Impact of memory allocation on the performance of a delegation synchronization algorithm

Written by Riyane SID-LAKHDAR (M2 MoSIG)
Supervised by Thomas ROPARS (LIG, team ERODS)

Abstract

Memory allocation is a complex operation that can have a deep impact on the performance of concurrent algorithms. Delegation represents an example of such algorithms. However, few studies have been conducted to assess such an impact.

This paper exposes our evaluation of two delegation algorithms with state of the art allocators on two multicore platforms.

We show that the used allocator may limit the performance of the considered multithread applications. We also demonstrate that the delegation algorithm performing best depends on the allocator performance and the considered hardware architecture.

1 Introduction

With the rise of many-core processors, concurrent programs have increased the efficiency of existing algorithms by simply splitting their tasks over concurrent cores. However, due to hardware and software issues, parallelizing the execution of a task using *N* cores does not mean dividing its execution time by *N*.

Among the things that can limit the efficiency of concurrent algorithms, memory allocation is one that is often disregarded but that can still deeply affect performance. The main problem when dealing with a large number of threads, that frequently allocate and free memory, is that the memory allocation system may receive requests from different threads at the same time. If the memory allocation system answers the requests sequentially, it becomes the performance bottleneck of multithreading [11].

Another issue for multithreaded applications is the need to synchronizing concurrent memory accesses [11]. In this context, *delegation* [9] is an example of contributions to implement efficiently mutual exclusion on shared objects. However, existing evaluations of *delegation* algorithms have been obtained using a custom allocator [1]. Hence our main objective is to study how delegation algorithms would perform when used with state-of-the-art general-purpose allocators.

Different strategies have been proposed to improve the efficiency of dynamic memory management. Thus, in the present work, we first introduce, in section 3, an overview of the main state-of-the-art allocators for multithreaded systems.

Section 4 and 5 present our experimental evaluation of two custom *delegation* algorithms over two specific processor architectures: a 20 cores Intel Sandy Bridge, and a 24 cores AMD Bulldozer.

Using *tcmalloc*, the allocator implementation that best fits our workbench, we are able to infirm some properties previously obtained on the considered AMD architecture. Thus the impact of memory allocation on shared memory algorithms is clearly highlighted.

2 Context and motivation

Any application that tries to dynamically manage memory on a *NUMA* multicore system faces a set of issues that impact its performance. In this section, we highlight these issues. We also explain how the complexity and the heterogeneity of these challenges motivate the need beneath the comparison of allocators: an inefficient allocator may dramatically influence the evaluation of thread-synchronization mechanisms.

2.1 Complex processor architecture

Let us consider a *Non Uniform Memory Access (NUMA)* processor (see Figure 1). To maintain a set of data coherent and up-to-date over such an architecture, an important time and space overhead is required. Indeed, this architecture represents a complex structure with several levels of caches which need to stay coherent [11]. Coherence property has to hold despite the different cores accessing the caches asynchronously.

The multicore processor architecture that we consider (NUMA) are made of multiple components in-

[1]Allocator that outperforms the time efficiency of general-purpose allocators by using a relatively large and thread local dynamic memory. Hence, such an allocator cannot be used as a general-purpose one.

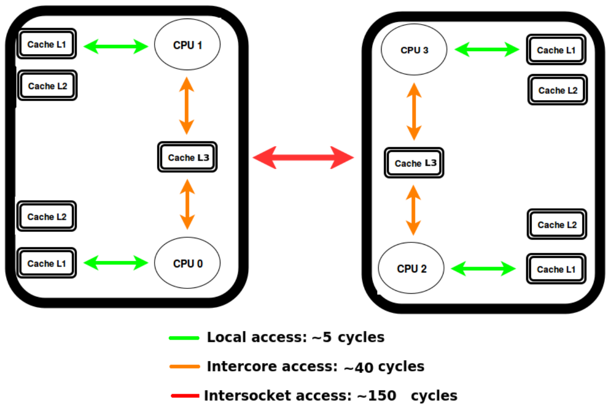


Figure 1: Simplified NUMA model and latency on *Intel Sandy Bridge* architecture

terconnected by a network-on-chip. Thus, any interaction between two independent components involves network communication which, in turn, induce a non negligible overhead.
When a thread tries to reach an address kept by its current L1 cache, the access may be done in a relatively short time (about 2 CPU cycles). However, this value may dramatically increase when the address is kept by a foreign core: up to 40 CPU cycles when the core is on the same NUMA node (L3 cache) and 150 CPU cycles when the core is on a foreign node (socket communication).

