A Study of the Behavior of Synchronization Methods in Commonly Used Languages and Systems

Daniel Cederman, Bapi Chatterjee, Nhan Nguyen, Yiannis Nikolakopoulos, Marina Papatriantafilou and Philippas Tsigas
Computer Science and Engineering
Chalmers University of Technology, Sweden
Email: {cederman, bapic, nhann, ioaniko, ptrianta, tsigas}@chalmers.se

***Abstract*—Synchronization is a central issue in concurrency and plays an important role in the behavior and performance of modern programmes. Programming languages and hardware designers are trying to provide synchronization constructs and primitives that can handle concurrency and synchronization issues efficiently. Programmers have to find a way to select the most appropriate constructs and primitives in order to gain the desired behavior and performance under concurrency. Several parameters and factors affect the choice, through complex interactions among (i) the language and the language constructs that it supports, (ii) the system architecture, (iii) possible run-time environments, virtual machine options and memory management support and (iv) applications.**

We present a systematic study of synchronization strategies, focusing on concurrent data structures. We have chosen concurrent data structures with different number of contention spots. We consider both coarse-grain and fine-grain locking strategies, as well as lock-free methods. We have investigated synchronization-aware implementations in C++, C# (.NET and Mono) and Java. Considering the machine architectures, we have studied the behavior of the implementations on both Intel's Nehalem and AMD's Bulldozer. The properties that we study are throughput and fairness under different workloads and multiprogramming execution environments. For NUMA architectures fairness is becoming as important as the typically considered throughput property. To the best of our knowledge this is the first systematic and comprehensive study of synchronization-aware implementations.

This paper takes steps towards capturing a number of guiding principles and concerns for the selection of the programming environment and synchronization methods in connection to the application and the system characteristics.

I. Introduction

Synchronization has always been a core research problem in parallel and concurrent programming. Synchronization is required to assure the correctness of multi-threaded applications, but it can also become a bottleneck for performance. It becomes even more crucial in the multi-core and many-core era when multiprocessor computers are widely used.

Modern processors are provided with machine instructions for synchronization primitives such as *test-and-set*, *compare-and-swap* and many more. Using them, several synchronization methods have been proposed in the literature, ranging from traditional lock-based methods, such as locks, semaphores and monitors, to non-blocking approaches, such as lock-free/wait-free synchronization and software transactional memory [1], [2], [3], [4]. Building on them, programming languages can now provide built-in support for synchronization constructs as either an API in the language (Java and C#) or as user-friendly libraries (e.g. Intel TBB, NOBLE [5] or PEPPHER [6]). This means that when selecting a language to write an application in, a programmer has implicitly chosen the synchronization constructs offered by the language API or language-specific third-party libraries. In addition, selecting a programming language to use also involves several other options, which in turn have their own importance to the performance of the concurrent applications. For example, C++ offers basic memory management functionality, but also allows the programmer to access low level memory. Java or C#, on the other hand, offer automatic garbage collection, but they limit direct access to the memory. Still, even after selecting a language, the programmer has a wide range of synchronization methods to choose from. We argue that selecting the best synchronization constructs to achieve the desired behavior is a non-trivial job, which requires thorough consideration of different aspects. Besides languages and their features, the selection is also governed by several other parameters and factors, and the interplay among them, e.g. the system architecture of the implementation platforms; possible run-time environments, virtual machine options and memory management support; and the characteristics of the applications.

The implementation hardware platforms have their own role to play in this context. Although the widely available multi-core processors are mainly based on a cache-coherent NUMA design, they differ in the way they have implemented multi-threading to exploit instruction-level and thread-level parallelism. These differences are not only in the size and speed of the cache, but also in the number of threads that can share resources simultaneously [7], the memory-controller mechanism and the inter-processor connector designs that are employed on and off the chip [8]. After selecting the language, the subsequent selection of a virtual machine and/or operating system, from a wide range of options, increases the complexity of the problem even further.

There are several available synchronization methods to select from. None of them is even close to be the silver bullet which can solve all the synchronization issues that the application developers have to address in all possible hardware and software environments in the domain of concurrent programming. In the literature a number of efforts have been made to evaluate such methods through micro benchmarks [1], [9], [10] as well as macro benchmarks [11], [12], [13]. These benchmarks try to rank synchronization constructs by measuring their potential for high throughput and also examine a subspace of the parameters that we examine in this paper. Evaluating synchronization mechanisms exclusively for high throughput [14] could give misleading results. For example, consider evaluating the throughput of a simple concurrent data structure, with little or no inherent potential for concurrency, using different synchronization methods. Among the methods that give the best throughput, methods that consistently favor the same set of threads to get access to the data structure, while leaving others to starve, have the potential to rank among the best. This underpins the importance to measure fairness of the synchronization methods for a particular application.

In this paper we evaluate different types of lock-based (from fine-grained to coarse-grained), as well as lock-free, synchronization methods with regard to their potential for high throughput as well as fairness. We will focus the discussion on how these two measurements relate to each other. The studied synchronization mechanisms are applied to two different types of data structures, that have different potential for concurrency. Considering the variation in contemporary multi-core architectures, the experiments are performed on two multiprocessor machines, one with two Intel (Nehalem) processors and another with four AMD (Bulldozer) processors. Further, to explore the variation due to the choice of language and runtime, as well as memory management, we have implemented the algorithms in C++, Java and C#. To the best of our knowledge this is the first head-to-head, systematic evaluation that considers the interactions among (i) the programming language and the language constructs that it supports, (ii) the system architecture where the application is running on, (iii) possible run-time environments, virtual machine options and memory management support, and (iv) characteristics of the applications.

Our experiments put forward an interesting observation that the change in the multi-threading model at the level of architecture brings a big difference in the behavior of synchronization primitives, even though the processors have comparable speed and inter-processor connection design. Furthermore, our experiments show that high performing synchronization methods may have very poor fairness, and a wise selection is very important to make a good trade-off between the two. We also show that the choice of memory management, runtime and operating system may significantly change the performance and behavior of a concurrent application. This paper takes a step towards improving methodologies for choosing the programming environment and synchronization methods in connection to the application and the system characteristics.

The structure of the paper is the following. In Section II we go through the different synchronization mechanisms that we have examined. In Section III we discuss the concepts of fairness and throughput and how they relate to each other. Here, we also give a new quantitative measure of fairness for synchronization-aware implementations and give arguments as to why the measurement we have selected is useful in this context. In Section IV we present the algorithmic designs of the data structures that were used in the experiments. Further in Section V, we present the design and the setup of the experiments, as well as the detailed architectures that we have chosen for implementation. Analysis of the results is presented in Section VI. Section VII concludes the paper.

