Using Elimination and Delegation to Implement a Scalable NUMA-Friendly Stack

Irina Calciu
Brown University
irina@cs.brown.edu

Justin E. Gottschlich
Intel Labs
justin.e.gottschlich@intel.com

Maurice Herlihy
Brown University
mph@cs.brown.edu

Abstract

Emerging cache-coherent non-uniform memory access (cc-NUMA) architectures provide cache coherence across hundreds of cores. These architectures change how applications perform: while local memory accesses can be fast, remote memory accesses suffer from high access times and increased interconnect contention. Because of these costs, performance of legacy code on NUMA systems is often worse than their uniform memory counterparts despite the potential for increased parallelism.

We explore these effects on prior implementations of concurrent stacks and propose the first NUMA-friendly stack design that improves data locality and minimizes interconnect contention. We achieve this by using a dedicated server thread that performs all operations requested by the client threads. Data is kept in the cache local to the server thread thereby restricting cache-to-cache traffic to messages exchanged between the clients and the server. In addition, we match reciprocal operations (pushes and pops) by using the rendezvous elimination algorithm before sending requests to the server. This has the dual effect of avoiding unnecessary interconnect traffic and reducing the number of operations that change the data structure. The result of combining elimination and delegation is a scalable and highly parallel stack that outperforms all previous stack implementations on NUMA systems.

1. Introduction

The current trend in computer architecture is to increase system performance by adding more cores so that more work can be done simultaneously. In order to enable systems to scale to hundreds of cores, the main hardware vendors are switching to non-uniform memory access (NUMA) architectures. Recent examples include Intel's Nehalem family and the SPARC Niagara line.

NUMA systems contain multiple sockets connected by an interconnect. Each socket (also called a node) consists of multiple processing cores with a shared last level cache (LLC) and a local memory (as in Figure 1). A thread can quickly access the local memory on its own socket and it can access the memory on another socket using the interconnect, so the programming model is similar to uniform memory architectures. The NUMA design allows systems to scale to hundreds of cores and provides inexpensive data sharing for cores on the same socket. However, remote cache invalidations and remote memory access can drastically degrade performance because of the interconnect's high latency and limited bandwidth. Therefore, in many cases, legacy code exhibits worse throughput when ported to NUMA machines than on non-NUMA ones.

Prior research addresses this by using a NUMA aware contention manager that migrates threads closer to the data they access [1]. However, migrating threads is a complex solution that, while feasible for operating systems, is not generally realistic for end-user applications. Alternatively, one could devise solutions in which the data are moved to the accessing threads. For example, cohort locks [2] and NUMA reader-writer locks [3] keep the data local to one cache as long as possible. This is implemented by transferring ownership of the locks from the threads finishing their critical sections to other threads on the same socket. Similarly, Metreveli et al. [4] minimize cache data transfers by partitioning a concurrent hash table and distributing operations for each partition to a specifically assigned thread. All threads wanting to access the hash table submit requests to these server threads through message passing implemented in shared memory. Essentially, the hash table resides in the caches of the accessing threads and the cache-to-cache traffic is limited to requests sent to and from the servers.

Making Data Structures NUMA-Friendly. To maximize performance, Metreveli et al. [4] leverage the concurrency properties of hash tables in their partition implementation. Namely, hash tables are highly concurrent, easily partitionable data structures. However, many data structures do not have the inherent concurrency benefits of hash tables. In this paper, we focus on a NUMA-friendly implementation of a stack. Nevertheless, the method presented can be applied to other data structures as well.

Stacks have a broad range of uses: from calculators to evaluate expressions to compilers to parse the syntax of expres-

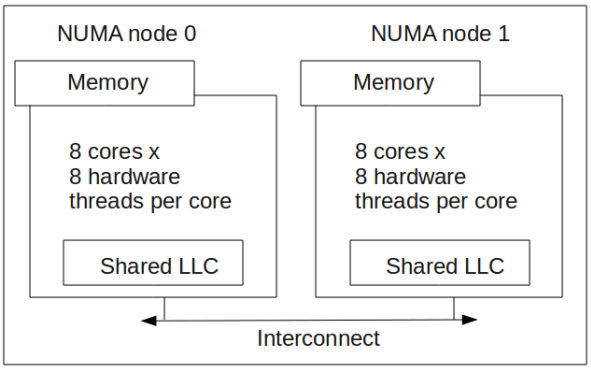

Figure 1: Example of a NUMA system with two nodes and 128 hardware threads.

sions and program blocks. In addition, stacks can easily be used to implement unfair thread pools and any containers without ordering guarantees. An example is the Java unfair synchronous queue [5].