2.2 False sharing, memory blowup and heavy kernel operations

The allocation and the liberation of dynamic memory is unpredictable and asynchronous[11]. Thus, after a period of time, any set of dynamically-used memory may become fragmented: relatively small blocks of free and used memory are highly interleaved.
Dealing with a fragmented memory may impact the allocation performance: When the user requires a given block of memory, the allocator may have no free chunk of memory large enough to satisfy the request, while the total free memory may still be enough. Depending on the allocator implementation, the request may remain unsatisfied, or the process heap may be dynamically expanded. But in all cases a non negligible delay is appended to the allocation time. This performance deterioration is known as ***memory blowup***.

Another performance drop, known as ***false sharing***[2] may appear at access time. Let us consider two separate objects accessed by two different threads running concurrently on two different cores. Let us also suppose that the two objects are stored in the same cache line (see Figure 2). As these objects have different address ranges, they could safely be accessed simultaneously. However, as the access granularity in the cache is a cache line (typically 64 Bytes), the cache controller will execute these accesses sequentially, increasing thus the total access time.

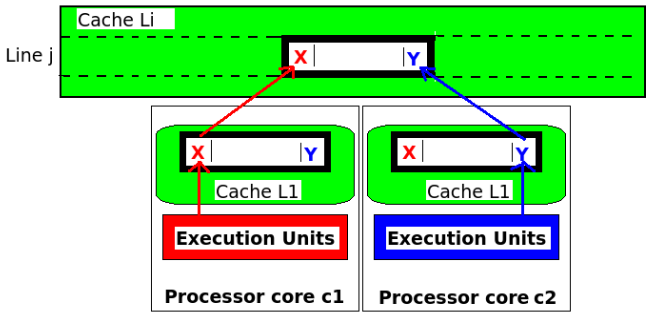


Figure 2: False sharing representation

Finally, the memory is a resource shared by independent processes running on separate cores. For memory coherence reasons, managing this memory must be done at kernel level. Thus, any memory allocator is subjected to the use of a set of heavy operations, often involving system calls.

2.3 Testing a new algorithmic approach: Delegation for mutual exclusion

Our paper follows the work of *D. Petrovic*, *T. Ropars* and *A. Schiper* [9]. In this former paper, an algorithm, based on the ***delegation approach*** to implement mutual exclusion is described (see Section 4.1). This algorithm is experimentally shown to achieve "1.4x (resp. 2x) higher throughput compared to the most efficient state-of-the-art delegation solution" [9].

One limitation of this study is that the experimental tests of the algorithms have been conducted using a *custom allocator*, built in order to allocate and free memory in short and roughly constant time. The principle of this allocator is to pin to each thread a block of memory at creation time. The dynamic memory management is optimized by always allocating fix-sized chunks from this block. No address of this block is freed until the thread destruction. As such, this allocator is ideal for this study. However, it can not be used for general purpose applications.

Regarding the wide scope and the complexity of the challenges associated with dynamic memory allocation, one can wonder how the use of a *general-purpose allocator* could affect the performance and behavior of the delegation algorithms. To answer this question, we need to identify the most suitable general-purpose allocator for our workload.

3 State of the art

As many-core machines have become widely used, different memory allocators [10; 2; 1; 7; 3; 5] have been proposed to try to fit the needs of this programming paradigm. Each of these implementations has either proposed some new memory-allocation strategies or has improved an existing one. In this section, we give an overview of some of these strategies. We highlight their advantages and link them to some existing implementations.

3.1 Delegate memory allocation to user space program

Among the set of functions that an operating system (OS) proposes to a user program, the ones that trigger a system call are probably the costlier in terms of time (only considering the non blocking instructions). They imply a switch to kernel mode, increasing the execution time of the task.

In the *Linux OS* C standard library ***glibc*** [10], the memory-allocation function regularly re-sizes a process heap at runtime. Thus, allocation functions may trigger system calls[2]. To decrease to almost zero the number of these system calls, all the memory allocators that we will consider use the same strategy: a large chunk of memory is initially allocated to all user programs. Allocation is then done at user level by simply splitting this initial chunk of memory.

