Concurrent Transformation Components using Contention Context Sensors

Erik Österlund
Software Technology Group
Linnaeus University
Växjö, Sweden
erik.osterlund@lnu.se

Welf Löwe
Software Technology Group
Linnaeus University
Växjö, Sweden
welf.lowe@lnu.se

ABSTRACT

Sometimes components are conservatively implemented as thread-safe, while during the actual execution they are only accessed from one thread. In these scenarios, overly conservative assumptions lead to suboptimal performance.

The contribution of this paper is a component architecture that combines the benefits of different synchronization mechanisms to implement thread-safe concurrent components. Based on the thread contention monitored at runtime, context-aware composition and optimization select the appropriate mechanism. On changing contention, it revises this decision automatically and transforms the components accordingly. We implemented this architecture for concurrent queues, sets, and ordered sets. In all three cases, experimental evaluation shows close to optimal performance regardless of the actual contention.

As a consequence, programmers can focus on the semantics of their systems and, e.g., conservatively use thread-safe components to assure consistency of their data, while deferring implementation and optimization decisions to contention-context-aware composition at runtime.

Categories and Subject Descriptors

D.1.3 [**Software**]: Programming Techniques—*Concurrent Programming*; E.1 [**Data**]: Data Structures

Keywords

context aware composition, concurrent components

1. INTRODUCTION

Parallel programs and concurrent components that can run safely in concurrent, i.e. contended contexts are increasingly important with the rise of symmetric multiprocessing (SMP) architectures. A common problem programmers face is that the level of runtime contention of a component is unknown at development time. If multiple layers of application programming interfaces (APIs) depend on one another, it becomes increasingly difficult to know which level of contention a component will be exposed to. Often, a conservative approach is taken. A thread-safe component is picked if there is a possibility that an API is used in a parallel context, e.g., by a multi-threaded application. However, if the contention was known, alternative component variants may be preferable.

The dream of not having to manually pick component variants based on assumed contention traces back to old classes like Vector in the Java class library. All methods were synchronized so that programmers could assume thread-safety always. Unfortunately, the approach led to bad sequential performance, and unscalable parallel performance due to the use of locks. Therefore, the dream was abandoned in later generations of the class library, and the responsibility of picking the appropriate component variant became a burden for programmers again. The burden got worse with the advent of lock-free components. Now in addition to knowing whether thread-safety is needed or not, the component variant picked also depends on assumed level of contention.

Universal constructions like, e.g., transactional memory (TM) [13] based on either software (STM) [12, 8, 27] or hardware (HTM) [2, 11] provide good scalable performance and ease of use. Lock-free components [23, 18, 14, 9] provide even better concurrent performance, while coarse grained biased locking [26] is the fastest in absence of concurrency.

There is no single best synchronization mechanism. Instead of trying to find the ultimate synchronization mechanism combining the pros of all existing approaches, this paper seeks to find a way of implementing and composing components using different synchronization mechanisms and pick the appropriate one automatically.

This paper proposes a novel methodology inspired by context-aware composition [25, 16] and adaptive spin locks [26]. Its goal is simple: provide the optimal uncontended performance and the optimal concurrent performance at the same time. The idea is to start with optimistic biased locks that are essentially for free in terms of performance. At signs of actual contention, adapt and break the lock by either switching to a more fine-grained locking scheme that scales better or transform into a completely lock-free solution for maximum scalability.

This methodology extends context-aware composition to use components in a concurrent context, where before only the components could split work to be done in parallel.

Adaptive spin locks are great at sensing different contention and adapting the time to spin before yielding to the OS. This paper goes even further and provides an API sig-

ASE '14 September 15 - 19 2014, Västerås, Sweden
ACM 2014 978-1-4503-3013-8/14/09 ...$15.00.
http://dx.doi.org/10.1145/2642937.2642995.

nalling into the application that there is a lock bottleneck. This allows arbitrary user defined action to deal with the problem. This paper contributes with: 1) contention-aware composition, a methodology for dealing with overhead and scalability problems of concurrent components under statically unpredictable contention. 2) contention-aware composition applied to Java concurrency data structures. 3. evaluations running on our own modified OpenJDK showing that these concurrent components perform (almost) as well as the best known component for each contention context.

