholes and 0.6 for facesim. The most memory consuming application in figure 11 is Aget ,in which the value of `numalloc` is 2.49, but it is still lower than TcMalloc and jemalloc and it is very acceptable.

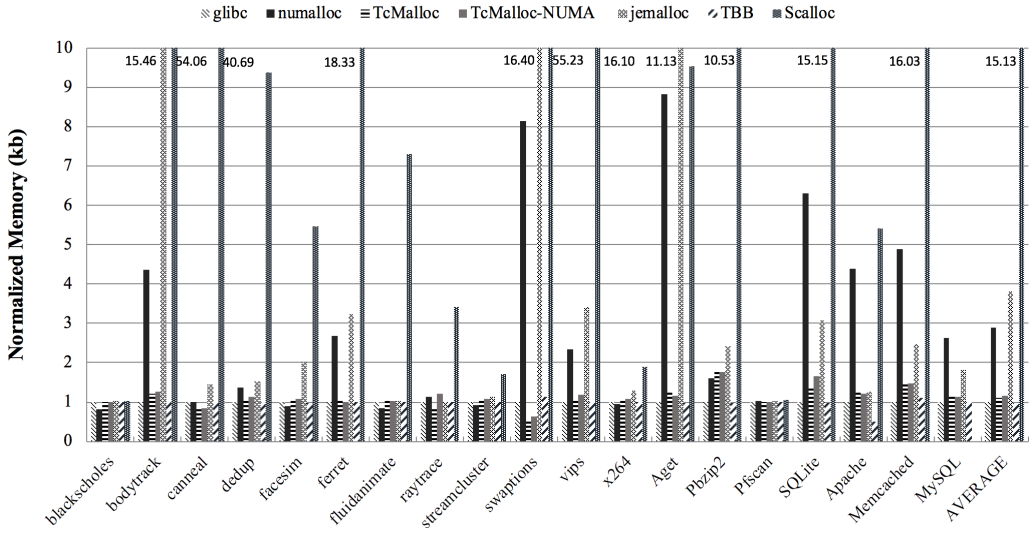

Fig. 12. Normalized memory overheads with different allocators for PARSEC benchmarks in Machine B

Figure 12 is the nurmalized memory overheads in Machine B. In this figure, we can see that some allocators get a high average nurmalized value compared with figure 11, that `numalloc` is 2.9, jemalloc is 3.7 and Scalloc is 15.1. The reason is that there are far more cores in Machine B which is 128 compare with Machine A which is 40. So these allocators will preserve more memories in their thread local area and also per node area in `numalloc`, that there are 8 nodes in Machine B and only 2 nodes in Machine A. But we can also see that in most applications like blackscholes, canneal and facesim etc, the memory overheads of `numalloc` are actually very low , equal or less than Linux's glibc. Compared with the its huge performance improvements, this memory overheads are negligible.