Unfortunately, stacks cannot be easily partitioned without forfeiting their last-in-first-out (LIFO) property. Because of this, multiple threads often contend on the single entry point providing access into the stack. It is primarily for this reason that stacks seem to be inherently sequential. However, prior work has shown that stacks can benefit from parallelism under balanced workloads (i.e., a similar number of push and pop operations) using a method called elimination [6, 7]. This is implemented by canceling concurrent inverse operations from different threads even before they reach the stack. Elimination is not specific to stacks. Moir et al. [8] have shown how to use elimination with queues. Although this method significantly improves scalability of stacks, it does not address our primary concern: i.e., remote cache invalidations on NUMA systems.

We combine a variation of this method and a slightly modified version of the delegation method introduced by CPHash [4] using only one server thread, to design what is, to the best of our knowledge, the first NUMA-friendly stack. We describe in more detail how we use elimination in Section 2.2 and delegation in Section 2.1.

Our goal is to reduce cache traffic and maintain data locality while using the properties of the underlying data structure to enable parallelism. The result is a scalable and highly parallel stack that outperforms all previous stack implementations on NUMA systems.

The technical contributions of this paper are as follows. 1) We explore previous concurrent stack implementations in the context of NUMA systems; 2) We design, implement, and evaluate a scalable concurrent stack optimized for NUMA machines by delegating responsibility for all requests to one server thread, keeping the data local to this thread's cache, avoiding synchronization, contention and cache to cache traffic; 3) We enable parallelism by using elimination.

2. NUMA-Friendly Stack

In this section we describe our use of delegation to implement a NUMA-friendly stack. At the highest level, our design provides efficiencies in increased cache locality and reduced interconnect contention. After discussing the design, we show how we employ the rendezvous elimination algorithm [7] to make this stack scalable. Moreover, we differentiate between global elimination, which is implemented using one rendezvous structure shared by all threads, and local elimination, which contains an elimination structure for each NUMA node.

2.1 Delegation

The idea of one thread helping other threads complete their work is a well-known concept [4, 9–14]. A recent example of this *helping* mechanism is called flat combining [9], in which a thread that acquires a lock for a data structure executes operations for itself and also for other threads that are waiting on the same lock. The global lock and the data remain in this thread's cache while it executes operations on behalf of other threads, thereby decreasing the number of cache misses and contention on the lock. Moreover, flat combining aligns well for data structures that are sequential, because only one thread would be able to operate on it at a time, regardless.

Due to the increasing number of hardware threads in a system, the helper thread could be a dedicated thread (called a server thread) used only to service requests from other threads (client threads). This is especially useful on heterogeneous architectures, where some cores could be faster than others. An example of this approach is CPHash [4], a partitioned hash table. Each partition has an associated server thread that receives add and remove requests from clients and sends back the responses obtained from performing the operations requested. Each client-server pair share a location where they exchange messages.

We use this delegation approach to implement a NUMA friendly stack. In particular, we use one dedicated server thread that accepts push and pop requests from many client threads. The communication is implemented in shared memory, using one location (which we call slot) for each client. The server loops through all the slots, collecting and processing requests and writing the responses back to the slots. The clients post requests to their private slots and spin-wait on that slot until a response is provided by the server. Figure 3a provides a high-level overview of this communication protocol.

We note that only the pop operations need to spin-wait until a response is provided. The push operations could return as soon as the server notices their requests. This optimization improves throughput, but we decided not to use it in our experiments, for a more fair comparison with the other methods.

A weakness of this design is that using a reserved slot for each client can result in wasted space if the clients' workloads are not evenly distributed. Furthermore, the server must loop through all slots, even those not in use, when looking for requests. These two drawbacks can result in increased space and time complexity. To overcome these limitations, we statically assign several threads to the same slot by thread id. [1] To synchronize the access of multiple threads to the same slot, we introduce an additional spinlock for each slot. Figure 3b reflects these changes to the communication protocol. Figure 2 shows the overall interaction between the server and the clients.