The paper is organized as follows: Section 2 introduces three standard synchronization mechanisms: locks, lock-free algorithms, and TM. Section 3 introduces transforming components using context-aware composition at runtime based on contention as a context property. Section 4 shows how contention can be monitored efficiently while Section 5 shows that an implementation of the actual component transformation can be implemented efficiently. Section 6 introduces the Java concurrency data structures used in evaluation, highlights some implementation details and finally describes the evaluation and the evaluation results. Finally, Section 7 discusses the related work and Section 8 concludes the paper.

2. SYNCHRONIZATION MECHANISMS

We refer to *concurrent components* as data structures implementing an API whose instances are consistent in a concurrent executiton context. In object-oriented languages, concurrent components are instances of classes that are consistent even if accessed by several threads concurrently.

We describe three different synchronization mechanisms used to implement concurrent components: 1) locks, 2) lock-free synchronization using atomic instructions like compare-and-swap (CAS), 3) TM. We argue they all have their weaknesses and strengths, specifically, for our main concern in this paper: *performance.* We discuss the theoretical pros and cons of each synchronization mechanism in both (almost) sequential and (highly) concurrent execution contexts.

2.1 Locks

A *lock* can be acquired by one and only one (owner) thread that may continue with its execution. It blocks other threads trying to acquire the lock until the owner releases it again. Locks are arguably the most common synchronization mechanism because of their availability and ease of use. They are also flexible in the sense that the locking granularity can be varied to achieve scalability. Problems such as deadlocks will not be discussed here as we are only interested in performance and not in programming and verification efficiency.

Sequential Context: In an (almost) sequential execution context, coarse grained locks and sequential algorithms implementing an API tend to always win. If it can be statically proven that objects never escape a thread (by manual or automated escape analysis), locks can even be elided completely and the sequential code can run at no additional overhead. Even when it is not provable statically that a lock belongs to a single thread exclusively, the lock may optimistically assume so using biased locking [26]. Therefore, it installs the presumed owner thread the first time it is locked using CAS. If the owner stays the same, locking and unlocking is performed simply by writing 1 and 0 correspondingly, and is very fast.

Concurrent Context: In concurrent execution contexts, locks suffer from problems predicted by Amdahl's law [1]. Since locks fundamentally only allow one thread to execute at a time, locked programs do not scale very well when the sequential parts dominate the concurrent parts.

There are ways to mitigate this bottleneck by using a different locking scheme. One solution is to use fine-grained locking where only some objects of a complex component local to a change are locked instead of locking the whole component instance. For instance, only some nodes in a tree are locked instead of locking the whole tree. Another solution is to use readers-writer locks that allow multiple readers but only one writer at a time.

Conclusion: Locking is superior for sequential execution but may not be the best candidate for concurrent use.

2.2 Lock-Free Synchronization

Lock-free algorithms [23, 18, 14, 9] were introduced to deal with the problems of locks. Synchronizing instructions are used only on memory locations where there are actual data conflicts. They read and remember values from the memory locations, perform calculations and write the result back to these locations. Using operations like CAS, they detect conflicts, i.e., if another thread has changed the value in between. Then they try again (in a biased loop with branch prediction optimized for the success, and not the try again branch). This allows concurrent modification of components as long as there are no data conflicts and, thus, somehow mitigates the problems of Amdahl's law.

Due to the inherent complexity of these algorithms, not all sequential data structures have lock-free implementations yet. Also, a truly lock-free algorithm requires the underlying execution environment, including the memory manager to be lock-free to retain its progress guarantee, which is difficult to assure. It requires, e.g., lock-free garbage collection [29] or hazard pointers [22]. However, this paper is not concerned with guaranteeing true lock-freedom or real-time properties. Our interest is only the scalable performance characteristics of this class of algorithms for concurrent components.