II. Synchronization Methods

There exists a multitude of common methods for synchronization. These can be divided into different categories depending on what kind of progress guarantees they provide. If no progress guarantee can be provided, which is the most common case and holds true for most locks, the synchronization construct is said to be *blocking*. If a synchronization construct can guarantee that at least one thread, out of the contending set, can finish its operation in a finite number of its own steps, the construct is said to be *lock-free*. What lock-free synchronization means in practice is that a thread does not need to wait for another thread to finish.

Of this great variety of synchronization methods, some are quite popular and well established in the literature. Many of them are available through the API specification of some of the tested languages (Java, C#). Others can be easily implemented by a programmer in many languages, while some more complex ones are usually implemented in standard or third party libraries. To allow for comparison, the following synchronization methods have been implemented in a similar manner for all the programming platforms that we have examined:

- Test-And-Set-based lock (TAS) – Mutual exclusion is achieved by repeatedly trying to set a flag using an atomic exchange primitive. The thread that manages to set the flag is given access to the critical section.
- Test-Test-And-Set-based lock (TTAS) – To lower the number of expensive atomic operations, the value of the flag is read before attempting to change it. If it is already set, no atomic operation is needed.
- Array lock – The lock consists of an array of flags and an index to the first flag. Initially only the first flag is set. A thread trying to acquire the lock atomically increments the index and spins on the flag in the array that the old index was pointing to. When the flag is set, the thread can enter the critical section. Upon exiting, it sets its own flag to false and raises the flag for the thread waiting at the next index [15], [16].

- Lock-free – The lock-free implementations used depend on the specific data structures and for the cases of our study they are described in Section IV.

Moreover, in today's great need of concurrency, every programming environment provides their own toolset of internal libraries or implicit language constructs. They are usually well integrated and easy to use, while they can also be optimized by the underlying virtual machine or just-in-time compiler. Below them the host operating system can also provide valuable tools for synchronization. In detail, the following are the common platform specific methods for synchronization that we also consider in our study:

- Reentrant lock – The Reentrant lock, provided by Java's `concurrent.locks` package, comes in two variations; a simple and a fair one. The Reentrant lock is based on an internal waiting queue which is a variant of the CLH lock [17], [18]. Specifically, the nodes of the queue are used to block the competing threads, while every node that releases the lock that it owned signals its successor. However, an important design difference is that being the first node in the queue does not guarantee the lock acquisition, but only the right to contend for the lock itself. A thread that tries to acquire the lock first contends for it using a compare and swap (CAS) operation. If it fails, it gets enqueued. This first step is not performed when the fair version of the Reentrant lock is used. An interesting observation here is that this internal queue acts as a backoff mechanism for the lock's contention.
- Synchronized/Lock – In Java and C# every object is associated with an intrinsic monitor. The use of a `synchronized` or `lock` statement respectively, with a specified object as an argument before a block of code, assures that the execution of that critical section will not take place unless the object's monitor is locked. The actual monitor implementation is platform and virtual machine dependent [19].
- Mutex in C# – Compared to the `lock` keyword, the Mutex construct in C# is a heavyweight implementation with a high overhead, as it is designed to work across multiple processes. Mutex can be used to synchronize threads across processes and requires inter-process communications.
- Pthread Mutex (PMutex) in C++ – The Pthread mutex construct is available in the Linux kernel from version 2.6.x and above. It is implemented using Fast Userlevel Locking (Futex), created by Franke H. et al. [20]. A futex consists of a shared variable in user space indicating the status of the lock and an associated waiting queue in kernel space. In the uncontended case, acquiring or releasing a futex involves only atomic operations on its lock status word in user space. In the contended case, a system call into the kernel is required to add the calling thread to the waiting queue or to wake up any waiting processes.

III. Behavior: Throughput and Fairness

One of the most desired properties of a synchronization method is having high throughput. The more successful operations that can be achieved in a unit of time, the more efficient the method is. Throughput is one of the two main properties that we consider in our study.

As NUMA architectures are becoming the standard in industry, and different ways of Simultaneous Multi-Threading are being presented, fairness of synchronization constructs is becoming important. Possible differences in the access latencies of competing threads for a memory location may even lead some of them to starvation. In preliminary experiments we observed that between different architectures, under identical conditions, different levels of fairness were provided to threads that were competing for atomically swapping a memory location.

A relevant definition of fairness was introduced into this context by Ha et al. [21] comparing the minimum number of operations a thread had with the average number of operations of all threads. This helps distinguishing cases of starving or less served threads. For identifying the opposite cases we can compare the average number of operations with the maximum ones among the threads. Since our goal is to detect any unfair behavior, we use as a fairness measure the minimum of the above values, formally:

$$fairness_{\Delta t} = \min\left\{\frac{N \cdot \min(n_{i_{\Delta t}})}{\sum_i n_{i_{\Delta t}}}, \frac{\sum_i n_{i_{\Delta t}}}{N \cdot \max(n_{i_{\Delta t}})}\right\}$$

where $n_{i_{\Delta t}}$ is the number of successfully performed operations by the thread i, in the time interval Δt. Fairness index values close to 1 indicate fair behavior, while lower values imply the existence of a set of threads being treated differently from the rest. The fairness index achieves value 1 when all the threads perform equal number of operations, i.e. perfect fairness. The fairness index is 0 when at least one thread completely starves. For a critical analysis of quantitative measures of fairness, one may refer to the paper by Jain et al. [22].

IV. Case Studies

A. Data Structures

We study the synchronization behavior of two types of data structures: FIFO queues and hash tables. They are both widely used and represent data structures with different number of contention points. The queues we are using in our case study are the lock-based and the lock-free linked list based queues introduced by Michael and Scott [23]. The lock-based queue uses locks to grant the enqueuer/dequeuer mutually exclusive access to either the head or the tail of the queue. Two locking strategies are applied to the lock-based queue: *coarse-grain* and *fine-grain* locking. The coarse-grained lock-based queue uses only one lock for both the head and the tail, while the fine-grained one uses two different locks, one for each of them. Hereafter, we refer to them as *coarse-grained queue* and *fine-grained queue*, respectively. The lock-free queue uses the CAS synchronization primitive to atomically modify the head or the tail without any locking mechanism.

The second case study is on the hash table data structure. The hash table we used is implemented as an array of buckets, each one pointing to a linked list that contains the key-value pairs which are hashed to the same bucket. The hash tables provide search, insert and remove operations. Insertion, removal or search for a key operate only on the linked list associated with the bucket to which the key is hashed to. This is where the synchronization is required. Both a lock-based and a lock-free hash table are implemented. The lock-based version has one lock for each bucket which, once locked, provide mutually exclusive access to the associated linked list. The lock-free version uses the implementation introduced by Maged Michael [24]. In this implementation, insertion of an item, i.e a node between two nodes in a linked list, is done with the help of a CAS to atomically swap the next pointer of the previous node to the new node. A thread which wants to remove a node first marks the last bit of the pointer to that node, so that other concurrent operations know its intention. Then the node is removed by using CAS, to make the previous node point to the next node. The design is proved to be correct and lock-free [24]. The reader can refer to that paper for more technical details.

