NUCAlloc: Fine-Grained Block Placement in Hashed Last-Level NUCA Caches

Raveendra Soori
Cloud Software Group

Shreyas Prabhu
Apple

Harpreet Singh Chawla
Texas A&M University

Michael Ferdman
Stony Brook University

ABSTRACT

Modern last-level caches are partitioned into slices that are spread across the chip, giving rise to varying access latencies dictated by the physical location of the accessing core and the cache slice being accessed. Although, prior work has shown that dynamically determining the best location for blocks within such Non-Uniform Cache Access architectures can provide significant performance benefits, current hardware does not implement this functionality. Instead, modern processors hash blocks across the LLC slices, obscuring the non-uniform architecture of the underlying cache and forfeiting the performance benefits of placing data in the nearest cache slices. Moreover, while prior work advocated improving performance by delegating control over block placement to the operating system at page granularity, modern processor hardware thwarts these approaches by hashing cache slice selection at cache block granularity.

In this work, we make two observations that enable us to improve software performance on modern NUCA architectures. First, we find that software can undo the hashing performed by hardware and efficiently manage data placement at cache block granularity. Second, that the complexity of fine-grained data placement can be hidden from the developer by embedding it in the dynamic memory allocator. Leveraging these observations, we design a new specialized memory allocator, *NUCAlloc*, suitable for use with C++ containers such as *std::map* and *std::set*. *NUCAlloc* handles the complexity of NUCA-aware block placement, improving the performance of containers by placing their data into the nearest LLC slices. We demonstrate that our *NUCAlloc* prototype consistently outperforms *std::allocator* and *jemalloc* for LLC-resident containers, improving performance by up to 20% in both single-threaded and multi-threaded software.

CCS CONCEPTS

• **Computer systems organization**; • **Software and its engineering** → **Software notations and tools**;

KEYWORDS

NUCA, memory allocation, micro-architecture, performance

ACM Reference Format:
Raveendra Soori, Shreyas Prabhu, Harpreet Singh Chawla, and Michael Ferdman. 2024. NUCAlloc: Fine-Grained Block Placement in Hashed Last-Level NUCA Caches. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3650200.3656604

1 INTRODUCTION

In modern processors, the last-level cache (LLC) is shared by the cores and is typically organized as a Non-Uniform Cache Architecture (NUCA). The cache is organized into partitions, or slices, which are physically located on the chip at varying distances from the cores. This varying proximity to cores causes non-uniform cache access latencies when a core accesses data in the LLC, with the latency depending on both which core is performing the access and which cache slice is receiving it. For any given core, accessing data in the cache slice closest to that core leads to the lowest access latency, and consequently results in higher software performance.

Extensive research has explored hardware techniques to dynamically place cache blocks used by a core in the best-performing (closest) locations in NUCA caches [1, 4–6, 11, 12, 20, 22, 28, 32]. However, modern hardware does not implement these hardware NUCA mechanisms. Instead, the hardware uses a hash function to uniformly randomize which slice handles a given cache block, sacrificing the latency benefits of using an LLC slice nearest to the accessing core.

Another class of works on NUCA caches proposes delegating block placement decisions to software [5, 27].

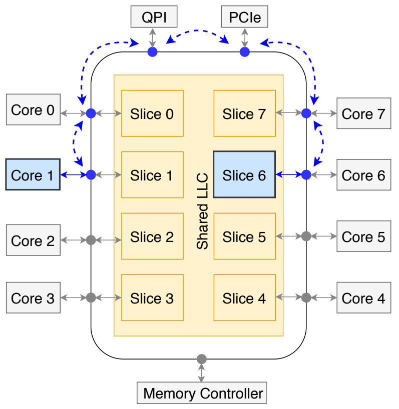

Figure 1: NUCA cache with eight slices. Dashed lines show interconnect hops when Core 1 accesses Slice 6.

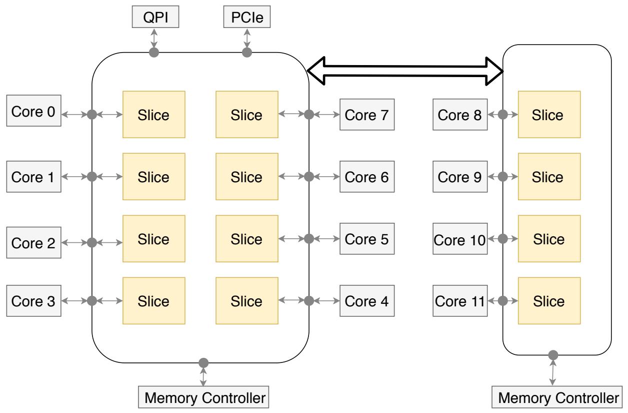

Figure 2: Intel 12-core NUCA processor with two interconnect rings.

These techniques perform block placement by leveraging the virtual memory system at page granularity, using the operating system to manipulate the page tables and TLBs to steer accesses to their desired cache slice. Unfortunately, modern processors preclude the use of these software techniques. The hardware placement hash function operates at block granularity, which uniformly spreads the blocks belonging to every memory page across all cache slices. Our work overcomes this hardware limitation, improving cache access latency in modern processors by performing fine-grained NUCA-aware block placement, despite the presence of the hardware hash function.

In this work, we make two key observations. First, we find that, although the block placement hash function used by the hardware is officially undisclosed, it has been successfully reverse engineered [13, 14, 18, 24, 31]. Second, we observe that user-level software can transparently manage NUCA block placement at cache block granularity by using a carefully-crafted memory allocator. Consequently, our approach unlocks the performance benefits of latency-aware NUCA block placement on modern off-the-shelf processors.

We leverage these observations to design *NUCAlloc*, a specialized allocator that transparently handles the complexity of NUCA-aware block placement. *NUCAlloc* works by determining the cache slice a memory block belongs to by applying the hardware hash function in reverse and placing the block on the free-list of the corresponding slice. Allocations take elements from the free-list of the desired slice, prioritizing the closest slice, and allocate from the other (farther away) cache slices as more memory is requested. Freed elements are returned to the free-list from which they were allocated. We make the *NUCAlloc* interface compatible with C++ containers, allowing it to be used transparently to improve the performance of accessing the data and the meta-data (e.g., pointers) of *std::map*, *std::set*, and other standard library data structures.

To demonstrate the effectiveness of *NUCAlloc*, we compared its performance with *glibc malloc* from Linux and *jemalloc* from FreeBSD on two generations of Intel server processors. We evaluate performance for single-threaded and multi-threaded applications. Our results show that *NUCAlloc* effectively leverages NUCA latencies, outperforming state-of-the-art allocators by up to 20%.

2 MODERN NUCA CACHES

Modern processors implement their last-level caches (LLCs) following a Non-Uniform Cache Access (NUCA) architecture. Figure 1 shows a high-level diagram of an example processor, an Intel E5-2650. In this processor, the shared LLC is divided into eight slices, with the eight cores using an on-chip ring interconnect to access the cache slices. In this organization, each core has a *local* LLC slice that is nearest to it, which can be accessed with minimal latency. Unfortunately, the latency of accessing the other slices can be significantly higher, as it depends on the number of hops that accesses must travel, along the on-chip ring interconnect.

Figure 2 shows the organization of A 12-core Intel E5-2670v3 processor has two rings, with the rings interconnected using a bridge which introduces additional latency to the accesses that traverse it. Moreover, the LLC access latency observed by the cores on the second ring is higher, owing to an unequal distribution of slices among the rings. In general, with an increase in the number of cores on the chip, the access latency difference among the slices also increases.

Although research on mechanisms to manage block placement within NUCA caches has shown promising results, modern processors implement a simple policy that randomly distributes cache blocks across all NUCA slices. To implement the random block distribution, when accessing data in the LLC, the physical address of the memory block is hashed; a subset of the bits from the physical address are XOR-ed together to compute values which decide the cache slice index.

