NextGen-Malloc: Giving Memory Allocator Its Own Room in the House

Ruihao Li
liruihao@utexas.edu
The University of Texas at Austin

Qinzhe Wu
qw2699@utexas.edu
The University of Texas at Austin

Krishna Kavi
Krishna.Kavi@unt.edu
Univerisy of North Texas

Gayatri Mehta
Gayatri.Mehta@unt.edu
Univerisy of North Texas

Neeraja J. Yadwadkar
neeraja@austin.utexas.edu
The University of Texas at Austin
and VMware Research

Lizy K. John
ljohn@ece.utexas.edu
The University of Texas at Austin

ABSTRACT

Memory allocation and management have a significant impact on performance and energy of modern applications. We observe that performance can vary by as much as 72% in some applications based on which memory allocator is used. Many current allocators are multi-threaded to support concurrent allocation requests from different threads. However, such multi-threading comes at the cost of maintaining complex metadata that is tightly coupled and intertwined with user data. When memory management functions and other user programs run on the same core, the metadata used by management functions may pollute the processor caches and other resources.

In this paper, we make a case for offloading memory allocation (and other similar management functions) from main processing cores to other processing units to boost performance, reduce energy consumption, and customize services to specific applications or application domains. To offload these multi-threaded fine-granularity functions, we propose to decouple the metadata of these functions from the rest of application data to reduce the overhead of inter-thread metadata synchronization. We draw attention to the following key questions to realize this opportunity: (a) What are the tradeoffs and challenges in offloading memory allocation to a dedicated core? (b) Should we use general-purpose cores or special-purpose cores for executing critical system management functions? (c) Can this methodology apply to heterogeneous systems (e.g., with GPUs, accelerators) and other service functions as well?

CCS CONCEPTS

- **Software and its engineering → Memory management**.

KEYWORDS

Memory Management

ACM Reference Format:
Ruihao Li, Qinzhe Wu, Krishna Kavi, Gayatri Mehta, Neeraja J. Yadwadkar, and Lizy K. John. 2023. NextGen-Malloc: Giving Memory Allocator Its Own Room in the House. In *Workshop on Hot Topics in Operating Systems (HOTOS '23), June 22–24, 2023, Providence, RI, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3593856.3595911

1 INTRODUCTION

Data intensive applications, including machine learning (ML) training and inference serving rely on efficient management of large amounts of data. With these increasing capacities and deep memory hierarchies, the memory wall [35] (viz., the gap between processor speeds and memory access latencies) is becoming an ever increasing impediment. ML accelerators spend only 23% of the overall cycles on computations and more than 50% on data preparation [15]. Even in general purpose warehouse-scale datacenters, 50% of compute cycles are idle waiting for memory accesses [14, 16], causing performance degradation for the applications. To alleviate such performance bottlenecks, efficient memory management mechanisms [14, 18, 20] that reduce memory needs (by reducing fragmentation) and cache misses (by reducing conflicts between metadata and application data) are crucial.

To address the need for efficient memory allocation and management, multiple software solutions, in the form of optimized Malloc libraries, have been created (e.g., TCMalloc [14] from Google and Mimalloc [18] from Microsoft). Such libraries are typically multi-threaded to support concurrent requests for memory allocation and management, and rely on complex metadata to accomplish the allocation

HOTOS '23, June 22–24, 2023, Providence, RI, USA

ACM ISBN 979-8-4007-0195-5/23/06.
https://doi.org/10.1145/3593856.3595911

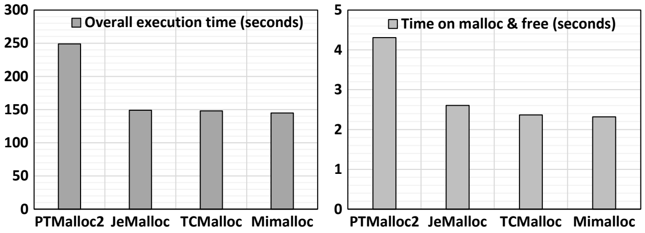


Figure 1: Execution time sensitivity to memory allocation: variations up to 72% in *xalancbmk*, though only 2% time spent on malloc and free.

and management. However, if the same core executes memory management and application code, maintaining such metadata can pollute processor caches, cause conflicts on other resources and can affect overall performance.