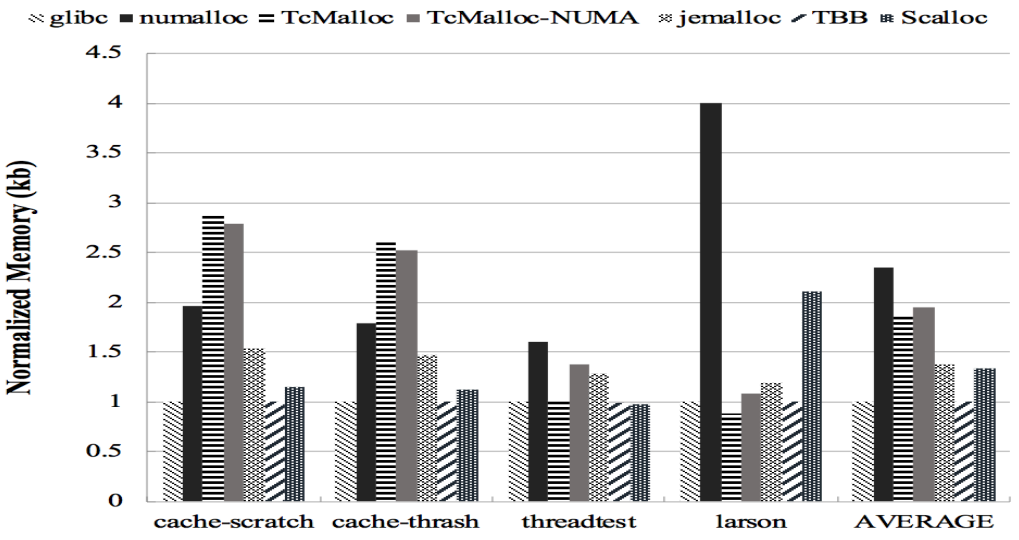

Fig. 13. Normalized memory overheads with different allocators for Hoard benchmarks in Machine A

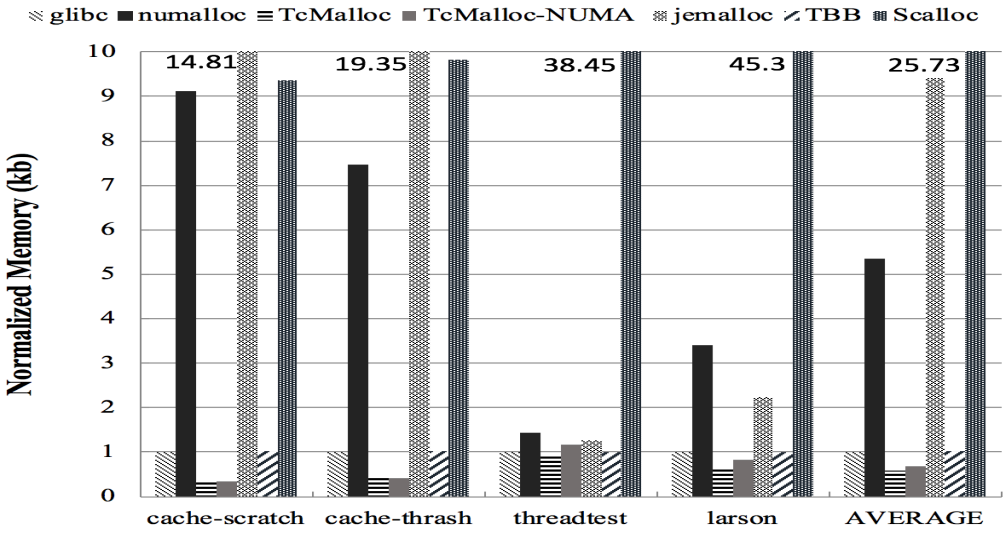

Fig. 14. Normalized memory overheads with different allocators for Hoard benchmarks in Machine B

Finally, in figure 13 and 14, the normalized memory overheads for Hoard benchmarks are given, and we can see that the average values of all allocators in figure 14 are bigger than figure 13. The reason is same as we mentioned above that there are more cores and more nodes in Machine B than Machine A, so that allocators usually preserve more memories in their thread local area or per node area. In figure 13, the average normalized value of `numalloc` is larger than others, but actually not too much, which is 2.3 for `numalloc`, 1.9 for TcMalloc-NUMA and 1.8 for TcMalloc. It is because that proper node management is utilized in `numalloc` and also in TcMalloc-NUMA, so that each node also preserves some memory not only thread locals.But we believe that this little more memory overheads are totally acceptable. It is also the same thing for figure 10, that the average value for `numalloc` is little higher than others, which is 5.3. But in this 8 nodes machine, numalloc is not the worst, that Scalloc's average value is 25 and jemalloc is 9.4. One main reason that the value of `numalloc` is smaller is that we use mini size bags in `numalloc` which is less than the size of one page for small objects and also memories for small objects are shared per node but per cores in Scalloc.

5 RELATED WORK

General Purpose Allocators:

NUMA-aware Allocators: Ogasawara focuses on finding the preferred node location for JAVA objects during the garbage collection and memory allocations [36], via thread stack, synchronization information, and object reference graph. The proposed method is not suitable for C/C++ applications, since the objects were tighten to physical pages.

[45] focuses on the specific scenario, which is in-memory databases.