Critically, because some low-order bits (just above the block offset) participate in the hash, cache blocks whose addresses are adjacent in the physical address space are always placed in different cache slices. Because any given cache block can be randomly placed in the closest or farthest slice, the LLC access latency observed by software is the average access latency of all slices. The hash function is not publicly disclosed, but security researchers seeking to study and manage conflicts in the shared LLC were able to reverse engineer it by observing the uncore performance counters [24]. Once the hash function is known, it is possible to determine the slice mapping of any block, potentially allowing control over the placement of data by allocating them at an appropriate physical address.

The pattern of which blocks within a page are located on which slice and the proximity of that slice to a particular core changes with the physical address of the page and the ID of the core performing the access. As a result, the complexity of manually performing fine-grained management of memory that could take advantage of the NUCA organization is unreasonable even for an extremely performance-conscious software developer. However, the implementation complexity can be hidden inside a dynamic memory allocator, and the performance cost of performing the block mapping can be shifted to the initialization of the allocator free-lists.

Notably, because blocks with adjacent physical addresses are mapped by the hardware to different cache slices, large objects cannot have all of their blocks mapped to cache slices with low latency. However, software using pointer-based data structures with large collections of small objects, such as C++ programs using standard containers, can have all of the container data (and the container meta-data) reside in low-latency nearby cache slices. We therefore develop *NUCAlloc* as a standard C++ allocator that can be used as a template argument for C++ containers such as *std::map* and *std::set*, transparently endowing these containers with the benefits of NUCA-aware placement and low access latencies.

3 *NUCALLOC* DESIGN

In this section we present the methodology and the design rationale behind *NUCAlloc*. We start by discussing how the hash function is recovered in Section 3.1, we then divide our discussion about design of *NUCAlloc* into Section 3.2 which presents a high level overview and Section 3.3 which presents lower level implementation details.

3.1 Recovering the Hash Function

Modern NUCA processors use a hash function to spread data among the LLC slices. By knowing the hardware hash function, software can determine the slice that an address maps to. The practicality of cache side-channel attacks such as prime and probe, and the ability to target particular slices of an LLC has attracted many security researchers to attempt to reverse engineer the LLC hash function [13, 14, 18, 24, 31].

Component Functions:

$$f_0 = b_{11} \oplus b_{16} \oplus b_{17} \oplus b_{21} \oplus b_{23} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{29} \oplus b_{31}$$

$$f_1 = b_9 \oplus b_{13} \oplus b_{14} \oplus b_{17} \oplus b_{18} \oplus b_{19} \oplus b_{20} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{25} \oplus b_{27} \oplus b_{30} \oplus b_{31}$$

$$f_2 = b_7 \oplus b_{12} \oplus b_{13} \oplus b_{17} \oplus b_{19} \oplus b_{22} \oplus b_{23} \oplus b_{24} \oplus b_{25} \oplus b_{27}$$

$$f_3 = b_6 \oplus b_{11} \oplus b_{12} \oplus b_{16} \oplus b_{18} \oplus b_{21} \oplus b_{22} \oplus b_{23} \oplus b_{24} \oplus b_{26} \oplus b_{30} \oplus b_{31} \oplus b_{32}$$

$$f_4 = b_8 \oplus b_{13} \oplus b_{14} \oplus b_{18} \oplus b_{20} \oplus b_{23} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{28} \oplus b_{32} \oplus b_{33} \oplus b_{34}$$

$$f_5 = b_9 \oplus b_{14} \oplus b_{15} \oplus b_{19} \oplus b_{21} \oplus b_{24} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{29} \oplus b_{33} \oplus b_{34}$$

$$f_6 = b_{10} \oplus b_{15} \oplus b_{16} \oplus b_{20} \oplus b_{22} \oplus b_{25} \oplus b_{26} \oplus b_{27} \oplus b_{28} \oplus b_{30} \oplus b_{34}$$

$$f_7 = b_6 \oplus b_7 \oplus b_8 \oplus b_9 \oplus b_{10} \oplus b_{12} \oplus b_{13} \oplus b_{14} \oplus b_{15} \oplus b_{18} \oplus b_{19} \oplus b_{20} \oplus b_{22} \oplus b_{24} \oplus b_{25} \oplus b_{30} \oplus b_{32} \oplus b_{33} \oplus b_{34}$$

Final Hash Function:

$$s_3 = f_3 \cdot [f_4 + f_5 \cdot (\neg f_6 + \neg f_7)]$$

$$s_2 = \neg s_3 \cdot f_2$$

$$s_1 = f_1$$

$$s_0 = f_0$$

Figure 3: The reverse-engineered Intel E5-2670v3 hash function. b_i corresponds to ith bit of the physical address and s_i corresponds to ith bit of the slice number.

Maurice, et al. [24] presented a technique to find the hash function used by Intel processors with 2, 4 and 8 cores. They noted that the hash function depends only on the core count and that the same hash function is used across different Intel processor generations. However, their method to derive the hash function is restricted to processors with 2^n cores as they assume the hash function only uses XOR of the address bits.

Yarom et al. [31] reverse engineered a 6-core hash and demonstrated that it comprises *component functions*. Each component is formed with XOR of some physical address bits, with the final hash function using operations other than XOR. We verified the 8-core hash function from the prior work [24] and reverse engineered the 12-core hash function.

To find the hash function of the 12-core processor, we followed a similar approach to Maurice, et al.[24], polling each cache block to collect data about the slice mapping. The uncore CBo counters monitor the `LLC_LOOKUP` event. An address is polled by flushing that address from all cache levels using the `clflush` instruction. `clflush` causes a lookup event in the LLC irrespective of whether the flushed address is cached or not. Repeated polling of the address will reflect in the CBo counters with a high value for the counter associated with the correct slice. This gives us a mapping of virtual addresses to the slice numbers.

We use `/proc/self/pagemap`, a Linux kernel interface that allows a process to determine the physical frame to virtual page mapping [29], to translate the virtual addresses

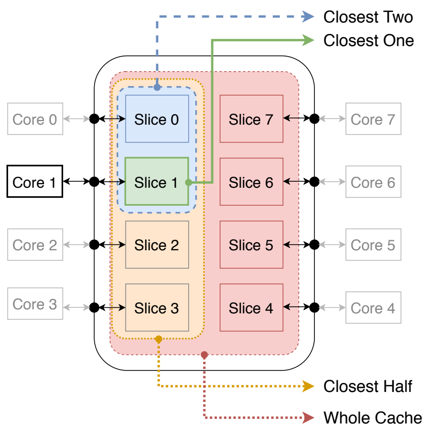

Figure 4: Illustration of the LLC regions used by the various block placement policies. In this figure the thread is running on Core 1.

to physical addresses. We then derive the component functions by processing an extensive list of physical addresses and their corresponding slice numbers.

Figure 3 shows the component functions and the hash functions for a 12-core processor. This function bears resemblance to the hash function recovered by Yarom, et al. [31], but the actual bits in the component functions differ. Additionally, we also have an additional bit in the hash function owing to the higher core count. The hash function used by a 8-core processor is much simpler because each slice bit corresponds to a single component function.

Following this approach, one can derive the LLC hash function of any Intel CPU. Leveraging the knowledge of the hash function, we can design an allocator which allocates memory such that blocks are placed in the slice that is in the proximity of the core requesting memory.

3.2 *NUCAlloc* Design Overview

NUCAlloc utilizes the reverse engineered hash function discussed in Section 3.1 to achieve fine-grained block placement on the LLC. Fine-grained block placement can follow a number of different strategies: Allocating the blocks such that they always correspond to the closest slice (*Closest One*), allocating such that the blocks are interleaved among the closest and the neighboring slice (*Closest Two*), interleave on the same side of the ring as the core on which the thread is running (*Closest Half*) or utilize the whole cache. Figure 4 shows that each of these strategies restricts the memory blocks to a portion of the LLC. Eventually, new accesses evict those allocated earlier. The threshold at which evictions occur is a crucial parameter that determines our allocation strategy. Notably, slice capacity is not the only contributing factor, but the associativity of the LLC also plays a crucial role, with conflict misses lowering the threshold.