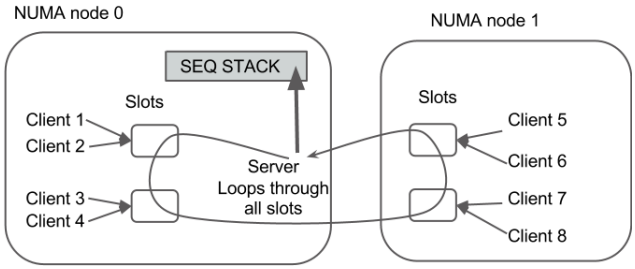

Figure 2: Clients posts their requests in shared local slots and wait for the server to process them. The server loops through all the slots, processes requests as it finds them and immediately posts back the response in the same slot. The sequential stack is only accessed by the server thread; therefore, the top part of the stack remains in the server's L1 cache or LLC all throughout execution.

2.2 Elimination

Stacks are generally seen as sequential data structures. This is because all threads contend for access to the stack at its top location. However, prior work has shown that stacks can be parallelized using a technique called elimination [6]. This technique uses an additional data structure to allow threads performing push operations to meet threads performing pop operations and exchange their arguments. This is equivalent to the push being executed on the stack and immediately being followed by a pop. The elimination data structure, generally implemented as an array, allows multiple such pairs to exchange arguments in parallel and decrease contention on the underlying lock-free stack. If one thread fails to find its inverse operation being performed by another thread, then the elimination attempt times out and the thread accesses the stack directly.

The rendezvous method [7] improves the elimination algorithm by using an adaptive circular ring for the additional elimination data structure. In this paper, whenever we refer to an elimination layer, we use a rendezvous structure to implement it.

Elimination generally works best when the number of inverse operations are roughly equivalent. For inequivalent, unbalanced workloads, many operations cannot be eliminated, thereby requiring a thread to access the data structure directly. This creates contention and cache-to-cache traffic because these operations could originate from different NUMA nodes. In order to solve these problems, we augment the delegation stack presented in the previous section with a rendezvous elimination layer. Threads first try to eliminate and, if they time out, they delegate their operation to the server thread. Delegation ensures that the data remains in the server's cache, while elimination enables parallelism, thus making the NUMA-friendly stack more scalable. Moreover, threads can continue to try to eliminate while they wait for the spinlock of their slot to be released. The complete algorithm is described in Figure 3c.

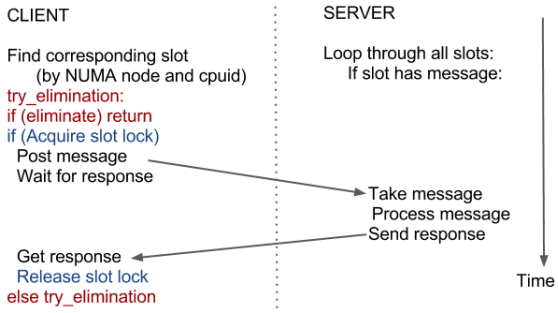

Figure 3: Communication protocol between a client thread and the server thread using slots

(a) (Black) Single thread per slot: each thread posts requests in its private slot, without any synchronization.

(b) (Blue) Multiple threads per slot: threads share slots, so they need to acquire the slot's spinlock before writing the request.

(c) (Red) Elimination: Threads first try to eliminate; if they fail they then try to acquire the slot spinlock and submit a request, but if the lock is already taken, they go back to the elimination structure; they continue this loop until either they eliminate, or they acquire the spinlock.

Local vs. Global Elimination. For the rendezvous stack, threads first try elimination and, in the case of failure, they then directly access the stack. Our NUMA-friendly stack is an improvement over this design, because it increases locality and reduces contention on the stack by replacing direct access to the stack with delegation. However, the initial stage

[1] It is important to note that all threads using the same slot need to be on the same NUMA node in order to maintain the slot's locality.

of elimination can still cause a number of invalidations between different NUMA nodes' caches because each of the threads accesses the same shared structure when performing elimination. To overcome this bottleneck, our NUMA-friendly stack splits the single elimination data structure into several local structures, equal to the number of NUMA nodes. To minimize interconnect contention, we limit elimination to occur only between those threads located on the same socket.

2.3 Advantages and Limitations

Our stack design is optimized for the NUMA architecture. Local elimination and delegation both contribute to removing the contention on the interconnect and on the stack. The number of remote cache misses is reduced to exchanging messages between the server and the clients. Furthermore, the stack is only accessed by the server thread, relieving it of any form of synchronization.