This method dramatically decreases the number of system calls. Yet, when the initially-allocated memory is exceeded, user allocators may trigger a system call to seek the OS for a new chunk of memory (as in ***Hoard*** [2], ***tcmalloc*** [5] or ***SuperMalloc*** [7]) or to answer each new allocation request made by the user (as in *Scalloc* [1]).

3.2 Core locality allocation

Within *UNIX* processes, process heap is a resource shared by many threads. Thus, to ensure consistency, synchronization is required between threads that try to access it, which obviously introduces a time override.

To decrease this synchronization override, ***Hoard*** [2], which is one of the first thread-scalable allocators, has introduced the idea of core locality: each processor core is mapped to a unique buffer. Threads running on this core are the only one authorized to allocate memory in this buffer. Hence, allocation requires no synchronization between threads from different cores. ***tcmalloc*** uses an adaptation of this method where a buffer is mapped to each thread. The core local principle has been further improved by ***Scalloc*** [1] by adapting the data structure used for the buffer (see Section 3.4)

One of the main advantages of the core local buffer is the increasing probability that a thread triggers a page fault. Indeed, a buffer is allocated as a block of contiguous addresses. All the threads running on a given core allocate memory within the same address range (addresses of the buffer). Thus the probability to trigger a page fault is low, even after a context switch.

3.3 Dealing with the shared pool contention

To feed each core-mapped buffer, an allocator may reuse a freed memory previously belonging to the same buffer (with no synchronization over-cost). It also needs a structure supplying an empty buffer request (see Figure 3).
This structure is made of one stack per core. Hence, it may be safely accessed by all the core-local buffers simultaneously and without any synchronization.

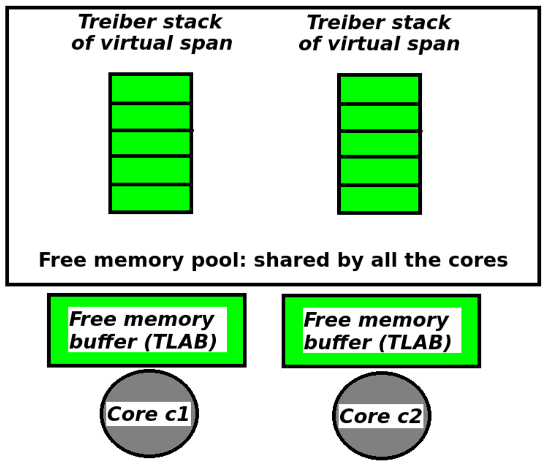

Treiber stack of virtual span
Treiber stack of virtual span
Free memory pool: shared by all the cores
Free memory buffer (TLAB)
Free memory buffer (TLAB)
Core c1
Core c2

Figure 3: Free memory pool: shared by all the cores. *TLAB*: Thread Local Allocation Buffer

One important efficiency factor of this shared structure is the number of requests it makes to the OS kernel to get more free memory. To reduce this number, the policy introduced by ***supermalloc*** [7] is to maintain the fill rate of all the core-mapped stacks roughly equal. The OS kernel requests are only made once the average level of the stacks reaches a given threshold value.
Using such a policy, a core-mapped stack that runs out of memory will very likely be filled from another stack. Contention on the core-mapped stacks has been lightened by ***scalloc*** [1] through the usage of the non blocking *Treiber stack* [6]. Similarly, ***tcmalloc*** uses a lock-free multilevel linked list.

These strategies have made it possible to remove almost all synchronization for allocation operations, making them having a roughly constant number of micro instructions. However, a chunk of memory allocated by a given thread may be freed by a different one. Thus, to avoid a synchronization over-cost for the *free* operation, ***scalloc*** [1] does not return a freed memory to the shared pool. It places it instead into the buffer of the thread which runs the *free* function.

3.4 Virtual span

So far, we have considered solutions that improve the efficiency of memory management by adapting the *allocation* and *free* operations. Further improvements can be obtained by reducing the access time to a given address.

Within ***scalloc*** [1], ***jemalloc*** [3] and ***tcmalloc*** [5] , the free memory is organized using the principle of so-called *virtual span*[3]. This structure consists of a fixed-size block of contiguous addresses. Nine different size classes exist (), each of them is a multiple of 4KB

[2]Call to the kernel functions *brk*, *sbrk* and *mmap*

[3]Also called *superblock* (resp *superpage*) for *Hoard* [2] (resp *jemalloc* [3]) implementation.