Each policy has benefits and drawbacks. For example, *Closest One* always allocates memory corresponding to the fastest (closest) slice. However, if the aggregate memory allocated with *Closest One* exceeds the slice capacity, successive allocations will conflict in the LLC. Thus, *Closest One* suffers from poor LLC utilization, but guarantees the best hit latency. *Closest Half* offers better LLC utilization, doing so at the cost of higher hit latency compared to *Closest One*. In all cases, the allocation must be done in conjunction with software thread management, requesting for the OS to pin software threads to the core, or group of cores, within the region where the corresponding memory allocations are placed.

When determining the default policy for *NUCAlloc*, we aimed at achieving lower latency benefits of *Closest One* without suffering LLC utilization penalty. Because each slice of LLC is at a different distance from the core, the access latency of each slice is different. The closest slice sees the least hit latency while the farthest sees the highest latency. The rest of the slices range from the least latency to the highest latency. Hence, it is beneficial to allocate memory in a dynamic strategy which prioritizes slices closer to the core and expands out to the higher latency slices. *NUCAlloc* follows a dynamic strategy by tracking the amount of memory allocated and selecting the appropriate slice ensuring the lower latency benefits of *Closest One* without sacrificing LLC utilization.

An important point to note here is that since *Closest One*, *Closest Two* and *Closest Half* always allocate memory based on their policy even if their allocations exceed the cache regions that they target, they can be used as a LLC isolation method. These policies can be useful in case of multi-threaded applications where we might want to restrict a thread to a section of LLC preventing it from polluting other threads' data. As such, *NUCAlloc* allows the application to override the policy it follows to one of these strategies.

3.3 Implementation Details

To facilitate NUCA-aware allocation, *NUCAlloc* maintains per-slice free-lists, populated with memory blocks that map to the corresponding cache slice. In our target systems, each cache slice corresponds to a core, making per-slice free-lists similar to the per-core free-lists of modern allocators. To populate the free-lists, *NUCAlloc* requests a memory page from the OS using `mmap`, splits that page into cache-block sized free blocks, and places each block on its corresponding per-slice free-list. To determine on which free list a block belongs, *NUCAlloc* computes the slice index of the block based on the offset of that block within the page, the hardware hash function, and the virtual-to-physical address translation.

The reverse-engineered component functions are shown in Figure 3. The component functions XOR subsets of the address bits and are combined together to yield the

```
populate_freelists():
  # Component functions of first block
  first_cf = { f[8] = {0} }

  page_va = mmap(...)
  page_pa = va_to_pa(page_va)

  for i = 0 to 7:
      first_cf.f[i] = f(i,page_pa)

  for i=0 to CACHE_BLOCKS_PER_PAGE-1:
      curr  = { f[8] = {0} }
      offset =  i * CACHE_BLOCK_SIZE
      for i = 0 to 7:
          curr.f[i] = first_cf.f[i] ^ f(i,offset)
      slice_num = get_slice(curr)
      block_va = page_va + offset
      add_to_freelist(block_va,slice_num)
```

Figure 5: Pseudo-code for populating the free-lists for a chip that uses eight component functions. The component function values for the blocks in the page are derived from the first block's component functions and the block's offset.

final hash function used to translate physical addresses to LLC slice numbers. Notably, because the hash function operates on the physical rather than the virtual address, *NUCAlloc* must use the physical addresses of the blocks. We use the address mapping provided by the Linux kernel via the `/proc/self/pagemap` to find the physical address corresponding to the virtual address of the allocation. Once the physical address of the page is known, the values of the component functions corresponding to the first 64-byte memory block in the page are determined using the reverse-engineered functions. Critically, to avoid repeatedly performing this costly computation, which includes complex component functions and the virtual to physical translation, we use a small meta-data hash table that maps the allocated virtual address to the component function values of the first block of the corresponding page. The slice number of any block within a page can be quickly computed based on the component function values of the first block within that page and the offset of the block within the page. Figure 5 shows the pseudo-code for populating the free-lists using this logic.

The process of mapping a page and populating the free-lists occurs each time we run out of memory blocks on a desired free-list. The page obtained from the OS is used to extend all of the per-slice free-lists. The meta-data required to maintain the free-lists is stored within the blocks themselves, requiring no additional memory overhead for maintaining the free-lists.

Memory is allocated to an object by first determining the cache slice and then returning a block from the corresponding free-list. Because this involves only removing a node from a pre-populated free-list, allocation involves little overhead. To make sure that the software threads remain running on the cores closest to their memory allocations, the threads are pinned to the cores by the OS, ensuring that allocations made from the associated free-list will remain close to the core that will be accessing them.

During de-allocation, the address of the block being freed is used to calculate the virtual address of the page storing the block. This address is used to look up the component function values of the first 64-byte memory block of that page in the hash table. In combination with the offset of the block being freed, the component function values allow us to calculate the slice number of the block being freed. Once the slice number is known, the block is added to the corresponding free-list.

The slice selection during allocation is determined as follows: Initially, memory is allocated such that it corresponds to the slice that is closest to the core that the thread is running on. On reaching the threshold of the slice - which is an empirically arrived value, we start allocating blocks corresponding to the neighboring slice. Upon reaching the threshold of the neighboring slice, we allocate blocks corresponding to next closest slice and so on. This ensures least cache access latency while maximizing LLC utilization. To make this selection possible, we store a priority ordering of the slices for each core in the allocator, this varies depending on the processor type. We demonstrate how to determine the slice ordering and the threshold parameter in Section 4.

When an application involving multiple threads needs to make allocations, *NUCAlloc* restricts the slice selection to a portion of LLC to prevent the threads from thrashing one another. For instance, when the application involves 4 threads making allocations on an eight core processor, *NUCAlloc* allocates memory from only the closest 2 slices for each of the threads. The assumption here is that each of the thread on average, makes use of the same amount of LLC data. By restricting their private data to the closer slices we gain in performance compared to an allocator which would interleave data among all the slices which increases the chance of one thread thrashing the other's LLC data. Because the free-lists for each core are mutually exclusive, allocation does not require locks.

3.4 *NUCAlloc* Applicability

NUCAlloc is designed for data structures whose elements are smaller than or equal to the size of a cache block and whose working set is small relative to the LLC capacity. By allocating just such data structures using *NUCAlloc*, allocated elements are placed in the cache slices closest to the cores from which they will be accessed, minimizing access latency. The biggest benefits would be seen for pointer-based data structures such as graphs, trees, and lists, as *NUCAlloc* improves latency without disrupting prefetcher accuracy for such data structures. Regular data structures (such as arrays or heaps) still experience *NUCAlloc* latency benefits, but may also be harmed by less effective prefetching into the L1 from the LLC.

Notably, *NUCAlloc* is not meant to replace the regular memory allocator, but should be used alongside it. The data structures that benefit from NUCA-aware placement should be allocated using *NUCAlloc*, while other allocations should use the regular general-purpose allocator. This selective allocation strategy ensures that the benefits of *NUCAlloc* are realized without adversely affecting the rest of the application.

For single-threaded applications, or applications not using all of the available cores, *NUCAlloc* will work best until the allocation capacity exceeds the size of one LLC cache slice. Beyond this point, *NUCAlloc* will continue to operate, but will begin placing blocks in cache slices that are farther away, gradually diminishing the latency benefits. If the allocated working set size is large and spans more than half of the cache slices, the system performance will match or fall below a regular memory allocator, because second-order effects, such as increased TLB pressure and lower prefetcher accuracy, may begin to have a negative impact.

For multi-threaded applications, *NUCAlloc* will work best when the data structures allocated with it are private to each thread, resulting in maximum locality for all threads and minimal interference among threads. *NUCAlloc* will benefit both LLC hits and LLC misses. LLC hits will observe a significantly lower access latency to the cache blocks fetched into the L1 from the local LLC slice. LLC misses will observe a reduced load-to-use time because the misses are discovered more quickly, allowing memory requests to be issued earlier.