To understand the effectiveness of existing software solutions for memory management, we compare the performance of four different memory allocators (PTMalloc2 [12], Jemalloc [9], TCMalloc [14], and Mimalloc [18]) for *xalancbmk* (a representative workload from SPEC cpu2017 [23, 25], which performs transformations on XML data). We observe that with an enhanced memory allocator, the overall system performance can be improved by as much as 1.72× (see Figure 1). On other allocation intensive workloads, e.g., *xmalloc* and *cache-scratch* in mimalloc-bench [18, 22], performance variance can be more than 10× depending on the allocator used. We notice that the choice of memory allocator not only impacts the performance of the allocator code itself but also other parts of the program. For example, although only 2% of the execution time is spent on malloc and free by *xalancbmk*, we can see as much as 72% difference in overall execution time when Mimalloc is used instead of PTMalloc2. This makes a compelling case for further investigating and optimizing these memory management functions.

Existing memory allocation mechanisms that are implemented in software achieve higher performance compared with the default Glibc (PTMalloc2) memory allocators. However, they still fall short in leveraging system-level optimizations, like reducing cache pollution and TLB misses, which cannot be implemented without understanding underlying hardware. In addition, software-only solutions always suffer from balancing allocation speed and memory consumption (partly due to fragmentation), making it difficult to develop a one-size-fits-all allocation solution.

Alternatively, hardware accelerators are considered for implementing memory allocation mechanisms. For instance, Kanev et al. [17], proposed a separate cache inside main CPUs for memory management algorithms, thus eliminating cache conflicts between memory management metadata and application data, while still utilizing powerful main CPUs for memory management. Mass et al. [19], used a near-memory accelerator to offload garbage collection functions. However, most hardware solutions rely on customized hardware units, that (a) make it infeasible to use them as general purpose memory management solution, or (b) necessitate frequent changes to the hardware as algorithms evolve.

Next generation memory allocators will likely continue to be complex. One way to handle their complexity without impacting application performance is to isolate the allocation functions from rest of the code and provide separate resources to it. This will prevent allocators from polluting the cache and interfering with other metadata of applications. However, current allocators cannot easily be "plucked" out of the code and offloaded to dedicated cores, because the metadata is usually tightly coupled to the user data.

In this paper, we introduce *NextGen-Malloc*, a novel memory allocator that restructures the memory management metadata and makes it possible to offload the allocation to a separate dedicated core. The design of *NextGen-Malloc* necessitates innovations in software, hardware and their codesign. Just like a baby in a big family, the memory allocator is growing up. Now it is time to give it a new room (core) in our house (CPU). New research questions will arise with the growing child *NextGen-Malloc*. We will explore these questions in this paper. How to tradeoff the overhead (additional inter-core communication) and the benefit (a reduction of cache pollution and asynchronous execution) of offloading memory allocators? The choice of the room type (viz., type of processing core) is another research question. Should the room be the same as other rooms (i.e., other CPU cores), or a small room is enough for memory allocation? Can the room be used for other functions instead of exclusively for memory allocation?

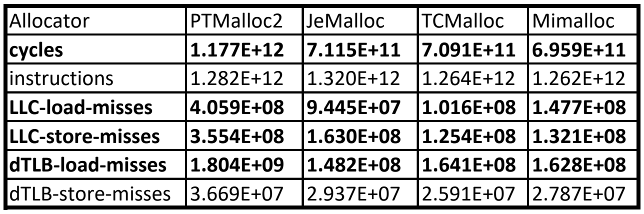
2 BACKGROUND AND MOTIVATION

This section identifies the primary reasons and inefficiencies contributing to the performance differences among different memory allocators.

2.1 Memory Allocators

Memory management operations exist at two levels: user level and kernel level. User-level memory management operations are implemented by a *UMA* (user-level memory allocator), such as PTmalloc2 [12] (the default Glibc implementation), TCMalloc [14] (by Google), and Mimalloc [18] (by Microsoft). There are other domain-specific UMAs as well, such as NVAlloc [7] for non-volatile memory. For the kernel-level operations, $mmap()$ system calls are used to manage private anonymous mapping segments for each process. But relying on $mmap()$ system call for each $malloc()$ can result

Table 1: Processor performance monitor data for *xalancbmk*. TLB misses vary more than 10x and LLC load misses vary 4x between TCMalloc and PTMalloc2.