Finally, our stack uses explicit communication, so it is easier to reason about its performance penalties because underlying cache coherency performance issues are eliminated.

One potential drawback of this approach is that the access to the stack is serialized by using only the server thread. However, the direct access of multiple threads to a stack would also be serialized by a lock to keep the stack's integrity. Moreover, we enable parallelism by using elimination, which compensates for accessing the stack sequentially.

Another drawback is the potential for additional communication overhead between the clients and the server. For example, if the stack is only rarely accessed, then direct access to it would likely be more efficient. However, the overhead of elimination and delegation is eclipsed by their benefits when there are many threads contending for access to the stack.

Finally, our description assumes one server thread for each shared stack. In order to maintain high throughput, this thread must always be available to handle queries. Therefore, each server thread is assigned a hardware thread and runs at high priority. Unfortunately, we might not have enough hardware threads if an application uses multiple shared data structures, so some of the structures might have to share a server. In this situation, the server could become a performance bottleneck. However, most applications use only a few shared data structures.

3. Experimental Results

We conducted our experiments on an Oracle SPARC T5240 machine with two Niagara T2+ processors running at 1.165GHz. Each chip has 8 cores and each core has 8 hardware threads for a total of 128 hardware threads (64 per chip). We implemented our NUMA stack algorithm in C++ and we compared it to previous stack implementations using the same microbenchmark as [7]: a rendezvous stack, a flat combining stack and a lock-free stack. The benchmark has flexible parameters, allowing us to measure throughput under different percentages of push and pop operations. The results we present were obtained using threads with fixed roles (e.g. push-only threads and pop-only threads). We allow the scheduler to assign our software threads to NUMA nodes and then pin them to their respective processors. [2] The server thread is created with increased priority compared to the client threads, to guarantee its availability.

For our experiments, we started by comparing our local elimination and delegation NUMA stack (nstack_el) with a lock-free stack (lfstack) [15], which has been the basis for other stack implementations such as rendezvous [7] and flat-combining [9]. Then, we compared our stack to the flat combining stack (fcstack) [9], which outperforms the rendezvous stack when there is no significant potential for elimination (i.e., in unbalanced workloads).

The scalable performance of the lock-free stack begins to degrade around 16 threads. The flat-combining stack, however, seems unaffected by the type of workload and achieves relatively stable scalability across different thread counts. However, the elimination based NUMA stack outperforms both of them by a large margin. These results can be observed in Figures 4, 5 and 6.

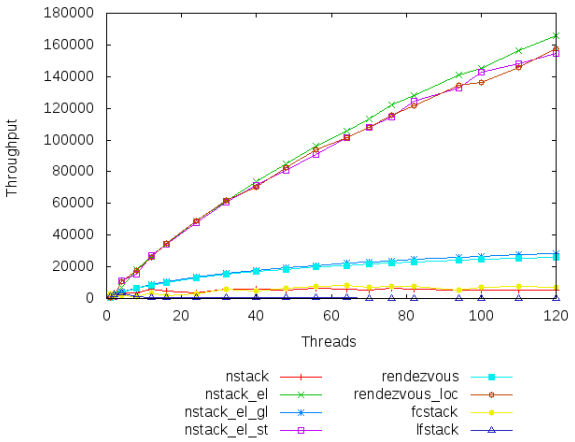

Figure 4: Results for 50% pushes and 50% pops

Effect of elimination. To judge the effect of the local elimination structures used in our implementation, we compared our NUMA stack (nstack_el) against two other versions; one without elimination (nstack) and one with global elimination (nstack_el_gl). As expected, the global elimination algorithm

[2] We also experimented with unbounded and variable role threads, but the results were too similar to warrant inclusion in this paper.