Finally, in multi-threaded applications with significant imbalance in the working set of different threads, *NUCAlloc* may provide limited benefits. Threads whose data structures fit into their local LLC slices will observe improvement, but threads with larger working sets will be adversely affected because they will not have access to the full cache capacity. Although it is possible to use the single-threaded *NUCAlloc* strategy to place allocations into neighboring cache slices for such cases, a careful analysis of the cache miss behavior would be required to determine if the use of *NUCAlloc* would still be beneficial.

4 EVALUATION

We evaluate *NUCAlloc* on an 8-core processor a 12-core processor, organized as shown in Figure 1 and Figure 2, respectively, where each core is associated with a 2.5MB cache slice. Table 1 provides the processor details.

To evaluate the performance of data structures when using *NUCAlloc*, we construct benchmarks based on the C++ standard template library (STL) containers, using a template parameter to specify which custom allocator the container should use. Our benchmarks populate the container under test with 64-byte elements containing random data and then measure the time to perform a pseudo-random sequence of accesses to the container. The same pseudo-random sequence is used when testing all allocators. To minimize the impact of cold-start effects, the access sequence touches each element in the container 100 times.

Table 1: Processor configurations used for evaluation.

	Intel E5-2650	Intel E5-2670v3
Core count	8	12
Threads per core	1	2
L1D	32 KB (8 way)	32 KB (8 way)
L2	256 KB (8 way)	256 KB (8 way)
LLC	20 MB (20 way)	30 MB (20 way)

To study the in-depth behavior of the allocators, we instrument our benchmarks with calls to `perf`, providing exact counts of clock cycles as well as cache hits and misses in each level of the on-chip cache hierarchy. For cache hit and miss information from specific LLC slices, we used the Intel uncore CBo counters[15, 16]. These CBo counters were configured for the `LLC_LOOKUP` event and then filtered based on the FMESI states to isolate hits and misses.

Although many container types are available in C++, some of the most-commonly used ones (e.g., *std::map*, *std::multimap*, *std::set*, *std::multiset*) are built around a red-black tree data structure [3], which stores a node with pointers to the left child, right child, and parent nodes for each element inserted into the container. Due to the inherent similarity between the containers and their behavior, we primarily focus our evaluation on *std::map*, and only briefly present results for the other container types in Section 4.6

We compare the performance of accessing LLC resident data structures for *NUCAlloc* with *std::allocator* and *jemalloc*. *std::allocator* is the default allocator used by modern Linux systems (it is used for C++ containers when the allocator template parameter is omitted). *jemalloc* is a state-of-the-art high-performance memory allocator, originally from FreeBSD, which was designed for efficient operation in multi-threaded environments [9]. Notably, *std::allocator* uses in-band meta-data, wherein the meta-data are stored next to the allocated data, while *jemalloc* stores meta-data separately, making the allocated data contiguous in memory. By its nature, *NUCAlloc* ensures cache-line alignment of its allocations. To ensure a fair comparison, we make sure that the allocation done by each allocator is aligned to the cache lines. Because we craft each allocation request to be 64 bytes in size, *jemalloc* naturally benefits from cache-line aligned allocations because it allocates memory in contiguous addresses. In case of *std::allocator* where this isn't the case, we craft the allocations to be such that every allocation starts at the cache-line boundary.

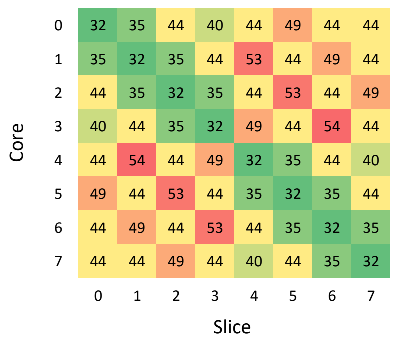

Figure 6: Slice access latency from different CPU cores measured in clock cycles for the 8-core processor.

We use 2MB huge pages to ensure that the performance gained by *NUCAlloc* is not obscured by TLB miss penalties. The rationale behind using huge pages is not to gain performance from fewer TLB accesses but to make the *NUCAlloc* gains transparent to the application. Because *NUCAlloc* makes allocations pertaining to the closest slice first, it would naturally require more pages than a NUCA unaware allocator which would repeatedly make allocations from a given page. However, *NUCAlloc* does not waste any segmented blocks of memory instead storing them in slice specific freelists from which subsequent allocations would be made once the threshold of the closer slices fill up. As such the number of pages used by both *NUCAlloc* and a NUCA unaware allocator would match as the LLC capacity fills up. We ensure that fewer TLB lookups aren't the cause of performance gains, we configure both *jemalloc* and *std::allocator* to make use of 2 MB huge pages as well. To avoid micro-architectural noise from hardware prefetchers, our micro-benchmark results are collected with hardware prefetchers disabled.

4.1 NUCA Hit Latency Analysis

The hardware random hashing needs to be undone to determine the slice number of a given block. Our target is to utilize the NUCA nature of modern hardware by designing a specialized memory allocator, which would place the data onto appropriate cache slices at a finer (cache block) granularity. To check the feasibility of this approach, we need to test whether it is possible to lower the LLC access latency through micro-managed data placement on the LLC.

We design an experiment by creating a linked list of 16,000 64-byte nodes. The total allocation (~1MB) can fit in one LLC slice (2.5 MB), but not in the local caches (Capacity of L1D + L2 = 288 KB). We walk through this list, 5000 times and measure the clock cycles taken to perform the accesses. The thread that performs the accesses is pinned to a particular core. We also set up the CBo counters to monitor the `LLC_LOOKUP` event to determine the slices being accessed.

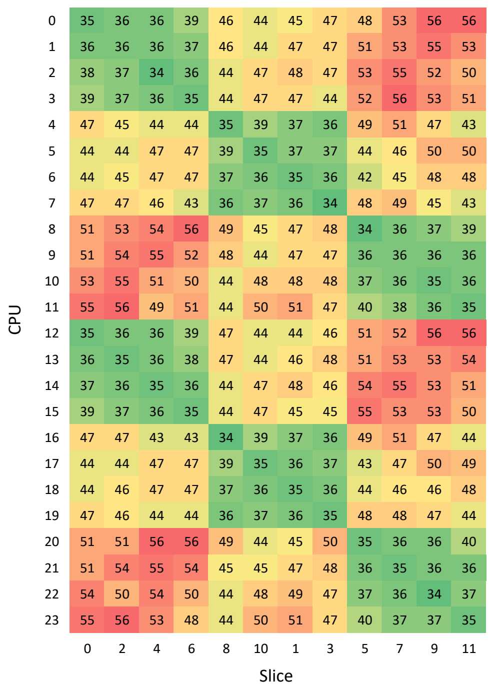

Figure 7: Slice access latency from different CPU cores measured in clock cycles for the 12-core processor.

We use the reverse engineered hash function to generate a pool of 64-byte blocks corresponding to a particular slice. We then allocate memory for the linked list nodes to evaluate the access latency of each slice. Because our access pattern ensures that most accesses miss the L1D and L2 caches, we measure the latency of accessing an LLC slice as the total cycles divided by the number of LLC accesses. We repeat this experiment for many combinations of slices and cores. The results for the 8-core machine and the 12-core machine are shown as a heat map in Figure 6 and Figure 7 respectively.

Single balanced ring. The hit latency heatmap of the 8-core processor, depicted in Figure 6, shows two red regions in the bottom left quarter and top right quarters. These correspond to slices present on the opposite side of the processor (Figure 1). The green line is along the diagonal from the bottom left to the top right, corresponding to the closest slice. Additionally, each core has at least two slices which can be accessed with latency under 36 clock cycles. These observations serve as motivation to design various strategies of NUCA-aware allocations.