Allocator	PTMalloc2	JeMalloc	TCMalloc	Mimalloc
cycles	**1.177E+12**	**7.115E+11**	**7.091E+11**	**6.959E+11**
instructions	1.282E+12	1.320E+12	1.264E+12	1.262E+12
LLC-load-misses	**4.059E+08**	**9.445E+07**	**1.016E+08**	**1.477E+08**
LLC-store-misses	**3.554E+08**	**1.630E+08**	**1.254E+08**	**1.321E+08**
dTLB-load-misses	**1.804E+09**	**1.482E+08**	**1.641E+08**	**1.628E+08**
dTLB-store-misses	3.669E+07	2.937E+07	2.591E+07	2.787E+07

in significant performance penalties caused by switching between user and kernel modes. To minimize these overheads, most UMAs request an entire page[1], which is usually larger than the size requested by $malloc()$ calls. For subsequent $malloc()$ calls, the UMA can satisfy the requests out of the page and make additional $mmap()$ calls only when space in the allocated page is exhausted.

Observation: *Managing the memory space within preallocated pages in a way that strikes a balance between allocation speed and memory fragmentation is an open research problem.*

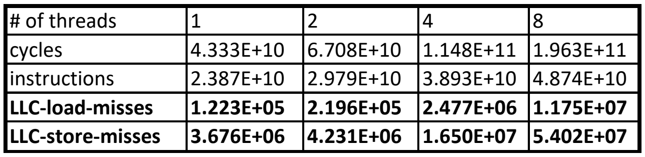
2.2 Malloc impacts more than you think!

The implementation of memory management functions can potentially have significant impact on the overall execution time of applications. In this section, we present evidence to support the claim we made earlier: *"Though only 2% of time is spent on memory allocation, more than 70% performance difference can exist with different implementations"*.

The above claim may appear counter-intuitive. Table 1 lists the hardware PMU (Performance Monitor Unit) event counts for *xalancbmk* with the four memory allocators. Based on the profiling results, the number of LLC (Last Level Cache) load and store misses, and the number of dTLB (data Translation Lookaside Buffer) load misses are reduced dramatically (besides absolute miss numbers, the miss rate reduced as well) on the three state-of-the-art industry-level allocators (Jemalloc, Mimalloc, and TCMalloc) compared with the default Glibc allocator (PTMalloc2). The impact of TLB misses is non-negligible, which can incur 100s of cycles in modern processors [11, 27]. In addition, in warehouse-scale computers, half of the CPU back-end cycles can be spent on serving data cache requests [16].

Observation: *TLB and LLC misses are key factors affecting the overall performance and sensitive to different memory allocators. Alleviating the cache pollution is a potential direction for the next-generation memory allocators to explore, to achieve a lower number of TLB and LLC misses.*

Table 2: PMU data for *xmalloc* on TCMalloc using different number of threads. LLC misses increase more than 10x when the number of threads increases from 1 to 8.



# of threads	1	2	4	8
cycles	4.333E+10	6.708E+10	1.148E+11	1.963E+11
instructions	2.387E+10	2.979E+10	3.893E+10	4.874E+10
LLC-load-misses	**1.223E+05**	**2.196E+05**	**2.477E+06**	**1.175E+07**
LLC-store-misses	**3.676E+06**	**4.231E+06**	**1.650E+07**	**5.402E+07**

2.3 Existing Allocators are Limiting

Modern UMAs support concurrent memory $malloc()$ and $free()$ operations for multi-threaded programs, which requires processing $malloc()$ and $free()$ requests from different physical cores and different address spaces simultaneously. As the number of cores within a single socket keeps increasing, it will compound the challenges for software UMAs in handling **thread contentions**: the inter-core metadata synchronization requires atomic operations (or locks) to maintain the upper-level global free lists. Software mutex locks are used to control access to metadata to process requests from different cores. The cost of using such software locks is high since cross-core communication is involved, causing a critical performance bottleneck as the communication overheads increase with the number of cores [29, 34].

Multiple software solutions have been used to address the multi-thread contention issue. TCMalloc [13] uses per-CPU/thread cache to maintain metadata associated with each logical core, avoiding locks for most memory allocations and deallocations. Mimalloc [18] uses three page-local shared free lists to increase locality, avoid contention, and support a highly-tuned allocation and free on fast path. However, maintaining thread-local caches will increase metadata size, resulting in more heap memory consumption and more cache pollution for the user program. Also, the number of logical threads may exceed the number of available physical cores, resulting in more overhead in switching between thread-local metadata.