(the size of a system page). An allocation results in the use of one independent span, no matter the size required by the user. The payload of the object is stored into the span. Additionally, some extra memory is used for alignment and uniformity purposes.

This architecture has been designed to improve *space locality*, avoid *fragmentation* and simplify *complex data structures* [4].

Space locality
The operating systems that we are considering implement *virtual paging*. Thus, a major concern for time efficiency is that memory addresses, which are more likely to be used together, may belong to the same page. This property reduces the number of page faults, which is a major time consuming factor.
As a virtual span is a set of contiguous addresses, it increases the probability that two addresses of the same object (allocated by the same call to the "allocation" function) belong to the same page.

Fragmentation and complex data structures
Storing exclusively a few number of size-range objects has an obvious interest for avoiding fragmentation and simplifying data structures used for management. On the other hand, as we treat uniformly large and small objects, an important extra memory may be required for small objects. However, thanks to the paging system, this unused memory is swapped out of the live memory (RAM) and will never trigger a page fault (never accessed by the user). Thus, the pagination system limits the time and space impact of the extra memory used by a span.

4 Experimental setup

It is noteworthy that the results of our experiments are highly dependent on the processor architecture and the operating
In this section, we explicitly define the hardware and software environment used to obtain the presented results. We also define the algorithm that we aim to evaluate, as well as the rules of the evaluation.

4.1 Custom delegation algorithms for synchronization

The algorithms we want to evaluate [9] implement a critical section using the *delegation* principle: given a critical section CS, each thread that tries to access it will send a request to a dedicated centralized thread (server). The server will answer the requests sequentially, by being the only one to access (and realize) the CS.

The first considered delegation algorithm is the ***combiner algorithm***. In this algorithm, the server thread is not a specific dedicated one. Indeed, when different client threads try to access a CS, one of these threads is elected to execute its own CS as well as the one of a certain number of other threads.

The other considered algorithm implements two main improvements to the delegation method. The first one consists in the use of a ***back-off*** on client side in order to reduce the overhead due to a remote memory reference: Let us consider a client thread that has written a request to the server, and which spins on a specific memory address M, waiting for the answer. The first improvement consists in adjusting this spinning rate to the *prefetcher* rate, in order to delay the first read access of the client. Thus, we increase the probability that the server accesses the CS between the client's write and the first client's read. As the server holds a *read write* access on M during this period, it may write the answer without updating the client's cache. Hence avoiding the server to trigger a costly remote memory reference to the client.

The second improvement consists in using a non-temporal memory access (***Streaming store***) for the communication between the clients and the server. Therefore, the server avoids waiting for the write operation to be effective.

4.2 Hardware and software environment

The results that are presented in this paper have been obtained on two x86 machines (accessed through the test-bed *Grid5000*). The first one is an *Dell Poweredge C6220* consisting of two 10-core *Intel Xeon E5-2660v2* (2.2GHz) chips. The second one is an *HP Proliant DL165 G7* consisting of two 12-core *AMD Opteron 6164 HE* (1.7 GHz) chips.
On both machines, a linux (3.16.7-4) operating system has been used based on the Debian kernel 3.16.7. The default kernel allocation policy (first touch) has been used to obtain all the presented results.
All the programs that implement the CS algorithms and the shared data structures have been implemented in C language. They have been compiled using *gcc 4.4.7*. The option O3 (maximum optimization level) has been used for all the compilation process.

For the experiments, the following memory allocators have been considered : *glibc* 3.74, *Hoard* v2.2.1, *scalloc* (released in August 2015), *Supermalloc* (released in June 2015), *gperftool-tcmalloc* v 3.2.1 and *jemalloc* 4.2.1.

4.3 Testing environment and rules

The experimentation that we lead have all been conducted using applications built on the same principle: A fixed number of threads concurrently access a queue. The queue is a *Michael Scott queue* [8] synchronized using one of the presented CS delegation algorithms. Each thread loops on the *enqueue* and *dequeue* actions. Each appended/removed element is allocated/freed dynamically.
Our implementation ensures that:

- Each application uses less threads than the number of cores available on the machine. This constraint helps in lightening the waiting time due to scheduling.

[4] *False sharing* is also dramatically reduced thanks to the virtual span. An experimental evaluation of this improvement is available in [1]

- Each thread is pinned to a distinct core. This constraint eliminates the cost of reaching an address owned by a cache within another core. It also eliminates the OS scheduler inference on the performance.