B. Programming Environments

In our study of the behavior of synchronization methods, we have examined three different programming environments, C++, Java and C#.

1) C++ with POSIX threads: C++, prior to the C++11 standard, does not contain built-in support for multi-threaded applications. Instead it relies on libraries and the operating system to provide such functionality. On Unix-like operating systems, *POSIX threads*, a.k.a Pthreads, is widely used to provide multithreaded programming support. The Pthreads library provides mutex constructs as means of implementing thread synchronization. In the C++ environment, it is possible for a programmer to *pin* a thread to a specific core, This prevents the scheduler from moving the thread from one core to another, thus avoiding unnecessary overhead. As we observed that pinning threads to cores benefited the throughput of the concurrent data structures, we applied it to all experiments in C++. We pin the threads to fill up one processor before assigning threads to the next one.

C++ provides very basic memory management functionality. Memory allocation/deallocation are done with the help of `new` and `delete`. In concurrent programming, especially lock-free programming, allocating and de-allocating memory is performed by multiple concurrent threads, which might need to be synchronized very often at runtime. Many implementations of lock-free data structures try to avoid that by using their own lock-free memory manager on top of C++ `new`/`delete`. In our context, we want to examine if user level memory management plays a significant role as a synchronization component.

Lock-free Memory Manager: We have implemented a lock-free memory manager (MM) for allocating and de-allocating memory for lock-free implementations in C++. The scheme contains two parts: one main memory allocator shared by all threads and per-thread allocators. The main allocator contains a number of blocks of pre-allocated memory that it gets from the system memory. It provides blocks of memory to the per-thread allocators. Every thread has one per-thread allocator. Whenever a thread wants to allocate memory for the data structure, it gets one from the per-thread allocator. When this allocator runs out of memory, it can request new blocks of memory from the main allocator. When a block of memory is no longer used by the data structure, it will be returned to the memory block where it is allocated from, to be reused later.

This memory manager can provide fast allocation for each thread since allocating new memory usually only involves operation on its local block, which does not require synchronization. Synchronization is only needed when the thread uses up the block assigned to it and needs to allocate a new block from the main allocator.

2) Java: Java offers an extensive API for concurrent programming via its `concurrent` package. In addition to several standard data structures, it also includes most of the low level synchronization primitives needed, such as TAS, CAS or Fetch-And-Add. However, whether these methods actually implement the respective machine instructions or include some implicit locks, depends entirely on the implementation of Java's Virtual Machine for each architecture and operating system [19]. Also, a well specified memory model accompanies the implicit synchronization constructs of the language. Memory management has been left to Java's implicit garbage collector.

3) C#: The native runtime environment for C# is the .NET framework provided by Microsoft which runs exclusively on Windows. To be able to perform our experiments in the same Linux environment used for the other languages, we have also used Mono. This is an open source runtime for the .NET framework which allows programs written in C# to be executed both on Windows and on Linux. The `System.Threading` namespace provides classes – Mutex, Monitor and Interlocked – for synchronizing thread activities. Mutex and Monitor classes represent locking synchronization whereas the Interlocked class comes with atomic primitives which can be used to create various non-blocking synchronization methods.

Languages	C++	C#	Java
Memory management	malloc, customized	implicit memory management	
Synchronization constructs and language features	PMutex, Lock-free MM	Mutex, `lock`	Reentrant, Synchronized
	TAS, TTAS, Lock-free, Array lock		
Contention	Low, High		
Number of Threads	2, 4, 6, 8, 12, 24, 48		
Measurement intervals (sec)	0.2, 0.4, 0.6, 0.8, 1, 2, 3, 4, 5, 10		

Table I: Experimental Setup

V. Experimental Setup

Our purpose is to evaluate the throughput and fairness values of the test cases in all the languages and for different contention levels. For every data structure – as discussed in Section IV – we ran a set of experiments consisting each time of a different number of threads, that were concurrently competing to access the data structure. Every such experiment ran for a fixed amount of time and multiple different time intervals were used. All the different parameters of our experiments along with their values can be seen in Table I. Every experiment was replicated 10 times resulting in a sample satisfying normality with $\alpha = 0.05$ level of significance (Shapiro-Wilk test). The means of these values are presented in our results. Furthermore, limited according to time and resources, samples from cases where the means were different but close were compared with ANOVA tests in order to confirm their difference, with the same level of significance.

In the queue case the operations were an enqueue or a dequeue with equal probability. Each thread was assigned the same probability distribution in all the experiment sets, across the different parameters respectively. In order to calculate the throughput value we used the 10 seconds long tests. There we counted the total number of successful operations for all the threads and divided by the exact duration on each experiment. The shorter time intervals were used for calculating the fairness index according to our definition in Section III. The reason for this variety of shorter intervals, is that fairness results can be deceiving the longer an execution runs. In order to vary the contention level in the queue experiments, dummy work was introduced in every thread between the operations on the data structure.

The operations on the hash table were insert and delete with 10% probability each and search with 80% probability. Again the same probability distributions were assigned per thread in all the experiments. The fairness index this time was furthermore calculated per operation basis. The contention level was varied by changing the number of buckets, 8 for the high contention and 32 for the low.

For the implementations in Java, the IcedTea6 version 1.11.3 of the OpenJDK6 Runtime Environment was used. We ran the C# implementations using version 2.10.5 of Mono. For the C++ case GCC 4.4.1 was used. The host operating system for all of the above was based in version 3.0.0 of the Linux kernel. The C# implementation was also tested in the .NET Framework version 4.0 on Windows 7.

We performed our experiments on an Intel based workstation with 2 sockets of 6-core Xeon E5645 (Nehalem) processors with Hyper Threading (24 logical cores in total). In order to investigate how a different hardware architecture can influence the fairness values of our case studies we also performed the experiments on a second contemporary workstation. That consists of 4 sockets with AMD Opteron 6238 (Bulldozer) 12-core processors (48 logical cores in total). The processors had comparable CPU clock speeds (2.4 and 2.6 GHz respectively) and both the machines had DDR3 at 1366 MHz main memory. The Intel machine is provided with Quick-Path Interconnect for connectivity between chips and I/O subsystem, whereas, the AMD machine had Hyper-Transport for the same [25]. However, the implementation of Simultaneous Multi-Threading [26] on the two architectures differ. In an Intel (Nehalem) processor two threads can share the resources on each physical core [27], making it appear as two logical cores to the operating system. The AMD (Bulldozer) processor follows a modular architecture [28]. Here inside each module, two threads share resources other than their individual integer cores.