We repeat the experiment to compare our NUCA-aware allocator, always allocating memory on the closest slice, to a NUCA-oblivious allocator that uses the *posix_memalign* function to allocate memory for the nodes of the linked list. The rationale behind using *posix_memalign* is to ensure that the NUCA-oblivious allocations are cache-line aligned and the comparisons are fair. For this experiment, we pin

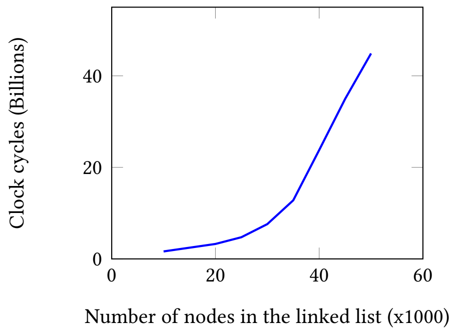

Figure 8: Single slice latency. Clock cycles for accessing each node of the linked list 5000 times.

the thread on Core 0 and make NUCA-aware allocations corresponding to slice 0 with respect to Figure 1.

We observed that NUCA-aware allocations benefited from 24% faster accesses to the elements of the linked list. We ascertained that the speedup is due to LLC block placement by making sure the L1D and L2 hit and miss values are similar for both allocations. The LLC miss values were also verified to be similar for both allocations. Further, the CBo counters confirm that NUCA-aware allocator made allocations that map to slice 0, while the NUCA-oblivious allocations were spread across the LLC slices. The total LLC accesses for each of the techniques is about the same, establishing the fact that the speedup observed is from the block placement in the LLC.

Two unbalanced rings. The heatmap of the 12-core processor is shown in Figure 7. A 12-core processor supports two threads per core, having 24 logical CPUs. Each thread running on a core sees the same hit latency for a particular slice; for instance, CPU 0 and CPU 12 have similar hit latency values for different slices. The processor is organized as shown in Figure 2. Notably, the processor cores are arranged in two rings. Examining the heatmap for this processor (omitted due to space limitations), we find that CPUs 0-3 have the least access latency to the slices on their side of the ring. Slices on the other side of the same ring see moderate hit latency. As expected, the slices on the second ring have the worst latency. CPUs 4-7, on average, observe the lowest LLC hit latency, because they are flanked by 4 cores on either side. Also as expected, the slices that are on the same ring have lower hit latency compared to the slices on the other ring. Specifically, CPUs 8-11 experience the worst average LLC hit latency, because they have only 4 slices in the same ring as the core. Running an experiment that accesses LLC data allocated using a NUCA-oblivious allocator shows that the average LLC hit latency in the second ring is approximately 5% higher than the cores in the middle of the processor (CPUs 4-7).

The LLC access latencies of the processors and the CBo counters confirm the expected behavior in a NUCA processor supporting the two motivations of this work. Modern

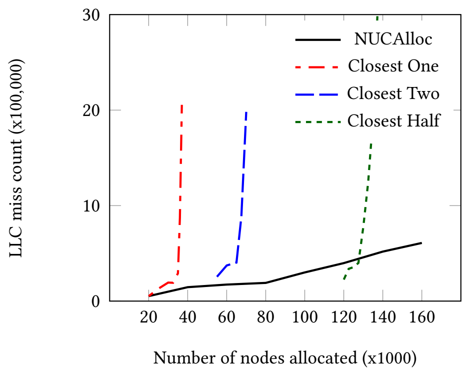

Figure 9: LLC misses for different allocation strategies.

processors use NUCA caches, but spread blocks across the LLC, effectively hiding the NUCA behavior. Reverse engineering the hash functions allows us to determine the physical address and slice mapping, granting us control over block placement. We therefore conclude that it is indeed possible to control block placement at a fine granularity to achieve lower latencies on LLC accesses and set out to build an allocator that leverages these observations.

4.2 Effective Capacity of LLC Slices

We next find the threshold at which allocating on a single slice begins to exhibit performance degradation. This threshold is necessary to determine the exact point at which *NUCAlloc* must switch from one slice to the next. To find this threshold, we repeat the earlier experiment involving a linked list, this time varying the number of elements in the list. The results are shown in Figure 8.

Because our working set is larger than the capacities of L1D and L2 combined, almost every access goes to the LLC. A fully associative LLC slice would theoretically hold up to 40,960 elements. However, the LLC we study is 20-way set-associative and we begin to see conflicts earlier. As seen in Figure 8, the latency sharply increases beyond ~35,000 elements. There is a minor change in slope starting from ~25,000 elements. Noting these observations, we set the threshold at 30,000 allocations.

To confirm that the threshold prevents LLC misses that would negatively impact access latency, we measure the LLC miss counts for *Closest One*, *Closest Two*, and *Closest Half* near their expected thresholds and compare them to *NUCAlloc*. Figure 9 shows the LLC miss trends of the different allocation strategies. We observe an exponential rise in the LLC miss count for *Closest One* at ~30,000 allocations. Similarly, *Closest Two* and *Closest Half* see a steep rise at ~60,000 and ~130,000 allocations, respectively. Notably, *NUCAlloc* sees higher miss rates than *Closest Half* at 100,000 allocations, because *NUCAlloc* fills each slice before switching to the next, seeing more frequent misses as the slice capacity approaches.

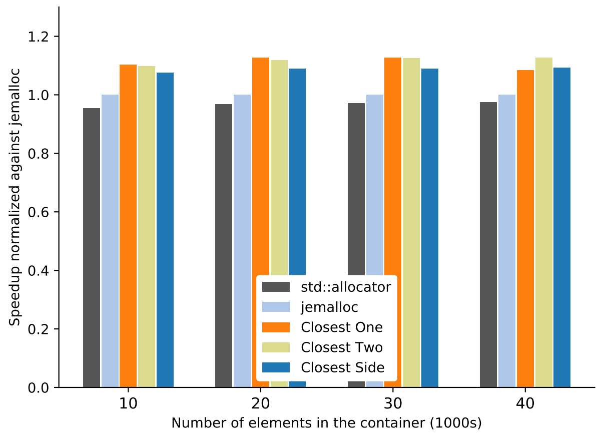

Figure 10: Performance of accessing the elements allocated through *std::allocator*, *Closest One*, *Closest Two* and *Closest Half* normalized to *jemalloc*, shown as a function of the container size. The total allocation in this graph is less than one slice's theoretical capacity.

Closest Half interleaves allocations between four slices, resulting in a slightly lower number of LLC misses. However, beyond 130,000 allocations, *NUCAlloc* still observes fewer misses due to its ability to use the entire cache capacity.

4.3 Containers Fitting into One LLC Slice

Figure 10 presents our results for different NUCA strategies with single-threaded applications whose working set fits within one cache slice. To highlight the differences between the allocation techniques, we normalize all results to *jemalloc*. We measure the performance in terms of clock cycles to access the elements of the container.

Theoretically, a cache slice can fit 40,960 elements in non-conflicting cache lines. However, allocation to the same slice increases the probability of conflicts, and their effect starts overshadowing the performance gains before reaching 40,000 elements. Figure 10 shows that the performance of *Closest One* starts to deteriorate when the number of elements exceeds 30,000 elements. Until that point, *Closest One* outperforms all other allocators.

The *Closest Two* behavior is similar to *Closest One*, even for allocations that fit within one cache slice. Because *Closest Two* interleaves allocations among neighboring slices, it has a much lower chance of LLC conflict misses compared to *Closest One*. At the same time, the access latency difference between the nearest and the neighboring slice is only ~3 cycles (Figure 6). *Closest One* performs better up to 30,000 elements, at which point performance starts to degrade sharply and, at 40,000 elements, the performance of *Closest One* matches *Closest Half*. Beyond that point, there is no benefit to using *Closest One*. Notably, all of the NUCA strategies have a speedup over both *std::allocator* and *jemalloc*.