Sequential Context: The performance of such lock-free algorithms greatly depend on the hardware and how fast its CAS instruction is compared to normal memory accesses.

Historically, x86 implemented `LOCK CMPXCHG` (CAS) by locking the whole memory bus globally during the execution of the instruction. Newer implementations, however, lock only the cache line where the CAS is executed.

A strong CAS requires acquire release semantics and hence needs to issue a full memory fence. In certain architectures this might be a relatively expensive operation, requiring all write buffers and caches to be serialized.

In the case of x86 (which is considered strongly consistent), all aligned memory accesses already have acquire release semantics, and the `LOCK CMPXCHG` instruction already implicitly issues a complete memory fence for free.

In other architectures such as POWER (which is weakly consistent) the cost is higher. Here, enforcing the acquire release semantics requires: (1) an expensive heavy weight `sync` memory fence instruction, (2) a `lwarx` (load link) and `stwcx` (store conditional) in a loop until the instruction can return a valid result free from spurious failures, (3) a lightweight `isync` fence to serialize all writes.

Lock-free algorithms can also rely on volatile memory accesses with acquire release semantics whose performance may similarly depend on which architecture it runs on.

Additionally, even if the volatile memory accesses (including CAS) are as fast as non-volatile memory accesses, lock-free algorithms are still typically slower because they are fundamentally designed to handle data conflicts, e.g., branches for loops retrying committing their changes, which is never necessary in a sequential execution. Moreover, simple component attributes such as the size of a collection are typically computed on demand in a lock-free implementation in order to reduce cache clashes. In the example of the size attribute, this leads to an $O(n)$ operation instead of an $O(1)$ operation of a corresponding sequential data structure that simply maintains a size counter because it does not need to optimize for concurrency.

Concurrent Context: is where the merits of lock-free programming becomes visible. Lock-free components commit changes at linearization points [15] using CAS and optionally lazily (asynchronously on any subsequent operations) update references not necessary for consistency, but for improving time complexity. Therefore, only true data conflicts violating consistency of the component require operations to be restarted. Other data conflicts not necessary for linearization can be tolerated and be lazily handled. This makes lock-free algorithms very scalable in terms of performance and typically the best option if available.

Conclusion: Even though CAS is implemented more or less efficiently by different hardware vendors, the performance of lock-free components is still inherently slower than the performance of its sequential counterparts in the absence of concurrency. However, when there is contention, this class of algorithm provides the most fine-grained synchronization fine-tuned by clever implementers.

2.3 Transactional Memory

Analogous to database transactions, *Transactional Memory* guarantees sequences of load and store instructions to execute in an atomic way. It is a universal construction for turning any sequential algorithm into a lock-free [28] and even wait-free [24] algorithm without any deadlocks. Despite this theoretical benefit, TM failed to become adopted in practice mainly due to performance reasons discussed below. Additionally and not discussed here in detail, speculative TM (performing in-place writes) cannot always guarantee that a transaction will ever end in case of contention since a speculative load could cause an infinite loop never reaching the commit operation.

Sequential Context: STM typically performs significantly worse in a sequential environment compared to a normal sequential solution.

Performance depends on whether the algorithm uses write buffering versus undo logging, pessimistic versus optimistic concurrency, cache line based versus object based versus word based conflict detection etc. An evaluation of the trade offs was done by Saha et al. [27].

HTM can accelerate for instance write buffering in hardware and allows to elide locks. It can reduce the cost compared to STM, but relies on hardware which may or may not exist. Biased locking already shows better performance characteristics without special hardware support in this case.

Concurrent Context: The idea of a universal construction that transforms sequential code into concurrent code automatically without the need to re-engineer or re-design fails because sequential code typically has sequential data dependencies everywhere (since it was not optimized for concurrent use) that, in turn, causes rollback storms.

STM is considered to have higher constant cost but better scalability than coarse grained locking. In previous publications, the break even point seems to be approximately four concurrent threads [4].