VI. Analysis

In order to present, comprehend and describe the observations of the wide extent of experiments that were performed, a summary of the main observations regarding each of the test cases are available in Table II and III. There, they are divided in common observations that stand for all the programming environments tested and then per language basis. In every type of measurement the observations are also grouped according to the most influential parameters (contention regarding throughput, architecture regarding fairness). A third column in every case exists for observations regarding the relation between throughput and fairness.

The discussion in the following subsections also follows a similar structure, namely key comments on common behavior for all the environments appear before comments regarding specific environments.

A. Queue: General Discussion

The fine-grained queues achieve in most of the cases higher throughput than their coarse-grained counterparts. This is expected, as doubling the locks allows up to two threads to operate in parallel, one enqueueing and one

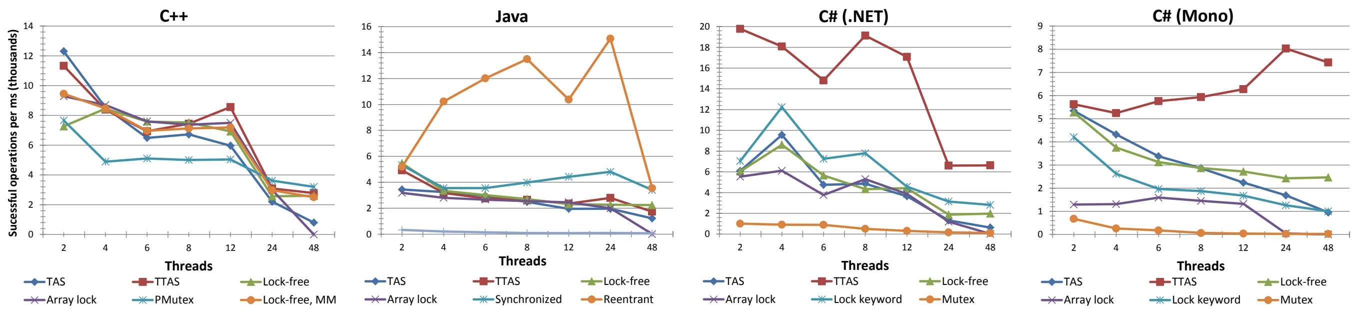

Figure 1: Throughput of the lock-free and fine-grained queues on the Intel system under high contention

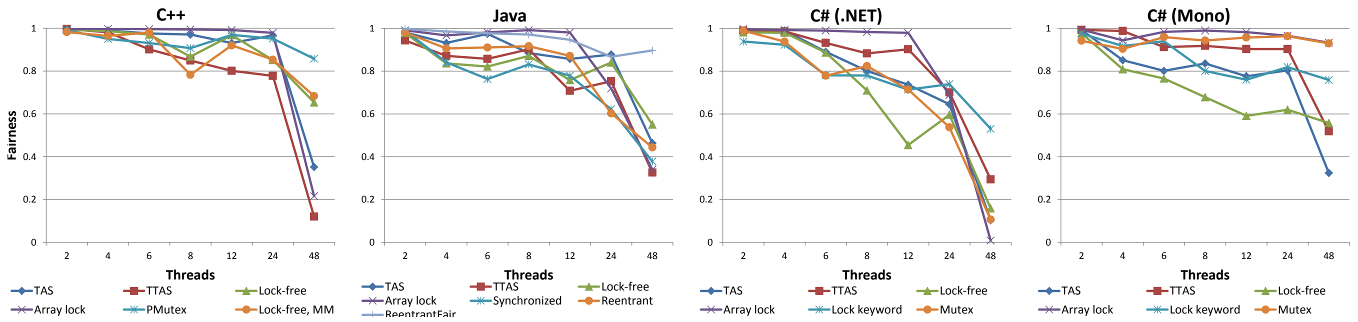

Figure 2: Fairness of the lock-free and fine-grained queues on the Intel system (600 ms time interval)

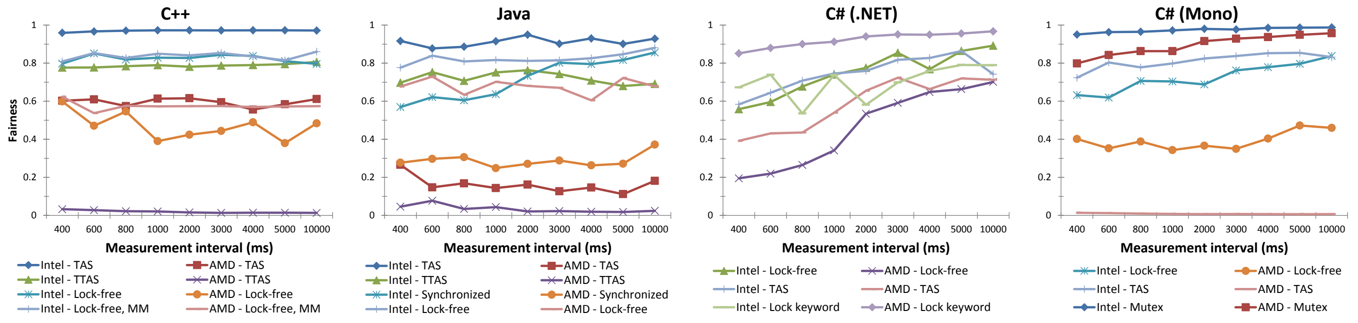

Figure 3: Fine-grained and lock-free queues which show major differences in fairness across platforms at 24 threads

dequeueing. The trends among the different lock types in the coarse-grained queues are similar comparing to the respective in the fine-grained ones. Therefore, unless explicitly mentioned, from now on all references to lock based queues will be based on the ones of the fine-grained kind.

The throughput results of lock-free and fine-grained queues of the case are presented in Figure 1. The constructions based on the array lock consistently achieve the worst throughput value in the case of 48 threads in all the studied programming environments. Since this is more than the number of the system's hardware threads, i.e. the hardware limit, any thread waiting in the array might be swapped out by the scheduler. This forces the remaining threads in the array to wait, until the former is swapped back in. Of course this also affects the fairness index of the method besides the throughput value. Due to the above, the results are in fact so low that we consider this solution inapplicable for this number of threads.

At first, for low numbers of threads and/or low contention, all methods show a high index of fairness. An interesting observation that occurs as the number of threads increases, and particularly in the high contention setting, is the sensitivity of the fairness values along the different time intervals. It is quite reasonable that during a small time interval even the slightest scheduling unfairness would affect the measured value. This is even more visible the more the threads are, since the one with the maximum or minimum number of operations affects less the average fairness.