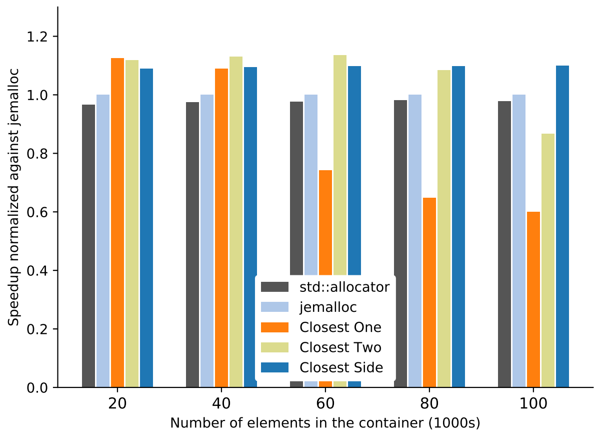

Figure 11: Behavior of *Closest One*, *Closest Two* and *Closest Half* allocation strategies when the container size exceeds a single slice.

Although *Closest One* does not appear to have much performance benefit over *Closest Two*, while at the same time benefiting from a larger number of possible allocations, *Closest One* is uniquely suited to isolate a slice from the rest of the LLC. This is particularly useful in multi-threaded applications that may want to restrict the LLC usage to a particular region to prevent pollution between threads pinned to neighboring cores. We further demonstrate these benefits in Section 4.8.

4.4 Containers Exceeding One LLC Slice

When the capacity of one slice is exceeded, *Closest Two* and *Closest Half* begin to outperform other allocation techniques. Figure 11 shows the performance characteristics of various allocators when the slice capacity is exceeded.

Closest Two performs the best once the slice capacity is exceeded, where *Closest One* starts experiencing conflict misses. This trend continues up to approximately 60,000 elements, from where the decline due to conflict misses in the LLC begins. By 80,000 elements, the theoretical maximum capacity has still not been reached, but *Closest Half* already performs better than *Closest Two*. Importantly, by 100,000 elements, *Closest Two* is performing much worse than NUCA-oblivious allocations by *std::allocator* and *jemalloc*.

In contrast, *Closest Half* experiences conflict misses well before the theoretical maximum number of elements are allocated, indicating that *NUCAlloc* should dynamically adapt from one strategy to another. A dynamic policy can benefit from the lower single-slice access latencies of *Closest One*, while matching *Closest Two* and *Closest Half* beyond one slice. Finally, we expect a dynamic policy to match the NUCA-oblivious allocators when LLC capacity is reached.

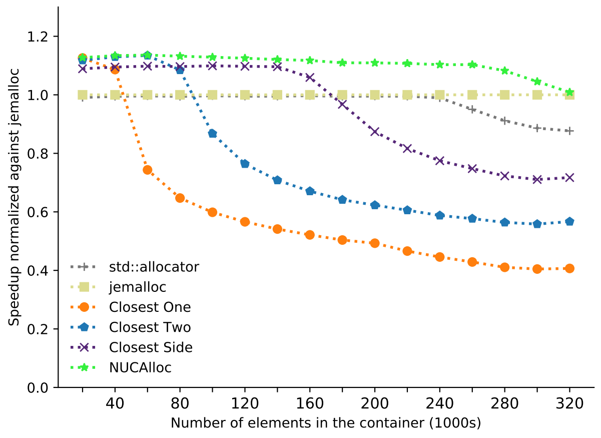

Figure 12: *NUCAlloc* performance on 8-core processor.

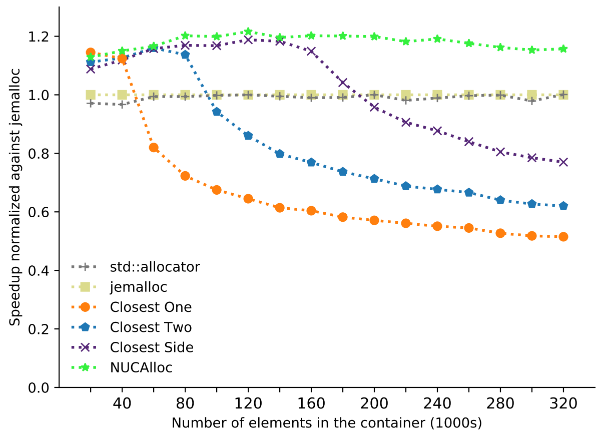

Figure 13: *NUCAlloc* performance on 12-core processor.

4.5 Further Portability Across CPUs

We evaluated *NUCAlloc* on a third processor, an Intel E5-2620v4. Like the 8-core processor presented earlier, the E5-2620v4 has a 20MB LLC divided into 8 slices, but the CPU micro-architecturally is similar to the 12-core processor, also having SMT support. We found that the hardware hash function is unchanged from the earlier results with the 8-core chip. Additionally, this processor allows software to determine the number of available LLC slices by reading the CAPID5 register located in the PCI configuration space [17]. In the interest of brevity, we omit the detailed results as they closely mirror Figure 6 (single ring) and Figure 7 (SMT).

4.6 Differences Across C++ Containers

We tested *NUCAlloc* with a variety of C++ containers, including *std::map*, *std::multimap*, *std::set*, *std::multiset*, and *std::list*. Due to the similarity of their internal red-black tree implementation, *std::set*, *std::multiset*, and *std::multimap* showed similar speedup in access latencies with up to 12-13% gain in single-threaded performance for the 8-core processor and 20% gain for the 12-core processor.

std::list is a linked list, similar in behavior to the list that we used in establishing the feasibility of our work in 4.1. The stronger dependence of performance on the access latency results in greater benefits for *NUCAlloc* when accessing the elements of a list. A single-threaded traversal of *std::list* observes speedups up to 21% for the 8-core and up to 30% for the 12-core processor for an LLC resident working set.

4.7 Single-Thread Performance

In this section, we compare the dynamic slice selection strategy employed by *NUCAlloc* to other NUCA allocation strategies: *Closest One*, *Closest Two*, and *Closest Half*, as well as to NUCA-oblivious allocation with *std::allocator* and *jemalloc*.

Figure 12 shows the speedup of *NUCAlloc* in a single-threaded scenario on the 8-core processor. In the beginning, *NUCAlloc* behaves similarly to *Closest One*, allocating memory blocks corresponding to the closest slice. Once the slice threshold is reached, *NUCAlloc* transitions to the next slice, matching the *Closest Two* performance. When the subsequent slice's threshold is reached, *NUCAlloc* transitions to the third and then fourth slice, exhibiting a performance increase comparable to *Closest Half*.

Beyond this point, the speedup gradually decreases, as the allocations start to correspond to the slices on the other side of the ring. By 240,000 elements, the slice threshold parameter of all the slices is reached, and *NUCAlloc* starts spreading allocations across the LLC by using round-robin selection among the slices' free-lists. Eventually, the performance characteristics of *NUCAlloc* match a NUCA-oblivious allocator.

Figure 13 shows the same experiment on the 12-core processor, having a similar trend. Due to the larger size of LLC in this chip, *NUCAlloc* continues to perform better beyond 320,000 elements. Moreover, the higher core count results in more potential to leverage the slice proximity, which also results in a greater speedup.

For single-threaded applications with LLC resident working sets, *NUCAlloc* always outperforms NUCA-oblivious techniques and reaches a peak speedup of 13% and 20% on the 8-core and 12-core processors, respectively.

4.8 Multi-Thread Performance

To evaluate the effectiveness of *NUCAlloc* in a multi-threaded application, we pin a thread on each core of the 8-core processor and measure the clocks cycles for simultaneously accessing LLC resident data. Each thread works on its own copy of the data and performs a pseudo-random sequences of accesses to its data. The results are shown in Figure 14.

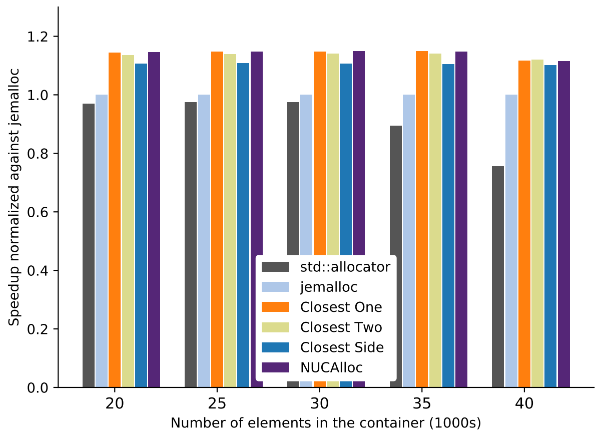

Figure 14: Performance comparison of various allocators for accesses from eight threads running on different cores of an 8-core processor.