To demonstrate the potential cache pollution in multi-threaded UMAs, we collect the PMU event counts for *xmalloc* [4] workload[2] on TCMalloc (listed in Table 2). As the number of threads increases, they contend with each other for accessing the thread-local cache. This results in a significant LLC miss increment (more than 10x when the number of threads increases from 1 to 8).

Observation: *It is challenging to address thread contention well without understanding the underlying hardware mechanisms for synchronization and communication.*

[1] The page size for UMA may be different from OS page size [14].

[2] *xmalloc* [4] is a multi-threaded benchmark by Lever and Boreham and used to exercise cases where a thread allocates data but a different thread deallocates the allocated blocks.

Expectation: *If the execution of the memory allocator and user program are separated, all thread-local metadata can reside within one core, alleviating cache pollution. This can be achieved by making malloc() run on a dedicated core separating from other user programs.*

3 OPPORTUNITIES

Merely offloading memory allocators to a dedicated core is not what *NextGen-Malloc* entails. There are several challenges to be solved to make it a reality and to achieve its full potential. We present some impactful avenues here.

3.1 Offload Malloc to its Own Core

By offloading memory management functions to a dedicated core, the overall performance can potentially benefit from parallel/concurrent execution and a reduction of cache pollution. However, offloading these fine granularity functions is still a challenge for current computer systems, considering the overhead of inter-core communications and the tight coupling between data and metadata. We detail the challenges and propose remedies that can make *NextGen-Malloc* a beneficial reality.

3.1.1 Challenges. While it is not uncommon to see CPU resource underutilization in datacenters (for example, 60% of the Virtual Machines on Microsoft Azure have an average CPU utilization lower than 20% [6, 33]), and while offloading system functions to free cores to boost overall performance has been tried for other problems (e.g., Shenango [24] dedicates a single busy-spinning core per machine to a centralized software entity), there are unique challenges to offloading memory allocators. The majority of memory allocation function calls are fine-grained and can be finished within 100 cycles [17]. Thus memory allocation amounts to 0.05 μs on a 2GHz machine, while in Shenango [24] the granularity is 5 μs (the busy-spinning core reschedules jobs among cores every 5 μs, and the duration of each job is usually longer than 5 μs). In comparison to allocation time-scales, the overhead of inter-core communication is non-negligible. A single Atomic Read-Modify-Write (RMW) instruction needed for inter-core synchronization can take 67 cycles on average on a Sandy Bridge machine [26], and almost 700 cycles in the worst case [3]. MMT [31] explored offloading memory allocation tasks to a memory management thread in a 2-core system. The performance did not improve without aggressive preallocations, which makes it only work for workloads with known allocation patterns.

Expectation: *Whether the overall performance can benefit from offloading memory allocators to a dedicated core is still an open question. It depends on whether the inter-core synchronization overhead can be reduced to acceptable levels.*

3.1.2 Strategy 1 – Decoupling. Decoupling the metadata memory space from user data is one of the solutions that may lead to the overall performance gains from offloading the memory allocator. Although MMT [31] offloads the memory allocator to a separate thread, the cache pollution issue cannot be alleviated unless all metadata is accessed exclusively by the dedicated core for memory management. Before discussing allocator decoupling, we will discuss Metadata Layout first. Figure 2 shows two representative metadata layouts, Aggregated Layout (used by Mimalloc) and Segregated Layout (used by TCMalloc).

In *Aggregated Layout*, the first 8 bytes (assuming 64-bit word size) of each free block are used as the pointer to the next free block. Global *Head* and *Tail* pointers are used for each page (each page has a different number of blocks, depending on the block size (the block size is not necessarily a power of 2 [13, 18]). If a block is accessed directly after the $malloc()$ call, since 56 bytes (assuming 64-byte cache line size) of the allocated data will already be cached by the $malloc()$ function, better spatial localities will result for the user program. In addition, the memory block can easily be located in the $free()$ phase, since each block can be indexed by the address directly.

However, optimizing locality for aggregated layout calls for (1) the allocator and user program must be executed on the same physical core, (2) the allocated memory must be accessed immediately after it is allocated, (3) the cached memory block must not be evicted before it is deallocated.