The fairness experiments are also studied for the AMD system, to gain better understanding of the influence of the hardware architecture. The methods where major differences were observed are presented in Figure 3. We should also

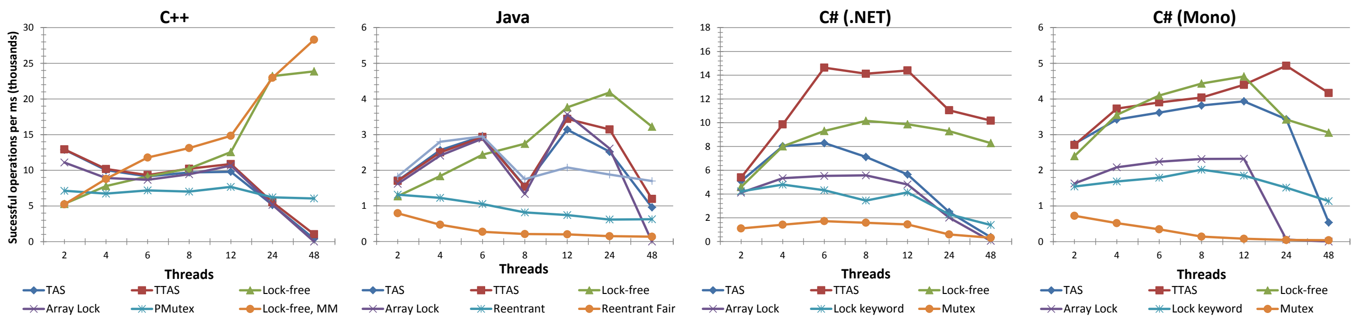

Figure 4: Throughput of all hash tables on the Intel system under high contention

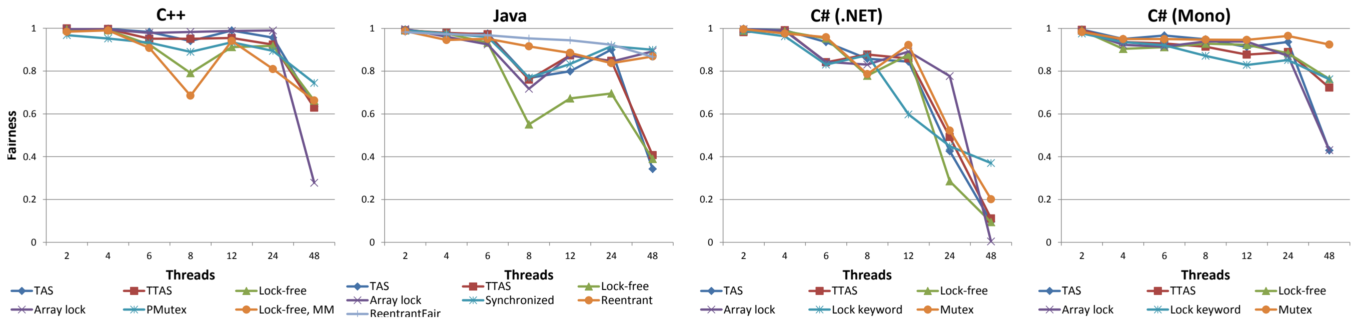

Figure 5: Fairness of all hash tables on the Intel system (600 ms time interval)

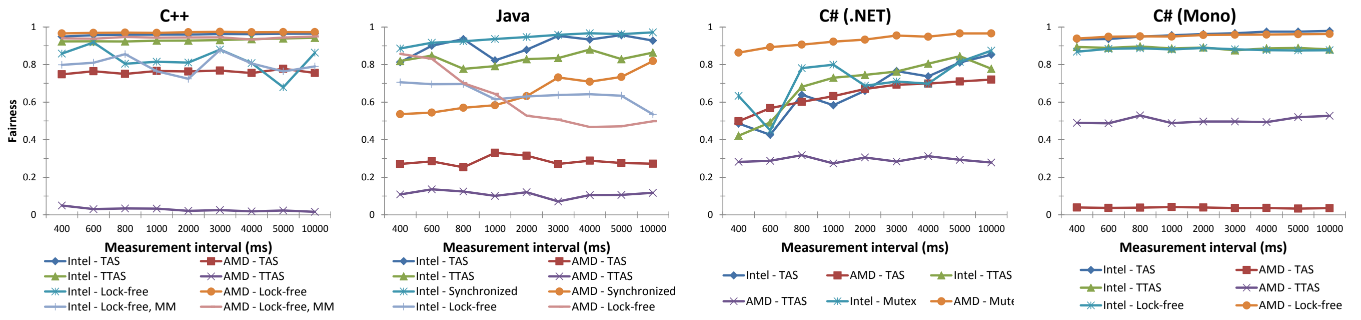

Figure 6: Hash tables which have major differences in fairness across platforms at 24 threads

point out that while the 48 threads exceed the hardware limit on the Intel system, this is not the case on the AMD system, which can support up to 48 hardware threads.

B. Queue: Environment Specific Discussion

As mentioned in Section IV-B1, in C++ the option to pin specific threads to specific processors is used. That explains the drop of throughput values showed in Table II. When the number of competing threads is up to 12, our pinning strategy schedules them in one processor in a socket. When the number exceeds 12, the next 12 threads, i.e. threads number 12 to 24, are scheduled on a second processor which do not share the same L3 cache with the first one. This increases the possibility of cache conflicts among threads, which results in the throughput drop at 24 threads.

Continuing in the C++ case, the TAS based and TTAS based queues are among the queues which achieve the highest throughput in the cases of up to 4 competing threads. This advantage comes from the fact that the lock is constructed from just one atomic operation. However, as the number of threads increases, the two end points of the queue become hot spots. The cost of dealing with high contention, such as cache conflicts, becomes higher, making such simplicity less important to the throughput results. As a result, the difference in throughput between the TAS and TTAS based queues, and the remaining queues, except for the PMutex one, is relatively small when the number of threads is above 4 up to the hardware limit.

The trend of the PMutex based queue's throughput when increasing the number of threads differs from the other implementations. It is lower than the other queues for thread counts between 4 and 12, but keeps almost the same throughput value in the case of 24 and 48 threads.