NUCAlloc restricts each thread to its closest slice and avoids the thread accessing other cache slices, mimicking the behavior of *Closest One*. Restricting the threads to their local slice achieves isolation, preventing one thread from thrashing the others' data. We observe that *NUCAlloc* benefits from this, while the NUCA-oblivious allocators experience more frequent LLC misses. *NUCAlloc* achieves 15% speedup in this scenario, 2% higher than the single-threaded case. We also performed similar tests with four threads, such that two threads run on each side of the chip. *NUCAlloc* behaves like *Closest Two* before performance starts to degrade.

4.9 *NUCAlloc* on Traditional Benchmarks

A specialized allocator is best suited for applications that can leverage its specific strengths. Inherently, a specialized allocator is not applicable as a replacement of a general purpose allocator. *NUCAlloc* is ideally suited for single-threaded applications that perform many small allocations, with a working set that occupies a significant fraction, but not the entire, last-level cache. *NUCAlloc* also works well for multi-threaded applications that have a balanced workload and whose working sets are blocked and specifically configured to fit in the last-level cache. Conversely, *NUCAlloc* is not useful for applications whose working sets dramatically exceed the last-level cache capacity or whose working sets fit into the L1 caches.

In the previous sections, we demonstrated the behavior of *NUCAlloc* for micro-benchmarks. This, however, naturally leads to the criticism that *NUCAlloc* is applicable only to a limited set of cases and not to real-world applications. In fact, any application for which *NUCAlloc* is a good fit would necessarily be one that is designed for a very specific task and tuned exactly for *NUCAlloc*, leading to the same criticism. To address this concern, we identified several

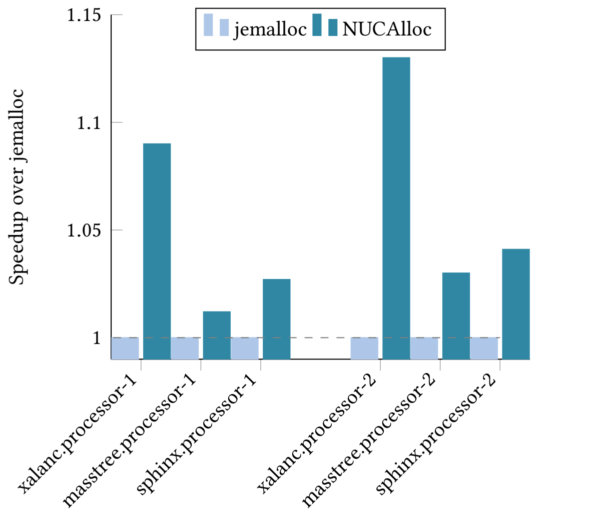

Figure 15: Speedup observed by *NUCAlloc*. Performance is measured in terms of the wall clock time for Xalanc, throughput for Masstree and 95th percentile tail latency for Sphinx.

well-established benchmark applications whose design is completely agnostic of the memory allocator used, that can nonetheless benefit from *NUCAlloc*. Crucially, rather than carefully analyzing these workloads and identifying specific places in their code where the use of *NUCAlloc* would be helpful, we used `LD_PRELOAD` to replace `malloc()` with a wrapper that invokes *NUCAlloc* for all small allocations without any source-level application changes. As a result, the potential performance benefits shown in Figure 15 are a lower bound of what can be achieved if the applications were re-engineered for *NUCAlloc*. Even with this less-than-ideal setup, we observe measurable performance benefits when using *NUCAlloc* for SPEC CPU2017 Xalanc, TailBench Masstree, and Sphinx (TailBench[21] and SPEC CPU2006). Importantly, these numbers are not hypothetical; this is actual "free" performance improvement on off-the-shelf Intel hardware across the architectures we study.

5 RELATED WORK

There is a rich history in the literature of various cache pollution and cache space management schemes that classify memory accesses and control how the cache should behave depending on the class of access being performed [7, 8, 19, 23, 25, 26, 30]. These techniques tend to work at coarse granularity, dividing the memory space into regions and either dictating or hinting how the hardware should handle cache blocks that fall within the given region. To minimize overheads and increase practicality, hardware techniques work on large memory regions. Software techniques also work on large regions, typically classifying and controlling cache behavior at OS-page granularity. These approaches manage the entire cache, treating the cache as monolithic, but tailoring its behavior with respect to placement and eviction.

Kaseridis et al. [20] developed a NUCA aware scheme that monitors the memory access pattern to dynamically partition NUCA banks among the cores. This technique effectively prevents destructive interference and significantly reduces the miss rate by using the proposed specialized hardware.

ROCS [27] presented a technique for software management of cache capacity, which used part of the cache as a *pollution buffer*. Cache-unfriendly pages would be steered into the pollution buffer in the LLC, avoiding interference with the rest of the cache blocks. Although separating polluting accesses was not the basis our work, the fine-grained allocation and placement schemes employed by *NUCAlloc* can be extended to implement a ROCS-like system that explicitly manages which region of the cache will receive each block.

Similar in spirit, SRM-Buffer [7] modified the buffer caches in operating systems to divide the LLC into sets. Each set received its own region to which cache pollution is limited. This technique also works on a page granularity and targets cache interference and pollution.

A seminal work by Cho et al. proposed a joint OS and microarchitecture approach of cache management by introducing the concept of *virtual multicores (VMs)* [5]. In this approach, cores and their adjacent caches are grouped together into virtual multicores, reducing the access latency from each group of cores to their caches. Running different applications in different virtual multicores allows them to receive a portion of the capacity of the on-chip cache, but they can access it without interference from the other cores and at lower latency. This cache management technique has similarities to our work, as it partitions the cache capacity and provides lower access latency to each partition from its corresponding cores. Unlike our work, this technique requires complex hardware support for virtual multicores and a modified operating system to perform space management at the OS-page granularity.

In a related study, Beckmann et al. [2] divided large caches into regions that appear to software as smaller caches. Cache capacity and latency are controlled by reconfiguring the cache partitions and their physical location on the processor. Unlike our work, a large part of this proposal focused on the movement of both computation and data, whereas we pin computation threads to cores.

Building on the observation of the wire-delay challenges of large on-chip caches, NUCA caches were proposed to replace monolithic caches with arrays of smaller cache slices [12, 22]. These works proposed static and dynamic block management that leveraged the non-uniform latencies within the cache for performance gains. Although the complex hardware mechanisms proposed in these works have not yet made it into commercial products, the statically mapped NUCA caches are built by processor vendors today, albeit with a hash function that spreads blocks randomly across the cache slices. Our work leverages the NUCA nature of the cache for performance benefits by undoing the hashing and exerting fine-grained control over the placement of cache blocks.

Farshin, Roozbeh, et al. [10] demonstrated slice-aware cache management with Intel DPDK network I/O, showing significant performance improvement on network applications by steering placement of packet headers into cache slices of the cores that process those packets. Similar to our work, such placement leverages the underlying NUCA organization of the cache. However, *NUCAlloc* takes a more generic approach to the opportunities offered by modern NUCA caches and provides a general-purpose memory allocator that can transparently improve the performance of small data structures by allocating data in the lowest-latency locations and then expanding to higher-latency locations as data structures grow.

Finally, R-NUCA [11] proposed placing blocks belonging to a thread onto the nearest cache slices to the core on which the thread runs, relying on the OS-page granularity classification to dictate how each block is cached. Much of the benefit of R-NUCA came from managing instruction block placement for server applications and replicating them in multiple slices. Our cache allocation policy is similar to this in spirit, but rather than separating instructions from data, we place individual data structures. Unlike R-NUCA, *NUCAlloc* works on off-the-shelf hardware and does not require hardware modification. Moreover, we perform cache placement transparently through the memory allocator, doing so at cache-block granularity rather than at the OS-page granularity.