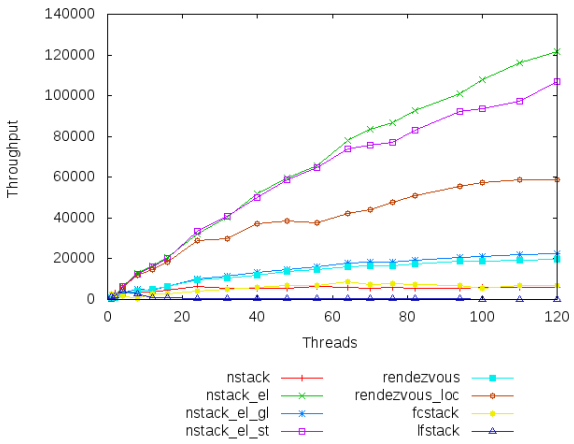

Figure 5: Results for 70% pushes and 30% pops

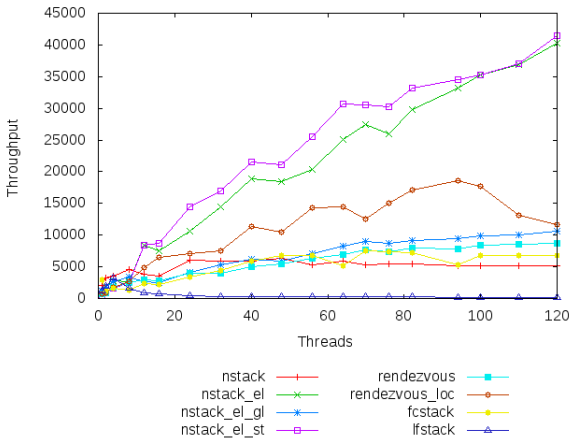

Figure 6: Results for 90% pushes and 10% pops

outperforms the algorithm without elimination, while both perform worse than local elimination. From Figures 4, 5 and 6, we conclude that local elimination is crucial for the scalability of our algorithm. Our experiments were performed on a 2-node NUMA system, but we expect that these results generalize to bigger systems with the same number of cores per node as our system, as long as the push and pop operations are distributed uniformly across all the nodes.

Effect of delegation. To better understand and characterize the impact of delegation, and because elimination has such a strong influence on performance, we compare our stack against two *elimination-variations* of the rendezvous stack: one uses local elimination and the other uses global elimination. The rendezvous stack (rendezvous) consists of global elimination and direct access. To provide a more fair comparison, we modified the rendezvous stack to perform elimination locally on each NUMA node (rendezvous_loc). Threads that fail to eliminate on each node must access the data structure directly. This local version of the rendezvous stack improves the scalability of the rendezvous stack for NUMA systems. However, our NUMA stack performs even better, indicating there is an observable performance benefit using delegation under high contention, for both balanced and unbalanced workloads (Figures 4, 5 and 6). We believe the benefit of delegation would become more apparent on a NUMA system with more sockets. Although the latency of an individual operation could increase because the server needs to inspect slots on more nodes, cache and memory locality would play an even more significant role than they do on a 2-node system. We leave evaluation on such a system as future work.

Balanced workloads. We experimented with different percentages of push and pop operations. Elimination works best when the number of pushes is very similar to the number of pops. In the balanced workload case, we use 50% push threads and 50% pop threads. Experimental results are shown in Figure 4. For this setting, elimination plays a significant role, as most operations will manage to eliminate. There is some benefit from delegation, as we can see when we compare to the local rendezvous algorithm, but not that significant.

Unbalanced workloads. For unbalanced workloads, elimination plays a much smaller role in reducing the number of operations. We present results for 70% pushes, 30% pops in Figure 5 and 90% pushes, 10% pops in Figure 6. In both cases, there is some elimination, but not as significant as in the balanced workload case. However, delegation plays a much more important role for these workloads, as more operations fail to eliminate and need to access the stack. Results show that preserving cache locality through delegation works much better than direct access to the stack.

Number of slots. Finally, we want to measure the impact of the synchronization introduced with sharing slots by different threads. We compared the implementation of the NUMA stack using shared slots (nstack_el) with the implementation using one slot per client thread, which does not require any synchronization to access the slots (nstack_el_st - nstack elimination single thread per slot). The results indicate that there is no clear winner in this case, which can be explained by the fact that the server has to loop through all the slots to service requests. Each request might have to wait a linear time in the number of slots to be found by the server. If the server finds too many of the slots empty, then much of the work performed by the server is wasted. However, if the server finds requests in most of the slots, then the algorithm can benefit from more slots because of the lack of synchronization. Our results seem to support this claim: the single thread (ST) per slot version outperforms the multiple threads per slot version (MT) for very unbalanced workloads as in Figure 6, while MT outperforms ST for more balanced

workloads, as in Figures 4 and 5. This is due to the elimination algorithm significantly reducing the number of requests sent to the server for balanced workloads, while for unbalanced workloads there is less elimination and more requests sent to the server.