	Throughput	**Fairness**	**Throughput versus Fairness**
All	- Fine-grained queues perform better than the coarse-grained ones in most of the cases. - The array lock based constructions consistently achieve the worst throughput values in the multiprogramming case of 48 threads.	- Fairness deteriorates as the number of threads increases. - The fine-grained queues are almost always fairer than their coarse-grained counterparts. - When the threads are more than the hardware limit, the results of the array lock are so low that the solution can be considered as inapplicable. - In lower contention scenarios everything is fair until the contention is practically increased by the number of competing threads. The trends there are similar to the high contention cases, but with better absolute values.	- A trade-off must be made between throughput and fairness in most of the synchronization methods. - The lock-free queues in general provide a fair balance between throughput and fairness.
C++	*High Contention* - Steep drop of throughput values when the number of competing threads increases from 12 to 24. - TAS and TTAS based queues are among the queues which achieve the highest throughput in the cases of up to 4 competing threads. TTAS performs better as the number of threads increases. - Pmutex performs lower than the other queues between 4 and 12 threads, but scales better from 24 to 48 threads where it achieves the higest throughput value. *Low Contention* - The lock-free queue with lock-free memory management outperforms the others.	*Intel* - Below the hardware limit, most of the implementations achieve very high fairness values. - Lock-free and PMutex based queues maintain high fairness values at and above the hardware limit. - For most methods the fairness values at 8 threads are lower than those at 6 or 12. *AMD* - Fairness values deteriorate sooner than the Intel case (12 vs 24 threads). In general, the structures and locks on the AMD machine are less fair than on the Intel machine. - The array lock based queue is the most fair, with the PMutex based queue usually performing fairer than the remaining methods.	- Up to 24 threads, the TTAS lock has its throughput among the highest and its fairness among the lowest. TAS based and PMutex have the exact opposite behavior. - The inverse relation does not cover all the methods. The array lock achieves high throughput and fairness up to 24 threads and the PMutex lock gives the highest throughput and fairness at 24 and 48 threads. - The lock-free queues achieve throughput among the highest while maintaining a good, though not top, fairness. Thus they manage to provide a balance between throughput and fairness.
C#	- Throughput is consistently higher with the .NET framework compared to Mono. - The Mutex lock constructs has distinctively lower throughput than the other synchronization methods. *High Contention* - The TTAS locks has significantly higher throughput. *Low Contention* - The lock-free implementation performs better than all other methods. - the TTAS locks display lower throughput than the language provided `lock` keyword.	- The language provided lock constructs have a very high fairness measure overall. - For a low number of threads, all methods show a high degree of fairness. *Intel* - In the 48 thread case the fairness drops drastically. *AMD* - The variation of fairness values along different numbers of threads is higher. - For more than 8 threads, the fairness drops by 50% for the TAS and TTAS locks. The lock-free version shows a similar trend. - For more than 12 threads, the fairness of the TAS lock drops close to zero.	- The TTAS locks provide high throughput, but low fairness. - Array locks and the language-provided lock constructs are very fair, but with low throughput. - The lock-free implementation provides a trade-off between throughput and fairness.
Java	*High Contention* - The constructions based on the simple Reentrant locks outperform all the rest in most of the cases. - The Synchronized based locks followed by the lock-free implementation have a scalable behavior and a relatively good throughput. *Low Contention* - The lock-free and the fine-grained queues based on TAS, TTAS and Synchronized blocks present the highest throughput. - The fair version of the Reentrant lock is the slowest except for the array lock that severely drops in 48 threads.	*Intel* - The absolute winners in most cases are the fair Reentrant lock and the array lock. - Also the TAS based queues follow closely, especially in the case of 24 threads, while in 48 the differences are widened. - The lock-free queue is the next to come with a similar behavior except for the 48 thread cases where it is slightly better than the TAS based queues. *AMD* - Worse fairness values for lower thread cases. - The Synchronized block, TAS and TTAS based locks are always worse than the Intel case (see also Figure 3). - For up to 12 threads the lock-free queue is fairer. After that it achieves lower values but it is still the third in order after the fair Reentrant and the array lock constructs.	- If fairness is a critical objective, then locking methods which inherently have a queue waiting structure (fair Reentrant or array lock), are definitely the choice, sacrificing throughput though. - The unfair Reentrant lock and the synchronized block give absolute throughput but fairness is not guaranteed at all. On the same side of the balance is TTAS. - The lock-free queue manages to balance this tradeoff with relatively good results in both sides.

Table II: A summary of the main observations regarding the *queue* case study

The internal design of PMutex based is different from the other locking methods. In contended cases, a thread goes to sleep if it fails to acquire the lock. We can observe that this mechanism, which is a form of backoff, penalizes the throughput in the cases of lower number of threads, i.e. below the hardware limit. However it helps the PMutex based queue deal with extreme contention cases, i.e. 24 and 48 threads, better than other implementations. The results show that both throughput and fairness benefit by this.

The thread pinning in specific processors also affects fairness. We observe that the fairness values at 8 threads are lower than those at 6 or 12 for most implementations. The reason is that in the case of 6 or 12 threads, all cores are scheduled to run either one or two threads, respectively. While in the case of 8 threads, some cores run one and some run two threads, which causes more fairness differences among the threads.

In Java, the throughput of the Reentrant lock and its difference from the rest is the most noticeable. This happens due to the Reentrant lock's inherent backoff mechanism – described in Section II – similar of which are not inherent in the other locks (e.g. exponential backoff). However, the overhead of the Reentrant lock's mechanism does not pay off in lower contention conditions as both versions of the lock are the lowest, with the fair one being by far the worst.

The C# implementations were tested in both Mono and the .NET Framework. The throughput results were consistently in favour of the latter. Furthermore, the low throughput of the Mutex based constructions is justified by its design, which is heavyweight due to the requirement that it should also provide interprocess synchronization. However this low throughput for Mutex, as well as for the `lock` construct, come in benefit of fairness.

C. Hash table: General Discussion

The throughput of all hash table implementations in different programming languages is presented in Figure 4. A summary of the main observations for throughput, fairness and their relation can be found in Table III.

The hash table is a data structure with many points where operations can be performed independently – the different buckets. Thus it allows more threads to be served concurrently and, since the keys that were used were uniformly distributed, it also allows for fairer executions. In fact we observe interesting variations of the fairness values between the different synchronization mechanisms in the cases where the number of competing threads is bigger than the number of the available buckets. Still though, concerningly low fairness values occur when the number of threads exceeds the hardware limit.

Due to the different nature of the hash table's methods, we first checked the values of the fairness index per operation, i.e. Insert, Remove, Search and also for the total number of operations regardless their kind. Since the patterns are similar, unless explicitly mentioned, the observations stand for any kind of operation.

As it can be seen in Table III, different synchronization mechanisms than in the queue case have to pay the tradeoff between throughput and fairness.

In fact the pattern that can be observed is that all the synchronization methods that achieved high throughput in the low contention cases of the queue are the ones that manage the best throughput performance in the hash table. This is because the hash table consists of multiple linked lists where the hashed values are stored, i.e. the same basic component as the queue. And since the contention and the requested operations of the competing threads is now uniformly distributed along the different linked lists, the contention is lowered in each of them. Therefore the best performing solutions locally form the final result for the hash table. Similarly we can see the local fairness behaviour of the queues magnified in the total fairness index of the hash table.

D. Hash table: Environment Specific Discussion

Again in the case of C++ we can see the advantages and disadvantages of specific thread pinning to cores. While generally when the number of competing threads is less than the hardware limit, i.e. 24 threads, all the hash tables behave very fair, this observation can not be applied for the case of 8 threads. The reason is that scheduling 8 threads into 6 cores with hyperthreading causes unfairness when some cores run only one thread and the other running two. In the case of 6 or 12 threads, they are scheduled evenly to cores.