5 Results

In this section we use the *throughput* (number of queue access operations per second) as metric for our comparative study. Other metrics, such as the *latency* or the *inter-core fairness*, have been considered. However, they have been omitted on purpose in this paper as they did not add any significant value to the study.

5.1 Choosing the most efficient memory allocator

First, we evaluate the impact of the memory allocator on the throughput of our test application. In the current case, the shared queue is implemented using the *combiner algorithm*. Figure 4 shows the throughput of this application when different allocators are used.

We can notice in this figure three main groups of allocators, gathered according to their performances:
On one hand, the *custom* allocator which, as expected, outperforms all the others from 20% up to 530%. Respectively, the *ptmalloc* allocator underperforms all the allocators from 100% up to 500%. On the other hand, the general purpose allocator may be sorted according to their implemented strategies:

- *Hoard*, which only implements the core local buffer, is the least efficient one. It has also the least stable behavior due to the non reuse of the freed memory: after a given period of time, the core local buffers get empty, and a system call (sequential) is required to feed them. As the importance of this phenomenon increases with the number of concurrent threads, this explains the dramatic drop of the *Hoards*'s curve after 10 threads.
- *SuperMalloc, scalloc* and *jemalloc* may be considered as equivalent. The difference that can be observed in their curves are due to the management of the released memory. Indeed, *jemalloc* places it into the core local buffer with no synchronization (one thread per core); whereas *SuperMalloc* and *scalloc* place the released memory into the shared pool which requires a synchronization with the other threads on the other cores.
- Similarly to the previous allocators, *tcmalloc* uses a thread local buffer and a shared free data structure. The efficiency gain that we notice comes from the use of a *garbage collector*: Thanks to this asynchronous and T periodic feature, the free operation is equivalent to a simple boolean variable modification. The overhead linked to the free operation is payed only once at each period T.

5.2 Custom delegation algorithm performance using a real allocator

Thanks to the previous experimental results, we can definitely choose *tcmalloc* as an allocator for the benchmark study of the considered delegation algorithms.

Using this allocator, we present in Figure 5 the performance comparison of a shared queue implemented using each of the considered delegation algorithms.

Our method confirmed the results obtained by *D. Petrovic et al* [9] on the considered Intel architecture. The optimized delegation algorithm reaches, in this specific condition, a higher throughput than the *combiner* algorithm (about 10%). And this result holds using both *custom* and *tcmalloc* allocators.

However, our experimental results infirm the claim of *D. Petrovic et al* [9] on the considered AMD architecture. Indeed, within this specific conditions, the combiner algorithm reaches a higher throughput.

5.3 Discussion

From our experimental results, two noteworthy results may come out. On one hand, the experimental comparison of the allocators shows the highly significant difference between the custom allocator used in the former test of the delegation algorithm, and the general purpose allocators. It also shows how stable is the behavior of this custom allocator compared to the others. This justifies the need of the current benchmarking study in order to validate any performance property of the custom delegation algorithms.

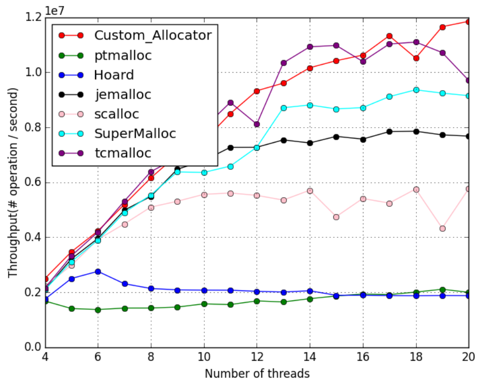
On the other hand, all the conclusions that come out of our experimental results are intrinsically dependent on the two considered hardware platforms and kernel policies. Indeed, the comparison between our experiments and the one conducted by *D. Petrovic et al* shows an obvious difference between the maximum threshold performance reached by the same algorithm (considering the same number of threads). Such a difference may be principally explained by the different number of NUMA nodes used during the two experiments: the highest is the number of NUMA nodes, the more the hosted application triggers costly remote memory references.