In *Segregated Layout*, metadata is decoupled from the allocated storage, which may address the cache pollution issue. Instead of an 8-byte pointer, a smaller index (16-bit for example) can be used to indicate the location of the free page, reducing memory fragmentation as well. Segregated layout requires more complicated control logic for locating the pointer to the freed data during the $free()$ phase. Cache pollution can still exist since CPUs have to access data needed for malloc functions and other computational functions as well.

Expectation: *Trade-offs always exist between the aggregated and segregated layouts and there is no global optimal layout. However, segregated layout is more suitable for offloading memory allocators in NextGen-Malloc. With segregated layout, the address space of metadata and user data can be separated. The additional cost in* $free()$ *would not be an issue, since the entire* $free()$ *phase is not on the critical path and can be executed asynchronously in the dedicated core.*

3.1.3 Strategy 2: Removing unnecessary atomic operations in UMAs. In current UMA designs, if one thread tries to free a memory block that was allocated by another running thread, contention will result (as described in Section 2.3) and atomic operations are required to sequentialize changes

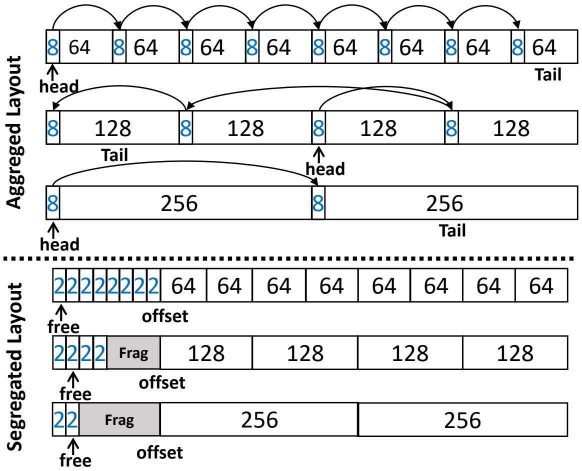


Figure 2: Metadata (pointers to blocks) shown in blue is decoupled from the allocated data in segregated layout whereas interspersed with data in aggregated layout. Data blocks are 64, 128, or 256 bytes.

made to critical resources to avoid race conditions. When offloading the memory allocator to a dedicated core, all memory management operations will be handled by one specific core, metadata contention should be eliminated theoretically (from a hardware perspective). Therefore, we can remove redundant atomic operations in UMAs, since sequential execution can be guaranteed if all allocation codes are running in one specific core.

Questions: *Whether the contention is reduced or not is still an open question, depending on the trade-off between the introduced synchronization overhead in NextGen-Malloc and removed atomic operations in current UMAs.*

3.2 Type of Core to Offload to

One possibility is to offload allocation to an idle core in the system. An alternative solution is to use a near-memory core. Harvesting an idle core may be more cost-effective, while a near-memory core could be energy-efficient and/or offer opportunities for customization. A general-purpose core makes it compatible with other system instruction sets and ease of compiling UMAs. A single-threaded in-order integer CPU may be adequate for a near-memory design, since memory address computations do not need floating point operations or complex control logic. The near-memory core will likely have lower memory access latencies; thus requiring only a small (micro) cache for buffering metadata. It is crucial to explore these design choices as they have the potential to improve performance while reducing energy/power consumption of memory management functions.

3.3 What else can *NextGen-Malloc* offer?

In addition to UMAs, *NextGen-Malloc* can also be applied to other memory management scenarios. There are several opportunities that can be explored extending *NextGen-Malloc.*

3.3.1 *Optimizing GPU Malloc.*

In addition to CPU-only memory allocation systems, memory management functions on CPU-GPU heterogeneous systems can be included in *NextGen-Malloc* as well. When GPU memory spaces are included, more trade-offs will be there. For example, in Nvidia GPUs, UVM (Unified Virtual Memory) [1, 36] is used to make the address space shared across the CPU hosts and GPU devices. Redundant memory transmission, data allocation granularity, and address space mapping are open questions. Asynchronous allocation can be used, which can also be part of the asynchronous CUDA memory copy. Both CPU and GPU memory allocators can be decoupled from user programs for faster address translation. All these features can be explored in *NextGen-Malloc.*

3.3.2 *Other Functions to Offload.*

In addition to user-level memory allocations, *NextGen-Malloc* can be used for other memory management functions. This flexibility comes from the programmable feature of *NextGen-Malloc.* In the kernel space, *NextGen-Malloc* can be combined with other resource management mechanisms, e.g., Caladan [10], to balance CPU and memory usage of different tasks. More intelligence can be programmed to observe allocation requests and utilize such information to predictively preallocate memory to reduce allocation latencies.