We also observe that, as the TAS- and TTAS based hash tables achieve very low throughput, even a small unfairness in the scheduling of threads can cause a negative effect on their fairness measures, especially at short time intervals. It is interesting though that the values can recover in longer time intervals.

The tradeoff between throughput and fairness appears when the number of threads is over a threshold, at which point we start to get contention at the sharing points in the data structure, i.e. the behaviour associated with the queue. These thresholds are usually at 8 and 24 threads in high and low contention scenarios, respectively. This result agrees with the fact that the hash tables have 8 or 32 buckets in each respective scenario. When the number of threads goes beyond the threshold, we see that some implementations, which achieve high throughput, might have to sacrifice the fairness. TAS- and TTAS based (and array lock based, to some extent) hash table represent this trend with high fairness, but low throughput. Lock-free hash tables also show a clear trend, but with high throughput and lower fairness results. Between the lock-free implementations with and without lock-free memory management, the former achieves higher throughput, but also gets lower fairness result than the latter, and vice versa. PMutex, the language

	Throughput	**Fairness**	**Throughput versus Fairness**
All	- The lock-free implementations perform better than most of the lock based implementations, on average, and show scalability as well.	- The fairness indices are generally quite high. The differences become more visible when the number of threads is greater than or equal to the number of buckets.	- Different synchronization methods excel in throughput or fairness than in the queue case.
C++	*High Contention* - TTAS, TAS and array lock based hash tables have similar throughput values with the first one usually performing slightly better. However they do not scale beyond 12 threads where their values drop significantly. - The PMutex based hash table achieves higher throughput than the previous group beyond 12 threads, but not before. - The lock-free implementations scale all the way up to 48 threads, achieving the highest throughput values in 8 or more threads. The one with the lock-free memory manager performers better than the simple one in most of the cases. *Low Contention* - All the implementations show higher values and better scalability than in the high contention case.	*Intel* - Below the hardware limit (24 threads) all the cases behave fair except for the 8 threads case. - At and above the hardware limit the values of lock-free implementations drop lower, though still about 0.75. - TAS and TTAS lock are very unfair in the 48 threads case. However they recover and achieve high values for longer time intervals. The PMutex construction maintains high values consistently throughout all the time intervals. *AMD* - TAS and TTAS locks are heavily influenced by the change of the architecture.	- When the number of threads is larger than the number of buckets, the lock free implementations achieve high throughput but moderate or low fairness. - TAS, TTAS and to some extent array lock based hash tables show the opposite trend with high fairness and lower throughput values. - The throughput of the PMutex based hash table is usually the lowest and its fairness, though decent, is not among the top. Nevertheless, it keeps steady performance at higher numbers of threads in terms of both throughput and fairness.
C#	- The .NET implementations on Windows perform significantly better (2x - 2.5x) than the Mono implementations on Linux. - The TTAS locks perform better than other locking methods. The lock-free implementation is the one that follows. - The Mutex based locking constructs gives the lowest throughput. - The methods scale up to 8 threads, the number of buckets, and after 12 threads the throughput starts decreasing. The exception is the Mutex lock on Mono which does not scale at all. - The relative order, with respect to absolute throughput, remains largely unchanged by the change in runtime system. *Low Contention* - Increasing the number of buckets causes an increase in throughput across the board.	- For up to 6 threads, all methods are highly fair, regardless of architecture and environment. *Intel* - No single algorithm is always the most fair one for thread counts ranging from 12 to 48. *AMD* - The TTAS locks drop in fairness after 8 threads. - For more than 12 threads, the Mutex lock is the most fair.	- The TTAS lock shows high throughput, but poor fairness. - The Mutex lock is very fair, but lacks in throughput. - The lock-free and the `lock` based hash table provide a good tradeoff between throughput and fairness.
Java	*Low Contention* - The highest throughput is usually achieved by the hash tables based on Synchronized blocks, array locks and the lock-free ones. However in 48 threads specifically the lock-free construction keeps increasing its performance while the array lock severely drops. - Both the Reentrant locks consistently show low values. *High Contention* - The behavior is similar to the low contention case except for the lock-free hash tables which performs 20-30% lower. Despite that, when the number of threads increases it achieves the highest performance.	*Intel* - The highest fairness values are achieved by both the Reentrant locks based hash tables. Closely follows the one built on the Synchronized blocks. - TAS and TTAS and the array lock are relatively fair in most of the cases. In 48 threads they are the least fair, with TAS and TTAS improving though their values in longer time intervals. - The lock-free hash table is the least fair. *AMD* - The fairness indices of TAS and TTAS based hash tables are heavily influenced. The change also hinders, in a smaller scale, the Synchronized block construction and slightly the lock-free one.	- The lock-free method sacrifices fairness for higher throughput. - The Reentrant locks provide high fairness values without managing decent throughput. - The implementations based on TAS, array lock and the Synchronized block manage to balance the tradeoff in a very efficient manner.

Table III: A summary of the main observations regarding the *hash table* case study

specific construct in C++ that we tested, surprisingly does not perform well in this case study in the cases of less than 24 threads.

The different runtime systems for C# do not cause any change in the relative order of the methods as of throughput performance, but still the values in .NET are consistently higher than the ones in Mono.

Regarding fairness, no single algorithm is always the most fair for the higher numbers of threads on the Intel machine. On the contrary, considerable differences occur when changing to the AMD architecture, leaving the Mutex construct as the most fair one.

Solutions with high overhead like the Reentrant locks do not pay off for the hash table in Java either. The throughput is the lowest, however their inherent queue structure benefits fairness. More lightweight solutions manage to balance this tradeoff.

VII. Conclusions

In this paper we evaluated different types of lock-based (from fine-grained to coarse-grained), as well as lock-free, synchronization methods with regard to their potential for high throughput and fairness.

Selecting the best synchronization constructs to achieve the desired behavior is a non-trivial task, which requires thorough consideration of different aspects. Besides languages and their features, the selection is also governed by several other parameters and factors, and the interplay among them, e.g. the system architecture of the implementation platforms; possible run-time environments, virtual machine options and memory management support; and the characteristics of the applications.

Our results show that the implicit synchronization constructs provided at the language level, for the managed languages used in our experiments, provide decent throughput and fairness for many scenarios. Much can however be gained by using more complex designs and implementations in C++, that does not rely on automatic garbage collection. This is especially true for data structures with a fine-grained design, where operations are not just simply serialized, but can actually take place concurrently. In general, it is clear that the more fine-grained the designs is, the higher the potential to achieve a higher degree of throughput, because of their high potential for parallelism. A fine-grained design also leads to increased fairness between the actors involved, as multiple operations can be performed in parallel without conflicts.