6 CONCLUSIONS

In this work, we developed a systematic way to leverage the NUCA caches of multiple generations of Intel processors to improve software performance. We presented *NUCAlloc*, a specialized memory allocator that performs fine-grained block placement in the last level cache (LLC), which takes advantage of the NUCA architecture and the LLC slice proximity to the cores. We prototyped a dynamic slice selection technique used by *NUCAlloc*, comparing it with other strategies of NUCA allocation, showing that the approach is effective for a wide range of sizes of LLC-resident C++ containers. *NUCAlloc* improves performance by up to 20% for single-threaded and multi-threaded applications when accessing commonly used C++ STL containers on off-the-shelf processors, requiring no modification to the hardware and being transparent to the software application.

REFERENCES

[1] Bradford M. Beckmann, Michael R. Marty, and David A. Wood. 2006. ASR: Adaptive Selective Replication for CMP Caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 443–454.

[2] N. Beckmann, P. Tsai, and D. Sanchez. 2015. Scaling distributed cache hierarchies through computation and data co-scheduling. In *21st International Symposium on High Performance Computer Architecture (HPCA)*. 538–550.

[3] Hervé Brönnimann and Jyrki Katajainen. 2006. Efficiency of various forms of red-black trees. *CPH STL Report* 2 (2006), 2006.

[4] Zeshan Chishti, Michael D. Powell, and T. N. Vijaykumar. 2005. Optimizing Replication, Communication, and Capacity Allocation in CMPs. In *32nd Annual International Symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, USA, 357–368.

[5] Sangyeun Cho and Lei Jin. 2006. Managing Distributed, Shared L2 Caches through OS-Level Page Allocation. In *39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 39)*. IEEE Computer Society, USA, 455–468.

[6] Michael D. Dahlin, Randolph Y. Wang, Thomas E. Anderson, and David A. Patterson. 1994. Cooperative Caching: Using Remote Client Memory to Improve File System Performance. In *1st USENIX Conference on Operating Systems Design and Implementation* (Monterey, California) *(OSDI '94)*. USENIX Association, Berkeley, CA, USA, Article 19.

[7] Xiaoning Ding, Kaibo Wang, and Xiaodong Zhang. 2011. SRM-buffer: An OS Buffer Management Technique to Prevent Last Level Cache from Thrashing in Multicores. In *Sixth Conference on Computer Systems* (Salzburg, Austria) *(EuroSys '11)*. ACM, New York, NY, USA, 243–256.

[8] Haakon Dybdahl and Per Stenström. 2006. Enhancing Last-level Cache Performance by Block Bypassing and Early Miss Determination. In *11th Asia-Pacific Conference on Advances in Computer Systems Architecture* (Shanghai, China) *(ACSAC'06)*. Springer-Verlag, 52–66.

[9] Jason Evans. 2006. A Scalable Concurrent malloc(3) Implementation for FreeBSD. (2006), 14.

[10] Alireza Farshin, Amir Roozbeh, Gerald Q. Maguire, Jr., and Dejan Kostić. 2019. Make the Most out of Last Level Cache in Intel Processors. In *Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. ACM, New York, NY, USA, Article 8, 17 pages.

[11] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. 2009. Reactive NUCA: Near-optimal Block Placement and Replication in Distributed Caches. In *36th Annual International Symposium on Computer Architecture* (Austin, TX, USA) *(ISCA '09)*. ACM, New York, NY, USA, 184–195.

[12] Jaehyuk Huh, Changkyu Kim, Hazim Shafi, Lixin Zhang, Doug Burger, and Stephen W. Keckler. 2005. A NUCA Substrate for Flexible CMP Cache Sharing. In *19th Annual International Conference on Supercomputing* (Cambridge, Massachusetts) *(ICS '05)*. ACM, New York, NY, USA, 31–40.

[13] Ralf Hund, Carsten Willems, and Thorsten Holz. 2013. Practical Timing Side Channel Attacks against Kernel Space ASLR. In *2013 IEEE Symposium on Security and Privacy*. 191–205.

[14] Mehmet Sinan Inci, Berk Guuml;lmezoglu, Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. 2015. Seriously, get off my cloud! Cross-VM RSA Key Recovery in a Public Cloud. *IACR Cryptology ePrint Archive* 2015 (2015), 898.

[15] Intel Corporation. 2012. *Intel® Xeon® Processor E5-2600 Product Family Uncore Performance Monitoring Guide*.

[16] Intel Corporation. 2015. *Intel® Xeon® Processor E5 and E7 v3 Family Uncore Performance Monitoring Reference Manual*.

[17] Intel Corporation. 2016. *Intel® Xeon® Processor E5 and E7 v4 Product Families Uncore Performance Monitoring Reference Manual*.

[18] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design (DSD '15)*. IEEE Computer Society, Washington, DC, USA, 629–636.

[19] Teresa L. Johnson, Daniel A. Connors, Matthew C. Merten, and Wen-mei W. Hwu. 1999. Run-Time Cache Bypassing. *IEEE Trans. Comput.* 48, 12 (1999), 1338–1354.

[20] Dimitris Kaseridis, Jeffrey Stuecheli, and Lizy K. John. 2009. Bank-aware Dynamic Cache Partitioning for Multicore Architectures. In *2009 International Conference on Parallel Processing*. 18–25.

[21] Harshad Kasture and Daniel Sanchez. 2016. Tailbench: a benchmark suite and evaluation methodology for latency-critical applications. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 1–10.

[22] Changkyu Kim, Doug Burger, and Stephen W. Keckler. 2002. An Adaptive, Non-uniform Cache Structure for Wire-delay Dominated On-chip Caches. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems* (San Jose, California) *(ASPLOS X)*. ACM, New York, NY, USA, 211–222.

[23] Wei-Fen Lin and Steven K Reinhardt. 2002. Predicting Last-Touch References under Optimal Replacement. (2002), 17.

[24] Clémentine Maurice, Nicolas Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *18th International Symposium on Research in Attacks, Intrusions, and Defenses - Volume 9404* (Kyoto, Japan) *(RAID 2015)*. Springer-Verlag New York, Inc., New York, NY, USA, 48–65.

[25] Thomas Piquet, Olivier Rochecouste, and André Seznec. 2007. Exploiting Single-Usage for Effective Memory Management. In *12th Asia-Pacific Conference on Advances in Computer Systems Architecture* (Seoul, Korea) *(ACSAC '07)*. Springer-Verlag, Berlin, Heidelberg, 90–101.

[26] Moinuddin K. Qureshi, Aamer Jaleel, Yale N. Patt, Simon C. Steely, and Joel Emer. 2007. Adaptive Insertion Policies for High Performance Caching. In *34th Annual International Symposium on Computer Architecture* (San Diego, California, USA) *(ISCA '07)*. ACM, New York, NY, USA, 381–391.

[27] Livio Soares, David Tam, and Michael Stumm. 2008. Reducing the harmful effects of last-level cache polluters with an OS-level, software-only pollute buffer. In *41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, USA, 258–269.

[28] Po-An Tsai, Nathan Beckmann, and Daniel Sanchez. 2017. Nexus: A New Approach to Replication in Distributed Shared Caches. In *26th International Conference on Parallel Architectures and Compilation Techniques (PACT 26)*. 166–179.

[29] Thomas Tuttle. 2015. pagemap, from the userspace perspective.

[30] G. Tyson, M. Farrens, J. Matthews, and A.R. Pleszkun. 1995. A modified approach to data cache management. In *28th Annual International Symposium on Microarchitecture (MICRO 28)*. 93–103.

[31] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. *IACR Cryptology ePrint Archive* 2015 (2015), 905.

[32] M. Zhang and K. Asanovic. 2005. Victim replication: maximizing capacity while hiding wire delay in tiled chip multiprocessors. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 336–345.