Also, there are opportunities in memory management for other scenarios, including managed languages (e.g., Java and .NET) and serverless functions (e.g., AWS Lambda [2] and Azure Functions [21]). With GC (Garbage Collector) invoked, LLC miss rate is reduced in .NET [8]. With different GC settings, the performance of the program can be affected a lot. Research opportunities for using *NextGen-Malloc* to process garbage collection will be worth exploring. Booting a function in FaaS (Functions-as-a-Service) systems through cold start can introduce extensive overhead, including additional memory consumption and allocation time [5, 28, 30, 32]. To avoid duplicate runtime library initialization in different containers, *NextGen-Malloc* can be extended to monitor inter-process memory heap similarities in FaaS systems as well.

4 NEXTGEN-MALLOC IS FEASIBLE

As outlined in previous sections, *NextGen-Malloc* can support asynchronous execution and reduce LLC and TLB misses by separating the memory space of memory management function metadata between user data, but at the cost of additional inter-core communication. In this section, we model this trade-off analytically and validate it by prototyping *NextGen-Malloc* for execution on real machines and comparing it with state-of-the-art memory allocators.

4.1 *NextGen-Malloc* Performance Estimates

We use *xalancbmk* as an example in an analytical model to demonstrate the potential performance improvement if *NextGen-Malloc* is used. In *xalancbmk*, *malloc*() and *free*() functions are called 279, 759, 405 times in total (138, 401, 260 *malloc*() and 141, 394, 145 *free*()). The performance overhead for these functions comes from atomic operations needed at the beginning and end of each malloc and free function calls. Assuming a 67-cycle latency for one atomic operation [3], there will be around 75 billion additional cycles introduced by *NextGen-Malloc*. We assume that any performance benefit of *NextGen-Malloc* is from reduced LLC and TLB misses. Comparing Mimalloc to Glibc, we can calculate that the average LLC and TLB miss penalty is 214 cycles. To amortize the overhead, *NextGen-Malloc* has to achieve a reduction of at least 1.25 Cache/TLB misses in each *malloc*()/*free*() and the subsequent user program before the next *malloc*()/*free*() comes. This is possible to achieve in *NextGen-Malloc*, since there are 7 load and store instructions in each *malloc*() and 10 in each *free*() (for Mimalloc).

4.2 Validating *NextGen-Malloc*

We prototype *NextGen-Malloc* on an AWS-A1 bare metal machine, with 16 Armv8-A Cortex-A72 (ARM uses a weaker memory model, which reduces the inter-core synchronization overhead.) cores and 32 GiB memory, by overwriting functions defined in standard C libraries, e.g., *malloc*() and *free*(), and compile it as a shared library. In *NextGen-Malloc*, we spawn a child thread from the main thread when the process is forked, pin it to a specific core, and let the spawned-thread check signals from the main thread for incoming *malloc*() and *free*() requests. We use the *malloc*() function as an example (shown in Code 1) to demonstrate the synchronization system in *NextGen-Malloc*, which is developed for the communication between the spawned child thread and main threads.

Two atomic variables *malloc_ready* and *malloc_done* are used at the beginning and end of *nextgen_malloc*() and *malloc*(), which are executed by the child thread and main thread, respectively. The *requested_size* and *allocated_block* are the input and output of *malloc*() functions, and this information is transferred between two threads. In this way, all other *malloc*() related metadata will be stored exclusively on the dedicated core, alleviating cache pollution that exists in state-of-the-art memory allocators.

```
void nextgen_malloc() { // executed in the child thread
    while (true)
        if (malloc_ready) {
            size_t size = atomic_load(requested_size);
            void *block =  normal_malloc(size);
            atomic_store(allocated_block, block);
            atomic_store(malloc_done, 1);
            atomic_store(malloc_ready, 0);
        }
}
void* malloc(size_t size) { // executed in main threads
    while(atomic_load(malloc_ready));
    atomic_store(requested_size, size);
    atomic_store(malloc_ready, 1);
    while(!atomic_load(malloc_done));
    return atomic_load(allocated_block);
}
```

Pseudo Code 1: *NextGen-Malloc* Synchronization.