HTM has the same scalability properties as STM but provides better constants as long as the hardware write buffers are large enough for the operations, lowering the break even point to two concurrent threads [4].

However, carefully implemented lock-free algorithms typically outperform TM. The problem is that TM aborts transactions for all data conflicts. A lock-free algorithm, conversely, may know that a speculatively loaded value may become invalid or a lazily updated reference may not be written, but it does not matter for the consistency of the algorithm. Therefore, the lock-free algorithm may continue whereas the TM based code has to restart the transaction.

Conclusion: TM does exhibit good concurrent scalability when there is no known lock-free component variant. TM provides decent sequential performance if hardware support is available. However, when there is no hardware support or a lock-free algorithm exists, other options are better.

3. TRANSFORMATION COMPONENTS

Research in finding the ultimate synchronization mechanism combining the best of them all has not found a conclusion yet. We must admit that the best synchronization mechanism depends on the context including both the application context and runtime context. The application context includes read to write ratio, the number of threads, and the actual contention. The runtime context includes properties of hardware, operating system, and language runtime environment such as the memory consistency of the architecture, the number of cores (and, hence, the need for scalability) whether there is a single chip system or multiple chip system, the number of hyper threads for cheaper context switching, the fairness of the scheduler, the fairness of the potential locking protocol (unfairness yields higher performance since TLB caches are already populated when reacquiring the lock), the availability of HTM, the size of its write buffers, the speed of CAS operations, the size of caches, how caches are shared, what cache coherence protocol they employ, the speed of the memory bus etc. Some of the properties are known at compilation time, some at deployment time, and yet some vary dynamically at runtime.

Since there is no single silver bullet that provides the best possible performance in all contexts, our approach is to bridge the gap between different solutions of different synchronization mechanisms by adapting to the actual context dynamically. We present contention sensors and transformation components as novel solutions to detecting the problems and dealing with the problems respectively.

This work builds on the previous work by Andersson et al. [3] introducing context-aware composition and by Österlund et al. [25] introducing dynamically transforming data structures and a framework for reasoning about them.

A *transformation component* consists of an *abstract data representation* and a set of *abstract operations o* operating on this data. The abstract data representation allows for different *data representation variants*, specialized for certain contexts. Each abstract operation o of a component also allows for different *algorithm variants*. In general, the same operation could come in different algorithm variants using the same data representation, each optimized for different contexts. In our approach, each representation variant matches one-to-one an algorithm variant for each o.

Transformation components follow a general design pattern for data structures with changable representation and algorithm variants as proposed in [20]. It uses a combination of the well-known *bridge* and *strategy* design patterns [10]. The special case of our approach is depicted in Figure 1. The component has a reference to the current representation variant. It could be any representation variant that is an instance of the abstract representation.

All state information of a component is contained in data representation variants. There must exist a state transformation from a data representation variant a to b to allow a component transformation. Transformation could use a clone() method accepting the current (unknown, abstract) representation as a parameter or any other read/write iteration of the contents that is complete and homomorphic w.r.t. the operations o.

The changeTo() operation invalidates the previous representation variant so that accesses to it will be trapped, creates a new representation variant of a new type and populates it using the clone() method.

For each operation the component also holds references to the current algorithm variant implementing an operation corresponding to the current representation variant. Algorithm variants are classes specializing an abstract operation class. The abstract class provides an execute method implemented by all subclasses. Calls to an operation are delegated to the current algorithm variant.

Since in the current paper, representation variants and algorithm variants are always connected, we additionally introduce *component variants* comprising both a representation variant and its associated set of algorithm variants.

Based on the changing contention context, i.e., the concurrency a component is exposed to, we strive to select the best performing component variant including the best synchronization mechanism and the algorithm variant with the lowest time complexity. Contention is the only dynamically changing context attribute that we consider here. The rationale is that whenever there is contention, the system experiences a massive slow down that typically outweighs other attributes. A more complex biasing would employ more context attributes like size of data, application profile, hardware, and scheduling. Since, the effects of these context attributes have been researched before [3, 16, 25], we focus on the contention context here.