[23]

[22] utilizes two mechanisms to support the NUMA architecture based on TcMalloc. First, it adds additional node-based freelists and free spans to store freed objects and pages belonging to the same node. Second, it also invokes the `mbind` system call to bind physical memory allocations to the node that the current thread is running on. However, it does not support huge pages and the

special allocations from the main thread, invokes too many `mbind` system calls, and does not handle the metadata's locality.

[33] proposes to set task-to-thread affinity, and pin threads to specific cores to achieve a better performance. It is not a memory management policy, and mainly talks about the detail implementation of task binding over different levels of the TBB library.

NumaGiC reduces remote accesses in garbage collection phases with a mostly-distributed design that each GC thread will mostly collect memory references locally, and utilize a work-stealing mode only when no local references are available [18].

[12] proposes to combine the task management and memory management to achieve the better performance. For task mapping, it proposes to place tasks that shared the data onto the cores that share the cache, in order to reduce the cache misses and the communication overhead. For memory management, it utilizes the page faults to analyze the memory access behavior, and then migrate pages between nodes to improve the performance. Basically, this is still a proactive approach, which cannot achieve the optimal performance promised by the hardware. Also, it requires the changes of the underlying OS, and will impose some overhead of understanding the communicating tasks and memory access behavior via analyzing page faults.

[44] Kaminski et al. proposes to make TCMalloc NUMA-aware [22], with the very minimum effort. The idea is to maximize the local memory allocation with the node-based free list and page heap. However, this mechanism assumes that the memory deallocation from the current node will be always allocated from the local node. This is unfortunately not true for many cases, such as a producer-consumer model. Also, this allocator does not take advantage of

[30] proposes to consider both data locality and cache contention, and combine memory management with task scheduling to achieve better performance. Mostly, it is focuses on the task scheduling.

We describe two scheduling algorithms: maximum-local, which optimizes for maximum data locality, and its extension, N-MASS, which reduces data locality to avoid the performance degradation caused by cache contention. N-MASS is fine-tuned to support memory management on NUMA-multicores and improves performance up to 32%, and 7% on average, over the default setup in current Linux implementations.

[7] proposes a new library that allows users to manage their memory in fine granularity by combining with multiple existing system calls. However, they are not targeting for a general purpose allocator, since it requires programmers to manage the memory explicitly. Unfortunately, this place unacceptable burden to programmers. More importantly, the explicit management based on one existing topology may not work well for the hardware with a different topology.

Other NUMA-related Systems: Memory system performance in a numa multicore multiprocessor

[33] proposes TBB-NUMA, a system that mainly focuses on task scheduling on NUMA architecture to achieve better performance. `numalloc` employs a similar mechanism as TBB-NUMA to manage threads explicitly, but without relying on the human hints. `numalloc` also deals with the memory allocation that is not presented in TBB-NUMA.

[32] shows that a set of simple algorithmic changes coupled with commonly available OS functionality suffice to eliminate data sharing and to regularize the memory access patterns for a subset of the PARSEC parallel benchmarks. These simple source-level changes result in performance improvements of up to 3.1X, but more importantly, they lead to a fairer and more accurate performance evaluation on NUMA-multicore systems. In the default configuration, OSs (such as Linux) tend to

change the thread-to-core mapping during the execution of programs, which result in large performance variations.

[6] talks about the virtualization of NUMA.

[14] introduces a round-robin fashion for arena allocation, and all locks and buffers will be local to the associated arena. Similarly, `numalloc` also takes the similar approach.

Scalloc utilizes the same-sized spans in order to encourage memory reuses [1], which is also the same for `numalloc`. Salloc has another two contributions, a global backend developed by concurrent data structures, and a constant-time front end that returns the spans to the backend.

Bolosky et. al. propose a simple mechanism to improve the performance by replicating read-only pages to multiple processors, moving pages to the processor that written them, and placing pages to the global memory if they are written by multiple processes [5]. bl

REFERENCES

[1] 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 (OOPSLA 2015)*. ACM, New York, NY, USA, 451–469. https://doi.org/10.1145/2814270.2814294

[2] 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*. ACM Press, New York, NY, USA, 117–128. https://doi.org/10.1145/378993.379232

[3] 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*.