We still use *xalancbmk* as the workload. We make a side-by-side comparison between Mimalloc and *NextGen-Malloc* in Table 3. *NextGen-Malloc* achieves a **4.51%** performance improvement, which is also coming from a reduction of dTLB load, LLC load, and LLC store misses.

Table 3: Execution time, dTLB MPKI (misses per kilo instructions), LLC load MPKI improve by 4.5%, 43%, and 22% respectively on moving from Mimalloc to the proposed *NextGen-Malloc*. AWS-A1 (ARMv8) Platform was used for this study.

	Mimalloc	*NextGen-Malloc*	**Improvement**
Execution Time	532*s*	508*s*	**4.51%**
dTLB-load-MPKI	6.092	3.452	43.33%
LLC-load-MPKI	8.911	6.889	22.69%
LLC-store-MPKI	0.783	0.496	36.59%

5 CONCLUSION

In this paper, we draw attention to the opportunity to achieve significant performance gains by offloading fine-granularity service functions like memory allocation to a separate core/device. These memory allocation functions cause cache pollution when they are run on the same core as the application code due to the tight coupling of their metadata with user data. We discuss challenges in offloading such management functions to a separate core and propose avenues for future research.

Acknowledgement: We thank our shepherd, Aurojit Panda, and the anonymous reviewers for their constructive feedback and insightful comments. This research was supported in part by NSF grant number 1763848 and 1828105, and gifts from Cisco and Amazon Web Services (AWS). The authors would also like to acknowledge the computing servers donated by Ampere Computing and TACC. Any opinions, findings, conclusions, or recommendations are those of the authors and not of the National Science Foundation or other sponsors.

REFERENCES

[1] Tyler Allen and Rong Ge. 2021. In-depth analyses of unified virtual memory system for GPU accelerated computing. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[2] Amazon. 2023. AWS Lambda. https://aws.amazon.com/lambda/.

[3] Ashkan Asgharzadeh, Juan M Cebrian, Arthur Perais, Stefanos Kaxiras, and Alberto Ros. 2022. Free atomics: hardware atomic operations without fences.. In *ISCA*. 14–26.

[4] David Boreham. 2000. Malloc () performance in a multithreaded Linux environment. In *2000 USENIX Annual Technical Conference (USENIX ATC 00)*.

[5] Joao Carreira, Sumer Kohli, Rodrigo Bruno, and Pedro Fonseca. 2021. From warm to hot starts: Leveraging runtimes for the serverless era. In *Proceedings of the Workshop on Hot Topics in Operating Systems*. 58–64.

[6] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. 2017. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 153–167.

[7] Zheng Dang, Shuibing He, Peiyi Hong, Zhenxin Li, Xuechen Zhang, Xian-He Sun, and Gang Chen. 2022. NVAlloc: rethinking heap metadata management in persistent memory allocators. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 115–127.

[8] Aniket Deshmukh, Ruihao Li, Rathijit Sen, Robert R Henry, Monica Beckwith, and Gagan Gupta. 2021. Performance characterization of. net benchmarks. In *2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 107–117.

[9] Jason Evans. 2006. A scalable concurrent malloc (3) implementation for FreeBSD. In *Proc. of the bsdcan conference, ottawa, canada*.

[10] Joshua Fried, Zhenyuan Ruan, Amy Ousterhout, and Adam Belay. 2020. Caladan: Mitigating interference at microsecond timescales. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 281–297.

[11] Jayneel Gandhi, Mark D Hill, and Michael M Swift. 2016. Agile paging: Exceeding the best of nested and shadow paging. *ACM SIGARCH Computer Architecture News* 44, 3 (2016), 707–718.

[12] Wolfram Gloger. 2022. "Wolfram Gloger's malloc homepage". http://www.malloc.de/en/.

[13] Google. 2023. TCMalloc. https://github.com/google/tcmalloc/.

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

[15] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. 2017. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*. 1–12.