We propose a single *contention manager* object for each transformation component object. It receives input from the specific active implementations in completely different ways. The contention manager is invariant of how it gets its information. It listens to contention level changes. If the current component variant is considered inappropriate for the current level of contention, it is invalidated and a new equivalent and hopefully more appropriate component variant is instantiated.

Therefore, we need to address three issues: (i) we need to assess the contention context, (ii) we need to be able to transform representation variant and algorithm variants safely, (iii), we need to learn, which changes in the contention level ought to trigger transformation. At least (i) and (ii) need to be performed online and contribute to the runtime overhead of transforming components. Hence, efficiency of context assessment and transformation is an issue. The issues (i)–(iii) will be addressed in the subsequent three sections.

4. CONTINUOUS CONTEXT FEEDBACK

We address issue (i) by introducing continuous context feedback for contention and call it a *contention sensor*. It is implemented differently for the three different synchronization mechanisms and provides a general interface, in essence hooks to user defined code in the contention manager, dealing with adaptation when contention levels change.

For each contention sensor implementation, it is beneficial to log only the *bad* uses of a certain synchronization mechanism. For instance, a lock would only notify the contention manager of bad uses of locks such as when blocking is necessary, rather than that it successfully acquired a biased lock local to one thread. Such a message would never lead to any change, and hence the message would only be an unnecessary overhead. The same principle applies to the contention sensors for all synchronization mechanisms described below.

Note that signs of high contention are monitored promptly so that components can eagerly transform into concurrent variants. The reason is that locks could be bottlenecks hampering global progress of the system, requiring immediate transformation into a concurrency friendly component.

Conversely, signalling low contention and transforming back into the lock based component variant is deferred until there is sufficient evidence that contention has indeed decreased to a single thread for a significant amount of time. The transformation is typically not necessary to prevent a serious performance bottleneck in the sense that scalability is prevented. It may however provide better performance if executed frequently.

4.1 Locks

The contention sensor for locks is quite important as locks tend to be the biggest bottlenecks in concurrent programs. Therefore, we must take extra care of minimizing the cost of this contention monitoring.

Modern virtual machines (VMs) employ adaptive locking schemes with support for biased locking. We will assume an ideal solution for us, presented by Pizlo et al. [26], that performs better when locking is done from a single thread, and adaptively spins when there is actual contention. We hereby define three levels of contention for a lock: 1) biased, 2) unbiased spin-lock, 3) blocking lock. Each level has its own type of lock that needs to be installed.

Level 1 is referred to as biased locking. The first owner is established using compare-and-swap. Once it has been established that the lock is biased towards a thread, it may lock and unlock simply by flagging 1 (locked) or 0 without atomic instructions. If another thread attempts to enter the critical section, it calls for bias revocation by invoking a local safepoint that freezes the owning thread and promotes its lock to level 2.

Level 2 is referred to as spin-locks. They spin for a bounded amount of time, waiting for the lock to be released. This is faster on SMP systems than blocking if the lock will soon be released. If, however, spinning does not end in time, the lock gets promoted to level 3.