We observed that most synchronization methods show reasonable fairness and throughput when used by a low number of threads, or for scenarios with very little contention. However, when the contention increases, and the number of threads that are executed concurrently passes the number that can be scheduled on a single socket, the behaviour starts to deviate. This can be mitigated by having a data structure design that supports more parallelism, allowing for a wider choice of concurrency mechanisms. Some lock constructs, that performed poorly in queues under high contention, worked fine when used in hash tables under high contention. The cause of this is the inherent distribution of data accesses in a hash table. Methods that use backoff were shown to work very well during high contention scenarios, but the extra overhead lowered the throughput during lower contention. Some constructs such as array locks are very fair, but drops quickly in throughput when faced with increased contention. In most cases, a trade-off between throughput and fairness has to be made, no matter the language or architecture. A reasonable such trade-off for many scenarios could be made using lock-free algorithms, which in most cases manages to pair good fairness with high throughput.

More knowledge about the specific execution environment could lead to more fine-tuned decisions on which synchronization mechanism to select. Our experimental observations shed some light in this direction.

The results in this paper allows us to take a step towards improving methodologies for choosing the programming environment and synchronization methods in connection to the application and the system characteristics.

Acknowledgment

This work was partially supported by the EU as part of FP7 Project PEPPHER (www.peppher.eu) under grant 248481, the Swedish Foundation for Strategic Research as part of the project RIT-10-0033 ”Software Abstractions for Heterogeneous Multi-core Computer”, and by the Swedish Research Council (Vetenskapsrådet) project ”Fine grain Synchronization and Memory Consistency in Parallel Programming” Contract nr. 621-2010-4801.

References

[1] B. N. Bershad, “Practical Considerations for Non-Blocking Concurrent Objects,” in *Proceedings of the 13th International Conference on Distributed Computing Systems*, 1993, pp. 264–274.

[2] K. Fraser and T. L. Harris, “Concurrent programming without locks,” *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 2, 2007.

[3] C. A. R. Hoare, “Towards a theory of parallel programming,” in *The origin of concurrent programming*, P. B. Hansen, Ed. New York, NY, USA: Springer-Verlag New York, Inc., 2002, pp. 231–244.

[4] J. M. Mellor-Crummey and M. L. Scott, “Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors,” *ACM Transactions on Computer Systems*, vol. 9, pp. 21–65, 1991.

[5] H. Sundell and P. Tsigas, “NOBLE: A Non-Blocking Inter-Process Communication Library,” in *Proceedings of the 6th Workshop on Languages, Compilers and Run-time Systems for Scalable Computers*, ser. Lecture Notes in Computer Science. Springer Verlag, 2002.

[6] S. Benkner, S. Pllana, J. Träff, P. Tsigas, U. Dolinsky, C. Augonnet, B. Bachmayer, C. Kessler, D. Moloney, and V. Osipov, "PEPPHER: Efficient and Productive Usage of Hybrid Computing Systems," *IEEE Micro*, vol. 31, no. 5, pp. 28–41, sept.-oct. 2011.

[7] H. Inoue and T. Nakatani, "Performance of multi-process and multi-thread processing on multi-core SMT processors," in *2010 IEEE International Symposium on Workload Characterization (IISWC)*, dec. 2010, pp. 1–10.

[8] M. Awasthi, D. W. Nellans, K. Sudan, R. Balasubramonian, and A. Davis, "Handling the problems and opportunities posed by multiple on-chip memory controllers," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques (PACT)*. New York, NY, USA: ACM, 2010, pp. 319–330.

[9] V. Nazaruk and P. Rusakov, "Blocking and non-blocking process synchronization: Analysis of implementation," *Scientific Journal of Riga Technical University, Computer Science. Applied Computer Systems*, vol. 44, pp. 145–150, 2011.

[10] A. Lamarca, "A performance evaluation of lock-free synchronization protocols," in *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM Press, 1994, pp. 130–140.

[11] M. M. Michael and M. L. Scott, "Relative Performance of Preemption-Safe Locking and Non-Blocking Synchronization on Multiprogrammed Shared Memory Multiprocessors," in *Proceedings of the 11th International Parallel Processing Symposium (IPPS)*, 1997.

[12] P. Tsigas and Y. Zhang, "Integrating Non-Blocking Synchronisation in Parallel Applications: Performance Advantages and Methodologies," in *Proceedings of the 3rd international workshop on Software and performance*. New York, NY, USA: ACM, 2002, pp. 55–67.

[13] ——, "Evaluating the Performance of Non-Blocking Synchronization on Shared-Memory Multiprocessors," *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, no. 1, pp. 320–321, 2001.

[14] J. Chen and W. W. III, "Multi-Threading Performance on Commodity Multi-core Processors," in *Proceedings of 9th International Conference on High Performance Computing in Asia Pacific Region (HPC Asia)*, 2007.

[15] T. Anderson, "The performance of spin lock alternatives for shared-money multiprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 1, pp. 6–16, jan 1990.

[16] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. Morgan Kaufmann, 2008.

[17] P. Magnusson, A. Landin, and E. Hagersten, "Queue locks on cache coherent multiprocessors," in *Proceedings of the Eighth International Parallel Processing Symposium*, apr 1994, pp. 165–171.

[18] T. Craig, "Building FIFO and Priority-Queuing Spin Locks from Atomic Swap," University of Washington, Technical Report 93-02-02, Tech. Rep., 1993.

[19] Oracle. Java standard edition documentation. [Online]. Available: http://docs.oracle.com/javase/7/docs/technotes/guides/concurrency/index.html

[20] K. M. Franke Hu., Russell R., "Futexes and furwocks: Fast userlevel locking in Linux," in *Proceedings of the 2002 Ottawa Linux Summit*, 2002.

[21] P. H. Ha, M. Papatriantafilou, and P. Tsigas, "Efficient self-tuning spin-locks using competitive analysis," *Journal of Systems and Software*, vol. 80, no. 7, pp. 1077–1090, Jul. 2007.

[22] R. Jain, D.-M. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *CoRR*, vol. cs.NI/9809099, 1998.

[23] M. Michael and M. 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, pp. 267–275.

[24] M. Michael, "High performance dynamic lock-free hash tables and list-based sets," in *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*. ACM, 2002, pp. 73–82.

[25] D. Hackenberg, D. Molka, and W. E. Nagel, "Comparing cache architectures and coherency protocols on x86-64 multicore SMP systems," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 413–422.

[26] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *ISCA*, 1995, pp. 392–403.

[27] M. E. Thomadakis, "The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms," A research report of Texas A&M University, Tech. Rep., 2011.

[28] M. Butler, L. Barnes, D. Sarma, and B. Gelinas, "Bulldozer: An Approach to Multithreaded Compute Performance," *IEEE Micro*, vol. 31, no. 2, pp. 6–15, march-april 2011.