[16] Svilen Kanev, Juan Pablo Darago, Kim Hazelwood, Parthasarathy Ranganathan, Tipp Moseley, Gu-Yeon Wei, and David Brooks. 2015. Profiling a warehouse-scale computer. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 158–169. https://doi.org/10.1145/2749469.2750392

[17] Svilen Kanev, Sam Likun Xi, Gu-Yeon Wei, and David Brooks. 2017. Mallacc: Accelerating Memory Allocation. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 33–45. https://doi.org/10.1145/3037697.3037736

[18] Daan Leijen, Ben Zorn, and Leonardo de Moura. 2019. *Mimalloc: Free List Sharding in Action*. Technical Report MSR-TR-2019-18. Microsoft. https://www.microsoft.com/en-us/research/publication/mimalloc-free-list-sharding-in-action/

[19] Martin Maas, Krste Asanović, and John Kubiatowicz. 2018. A hardware accelerator for tracing garbage collection. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 138–151.

[20] Martin Maas, Chris Kennelly, Khanh Nguyen, Darryl Gove, Kathryn S. McKinley, and Paul Turner. 2021. Adaptive Huge-Page Subrelease for Non-Moving Memory Allocators in Warehouse-Scale Computers. In *Proceedings of the 2021 ACM SIGPLAN International Symposium on Memory Management* (Virtual, Canada) *(ISMM 2021)*. Association for Computing Machinery, New York, NY, USA, 28–38. https://doi.org/10.1145/3459898.3463905

[21] Microsoft. 2023. Azure Functions. https://azure.microsoft.com/en-us/products/functions/.

[22] Microsoft. 2023. Mimalloc-bench. https://github.com/daanx/mimalloc-bench/.

[23] SPEC org. 2022. SPEC CPU 2017. https://www.spec.org/cpu2017/.

[24] Amy Ousterhout, Joshua Fried, Jonathan Behrens, Adam Belay, and Hari Balakrishnan. 2019. Shenango: Achieving high {CPU} efficiency for latency-sensitive datacenter workloads. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*. 361–378.

[25] Reena Panda, Shuang Song, Joseph Dean, and Lizy K John. 2018. Wait of a decade: Did SPEC CPU 2017 broaden the performance horizon?. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 271–282.

[26] Bharghava Rajaram, Vijay Nagarajan, Susmit Sarkar, and Marco Elver. 2013. Fast RMWs for TSO: Semantics and implementation. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 61–72.

[27] Jee Ho Ryoo, Nagendra Gulur, Shuang Song, and Lizy K John. 2017. Rethinking TLB designs in virtualized environments: A very large part-of-memory TLB. *ACM SIGARCH Computer Architecture News* 45, 2 (2017), 469–480.

[28] Divyanshu Saxena, Tao Ji, Arjun Singhvi, Junaid Khalid, and Aditya Akella. 2022. Memory deduplication for serverless computing with medes. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 714–729.

[29] Hermann Schweizer, Maciej Besta, and Torsten Hoefler. 2015. Evaluating the cost of atomic operations on modern architectures. In *2015 International Conference on Parallel Architecture and Compilation (PACT)*. IEEE, 445–456.

[30] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. 2020. Serverless in the Wild: Characterizing and Optimizing the Serverless Workload at a Large Cloud Provider. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 205–218. https://www.usenix.org/conference/atc20/presentation/shahrad

[31] Devesh Tiwari, Sanghoon Lee, James Tuck, and Yan Solihin. 2010. Mmt: Exploiting fine-grained parallelism in dynamic memory management. In *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE, 1–12.

[32] Dmitrii Ustiugov, Plamen Petrov, Marios Kogias, Edouard Bugnion, and Boris Grot. 2021. Benchmarking, analysis, and optimization of serverless function snapshots. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 559–572.

[33] Yawen Wang, Kapil Arya, Marios Kogias, Manohar Vanga, Aditya Bhandari, Neeraja J Yadwadkar, Siddhartha Sen, Sameh Elnikety, Christos Kozyrakis, and Ricardo Bianchini. 2021. SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud. In *Proceedings of the Sixteenth European Conference on Computer Systems*. 1–16.

[34] Qinzhe Wu, Jonathan Beard, Ashen Ekanayake, Andreas Gerstlauer, and Lizy K John. 2021. Virtual-Link: A Scalable Multi-Producer Multi-Consumer Message Queue Architecture for Cross-Core Communication. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 182–191.

[35] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[36] Weixi Zhu, Guilherme Cox, Jan Vesely, Mark Hairgrove, Alan L Cox, and Scott Rixner. 2022. UVM Discard: Eliminating Redundant Memory Transfers for Accelerators. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 27–38.