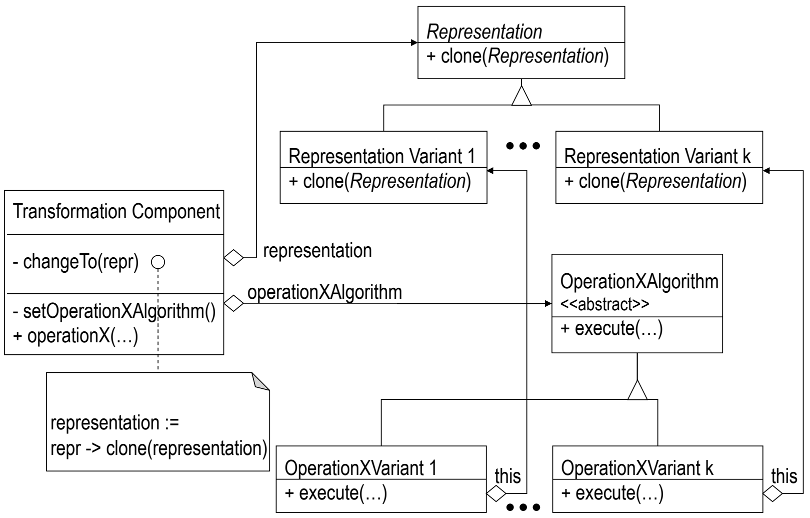

Figure 1: Transformation Component. UML diagram of the conceptual design pattern. Implementation design could be different to optimize performance.

Level 3 is referred to as blocking lock. If spinning for a bounded number of times was not enough, the thread blocks until it may be acquired again. On this level, it additionally signals wether blocking was necessary or not. This comes at no additional cost since the cost of blocking is significantly higher than the monitoring code.

Contention sensors allow monitoring changes between different contention levels. Fortunately, since each contention level corresponds to a new type of lock being installed into the cell header, the performance cost of such contention monitoring is insignificant since it does not get invoked for each lock operation (especially fast paths), but only for the slow paths when locks are promoted due to contention or demoted due to lack of contention.

4.2 Lock-Free

In a lock-free component, sensing the absence of contention would be useful. In order to do this, we piggyback on a garbage collector (GC) and use a similar idea as normal *lock deflation*, i.e., adaptive locking that decides to demote a level 3 type lock and to install a lower level. The VM normally aggressively deflates locks that are not held during garbage collection. Similarly, the lock-free component creates a probe object which is garbage on creation (no references to it). It has a finalizer which can check the ownership of a lock-free component when the probe becomes finalized by the garbage collector at the next GC cycle. If the component indeed was exclusively owned by a single thread, the probe did not sense any contention.

Deciding this exclusive ownership when the probe finalizes can be done in two ways. In the first approach, *accurate probing*, threads read a special volatile field of a lock-free component for determining its owner thread. For each operation the component checks if the owner is `NONE`. If so, the thread could attempt to install itself as owner *safely* using CAS. If the CAS fails or the owner was not `NONE`, then `MULTIPLE` is written to the owner field, symbolizing that the component is used by multiple threads. Now the probe can see, when triggered by the GC finalizer, that the status is either `NONE`, a single thread or `MULTIPLE`. This value stabilizes quickly and its cache line can be shared with no conflicts.

In practice, since monitoring is only providing hints for when to transform, the field can be updated using normal memory accesses for increased performance. This approximation could be a victim of data races and hence provide inaccurate hints. However, that does not affect the consistency of the component, only the hints used for optimization.

The second approach, *random probing*, always decides to transform back to the single-thread optimized variant regardless of the actual contention, for a chance to eventually revise the assumption that contention is high. Obviously, this could be the wrong decision, causing an immediate (and unnecessary) transformation back to the concurrency favoured variant again. However, this only happens infrequently (once every GC cycle), and the amortized cost of such bad transformations is low.

4.3 Transactional Memory

Monitoring contention in TM is particularly trivial: simply count the number of transactions free from data conflicts (which must be tracked anyway) and feed it into the contention manager.

5. SAFE COMPONENT INVALIDATION

Knowing that a particular component variant is not suitable is not enough; we also need to address issue (ii) and convert to a new variant safely and efficiently. We propose *component variant invalidation* as a mechanism to invalidate the previous unsuitable component variants in the sense that no operation (read or write) may complete nor harm the consistency of the component once invalidation has finished. We provide safe component variant invalidation mechanisms for each synchronization mechanism.

Note that in order to instantiate and initialize a new more suitable component variant, it must still be possible to read the state of the invalidated component variant. If all reading methods throw exceptions it becomes impossible to read the last valid state, hence installing a new component variant is not possible either. Therefore every component variant must provide a special yet general read-only version, e.g., a read-iterator, which is only used when the component variant is rendered invalid and does not need protection. The approach we chose is to provide readable clone methods, Representation readableClone(), for every representation variant.