[4] 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 (USENIXATC'11)*. USENIX Association, Berkeley, CA, USA, 1–1. http://dl.acm.org/citation.cfm?id=2002181.2002182

[5] 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)*. ACM, New York, NY, USA, 19–31. https://doi.org/10.1145/74850.74854

[6] Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. 2019. When eXtended Para - Virtualization (XPV) Meets NUMA. In *Proceedings of the Fourteenth EuroSys Conference 2019 (EuroSys '19)*. ACM, New York, NY, USA, Article 7, 15 pages. https://doi.org/10.1145/3302424.3303960

[7] 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).

[8] HyperTransport Consortium. [n. d.]. HyperTransport. https://en.wikipedia.org/wiki/HyperTransport.

[9] Jonathan Corbet. [n. d.]. AutoNUMA: the other approach to NUMA scheduling. https://lwn.net/Articles/488709/.

[10] 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 (ASPLOS '13)*. ACM, New York, NY, USA, 381–394. https://doi.org/10.1145/2451116.2451157

[11] SQL Developers. [n. d.]. How SQLite Is Tested. https://www.sqlite.org/testing.html.

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

[13] 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.

[14] Jason Evans. [n. d.]. Scalable memory allocation using jemalloc. https://www.facebook.com/notes/facebook-engineering/scalable-memory-allocation-using-jemalloc/480222803919/.

[15] The Apache Software Foundation. [n. d.]. ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[16] 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 (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 231–242. http://dl.acm.org/

citation.cfm?id=2643634.2643659

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

[18] 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 (ASPLOS '15)*. ACM, New York, NY, USA, 661–673. https://doi.org/10.1145/2694344.2694361

[19] Mel Gorman. [n. d.]. Huge pages. https://lwn.net/Articles/374424/.

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

[21] 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 (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 263–276. http://dl.acm.org/citation.cfm?id=2813767.2813787

[22] 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.

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

[24] 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 (USENIX ATC'12)*. USENIX Association, Berkeley, CA, USA, 5–5. http://dl.acm.org/citation.cfm?id=2342821.2342826

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

[26] 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 (USENIX ATC '15)*. USENIX Association, Berkeley, CA, USA, 277–289. http://dl.acm.org/citation.cfm?id=2813767.2813788

[27] Felix Xiaozhu Lin and Xu Liu. 2016. Memif: Towards Programming Heterogeneous Memory Asynchronously. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 369–383. https://doi.org/10.1145/2872362.2872401

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

[29] 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

[30] 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 (ISMM '11)*. ACM, New York, NY, USA, 11–20. https://doi.org/10.1145/1993478.1993481

[31] Zoltan Majo and Thomas R. Gross. 2011. Memory System Performance in a NUMA Multicore Multiprocessor. In *Proceedings of the 4th Annual International Conference on Systems and Storage (SYSTOR '11)*. ACM, New York, NY, USA, Article 12, 10 pages. https://doi.org/10.1145/1987816.1987832

[32] 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

[33] 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 (PPoPP 2015)*. ACM, New York, NY, USA, 227–238. https://doi.org/10.1145/2688500.2688509

[34] Zoltan Majo and Thomas R. Gross. 2017. A Library for Portable and Composable Data Locality Optimizations for NUMA Systems. *ACM Trans. Parallel Comput.* 3, 4, Article 20 (March 2017), 32 pages. https://doi.org/10.1145/3040222

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

[36] 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 (OOPSLA '09)*. ACM, New York, NY, USA, 377–390. https://doi.org/10.1145/1640089.1640117

[37] 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

[38] Sean Reifschneider. [n. d.]. "Pure python memcached client". https://pypi.python.org/pypi/python-memcached.

[39] 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

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

[41] 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

[42] Christian Terboven, Dirk Schmidl, Tim Cramer, and Dieter an Mey. 2012. Assessing OpenMP tasking implementations on NUMA architectures. In *International Workshop on OpenMP*. Springer, 182–195.

[43] Christian Terboven, Dirk Schmidl, Tim Cramer, and Dieter an Mey. 2012. Task-parallel programming on NUMA architectures. In *European Conference on Parallel Processing*. Springer, 638–649.

[44] 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

[45] 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.

[46] 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