6 Conclusion

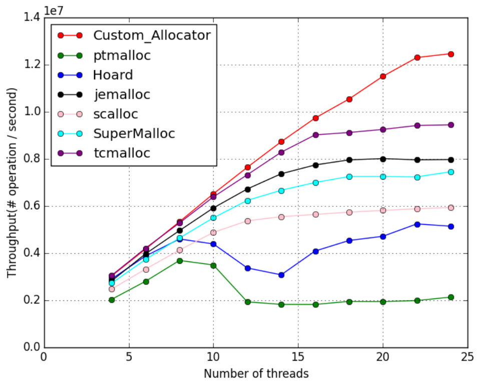
The current paper presents our method to prove the impact of memory allocation on the performance of two custom delegation algorithms. Our evaluation shows that such an impact does not corrupt the expected behavior of the delegation algorithms. However, we have proved that depending on the considered hardware platform, the validation of an improvement of the delegation approach may be lightened or even go on default.

References

[1] M. Aigner, C. M. Kirsch, M. Lippautz, and A. Sokolova. Fast, multicore-scalable, low-fragmentation memory allocation through large virtual memory and global data structures. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 2015.

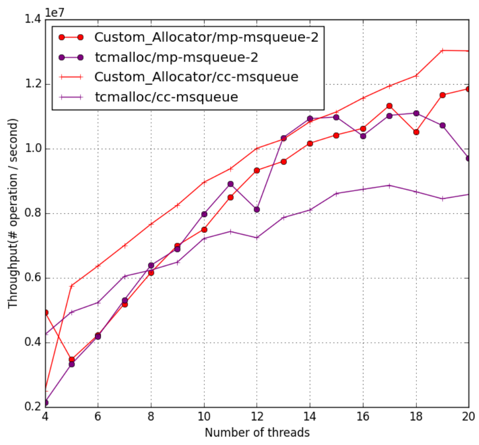


(a) Intel Xeon E5 2620

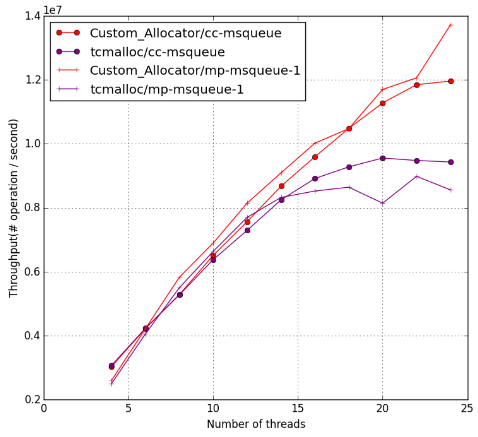


(b) AMD Opteron 6164 HE

Figure 4: Impact of the memory allocator on the performance of multithreaded application



(a) Intel Xeon E5 2620



(b) AMD Opteron 6164 HE

Figure 5: Delegation algorithm comparison using the tcmalloc allocator. Custom-Allocator used as a benchmark. *cc-msqueue*: shared queue implemented using the combiner algorithm. *mp-msqueue-2*: shared queue implemented using the backoff/streaming delegation algorithm

[2] E. D. Berger, K. S. McKinley, R. D. Blumofe, and P. R. Wilson. Hoard: A scalable memory allocator for multithreaded applications. *ACM Sigplan Notices*, 2000.

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

[4] T. B. Ferreira, R. Matias, A. Macedo, and L. B. Araujo. An experimental study on memory allocators in multicore and multithreaded applications. In *Parallel and distributed computing, applications and technologies (pdcat), 2011 12th international conference on*. IEEE, 2011.

[5] S. Ghemawat and P. Menage. Tcmalloc: Thread-caching malloc, 2007.

[6] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*. ACM, 2004.

[7] B. C. Kuszmaul. Supermalloc: A super fast multithreaded malloc for 64-bit machines. In *Proceedings of the 2015 ACM SIGPLAN International Symposium on Memory Management*. ACM, 2015.

[8] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*. ACM, 1996.

[9] D. Petrović, T. Ropars, and A. Schiper. On the performance of delegation over cache-coherent shared memory. In *Proceedings of the 2015 International Conference on Distributed Computing and Networking*, ICDCN '15, 2015.

[10] W. K. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time detection of heap-based overflows. In *LISA*, volume 3, 2003.

[11] D. J. Sorin, M. D. Hill, and D. A. Wood. A primer on memory consistency and cache coherence. *Synthesis Lectures on Computer Architecture*, 6, 2011.