In our experiments, we assumed that we know the maximum number of client threads in the system and always check all the slots, even when running with fewer threads. This could be improved using an adaptive way of determining the number of slots, but we leave that as future work.

4. Discussion

Hardware's shift towards NUMA systems urges a compatible software redesign. Basic data structures are not optimized for these architectures. We propose the first NUMA-friendly design of a stack, using local elimination and delegation. Combining these two methods is favorable across a number of scenarios: elimination works best when the number of pushes and pops is roughly the same, while delegation significantly reduces contention in the cases in which there is not enough potential for elimination because the workload is not very balanced. Our NUMA-friendly stack outperforms prior stack implementations across different scenarios from completely balanced workloads to the more unbalanced ones.

However, this is just the first step in transitioning to NUMA systems. There are vast and exciting opportunities for exploring the design of other NUMA-friendly data structures. We presented one technique and showed that it works well for a stack. The same technique could be applied to other data structures, such as queues and lists, which also admit inverse operations. In contrast, other data structures might not be suitable for elimination or might suffer from the serialized access of the server thread. For these data structures, we need to find new tools that allow us to redesign them for the NUMA space.

Acknowledgments

We thank Yehuda Afek and Tel-Aviv University for access to the NUMA machine we used for our experiments, Dave Dice for useful discussions and our anonymous reviewers for valuable feedback.

References

[1] Sergey Blagodurov, Sergey Zhuravlev, Mohammad Dashti, and Alexandra Fedorova. A case for numa-aware contention management on multicore systems. In *Proceedings of the 2011 USENIX conference on USENIX annual technical conference*, USENIXATC'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

[2] David Dice, Virendra J. Marathe, and Nir Shavit. Lock cohorting: a general technique for designing numa locks. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 247–256, New York, NY, USA, 2012. ACM.

[3] Irina Calciu, Dave Dice, Yossi Lev, Victor Luchangco, Virendra J. Marathe, and Nir Shavit. Numa-aware reader-writer locks. In *Proceedings of the 18th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '13, pages 157–166, New York, NY, USA, 2013. ACM.

[4] Zviad Metreveli, Nickolai Zeldovich, and M. Frans Kaashoek. Cphash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 319–320, New York, NY, USA, 2012. ACM.

[5] William N. Scherer, III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. *Commun. ACM*, 52(5):100–111, May 2009.

[6] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. *J. Parallel Distrib. Comput.*, 70(1):1–12, January 2010.

[7] Yehuda Afek, Michael Hakimi, and Adam Morrison. Fast and scalable rendezvousing. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Mark Moir, Daniel Nussbaum, Ori Shalev, and Nir Shavit. Using elimination to implement scalable and lock-free fifo queues. *SPAA*, 2005.

[9] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, pages 355–364, New York, NY, USA, 2010. ACM.

[10] Gokcen Kestor, Roberto Gioiosa, Tim Harris, Osman S. Unsal, Adrin Cristal, Ibrahim Hur, and Mateo Valero. Stm2: A parallel stm for high performance simultaneous multithreading systems. In Lawrence Rauchwerger and Vivek Sarkar, editors, *PACT*, pages 221–231. IEEE Computer Society, 2011.

[11] M. Aater Suleman, Onur Mutlu, Moinuddin K. Qureshi, and Yale N. Patt. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIV, pages 253–264, New York, NY, USA, 2009. ACM.

[12] Silas Boyd-Wickizer, Haibo Chen, Rong Chen, Yandong Mao, Frans Kaashoek, Robert Morris, Aleksey Pesterev, Lex Stein, Ming Wu, Yuehua Dai, Yang Zhang, and Zheng Zhang. Corey: an operating system for many cores. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 43–57, Berkeley, CA, USA, 2008. USENIX Association.

[13] Panagiota Fatourou and Nikolaos D. Kallimanis. Revisiting the combining synchronization technique. In *Proceedings of the 17th ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 257–266, New York, NY, USA, 2012. ACM.

[14] Y. Oyama, K. Taura, and A. Yonezawa. Executing parallel programs with synchronization bottlenecks efficiently, 1999.

[15] R. Kent Treiber. Systems programming: Coping with parallelism. Technical report, IBM Almaden Research Center, 2006.