5.1 Locks

The general solution makes lock invalidation a part of the locking protocol. That is, a lock can be permanently invalidated requiring one extra bit for the locking protocol state. A slow path can be executed when it is entered and invalid, causing an exception to be thrown to the component which handles this accordingly. If integrated into the locking protocol's slow path, this operation is free of charge when using the fast paths which need to be optimized.

For the locking, readableClone() returns a new object wrapper with the same internal data, but with a new lock allowing access to the representation variant.

5.2 Lock-Free

For discussing lock-free algorithms we first need to differentiate linearizing CAS operations from not linearizing CAS operations. A linearizing CAS is a CAS that atomically commits the change made by an operation at a linearization point [15] and either fails to commit due to contention (and hence making the wanted change not visible at all to other threads) or succeeds (and hence making the new state of the component completely visible to all other threads). There may be other not linearizing CAS operations that update the component lazily and if not successful, they do not hamper the consistency of the component.

Our main idea is the following: Lock-free components are built using our own invalidatable IAtomicReference<T> that has a special linearizingCAS() operation. This class blocks linearization points in lock-free algorithms from committing its changes after an invalidation of a reference. Instead it throws an exception to signal that it has been invalidated. A load of the reference, get(), will also throw an exception.

How it works in detail: The generic type argument T is required to be a subclass of Object. An internal atomic reference assumes the Object type, and get() casts the internal load from Object to T. Instances of a special internal class Invalidated<T> represent invalidated references, and get() will trigger a class cast exception (since Invalidated<T> is not a T) indicating that the reference has become invalid. Similarly, the linearizingCAS() operation returns immediately if the internal CAS worked, otherwise it checks if it failed due to invalidation by loading the current value and checking if it is an Invalidated<T> object. Any normal non-linearizing CAS operation does not need to throw such exceptions as the consistency of the data structure is not harmed by them according to the algorithm specification.

Invalidation of a reference is performed by installing an Invalidated<T> object containing the original value using CAS in a loop that terminates when CAS succeeds.

Like with the other solutions, there must be a read-only variant that will not throw exceptions, allowing copying of the old invalidated representation variant and create a new one. This is done with a special T loadLastValid() method implemented in IAtomicReference<T>, which is also used to implement the readableClone() methods of the representation variants.

To invalidate a component variant, it is traversed using T loadLastValid() and then calls invalidate() on all existing references. When the whole representation variant has been traversed, it is certain that the component variant is logically invalidated in the sense that any linearization point for any operation will fail and throw an exception.

5.3 Transactional Memory

Implementing safe component variant invalidation for TM is done by simply inserting a data dependency in the beginning of the transactions, checking a flag if the representation variant is invalid; if so it throws an exception, otherwise it continues executing. When the component variant is invalidated, the flag is set causing current transactions to fail. In a strongly consistent TM the flag is set normally, while in a weakly consistent TM the setting of the flag is wrapped in a transaction. If the flag checking operation's transaction failed it checks if the invalid flag is set, and if so throws an exception like the locking approach. Unlike the locking invalidation, we do not know how to make this free of charge in the common case as infusing a data dependency is integral for making the transaction fail.

Assuming write-buffered TM, readableClone() methods are implemented by using plain memory read operations.

5.4 Safe Component Transformation

Now we can, for all synchronization mechanisms, sense inappropriate component variants due to changing contention and safely invalidate them for any synchronization mechanism. In order to complete the transformation there must be a way of handling transformation races. We simply solve that uniformly for all synchronization mechanisms by wrapping the actual transformation in a synchronized block with a lock owned by the contention manager. We argue that if contention for transformation is so heavy that this becomes a bottleneck, choosing to transform is probably a bad decision in the first place and the contention manager should simply not trigger such excessive transformations.

6. EVALUATION

In this section we assess transformation components in different contention contexts and compare them to components using a single, monolithic synchronization mechanism. This will also address the issue (iii), i.e. the question, which changes in the contention level ought to trigger transformation since we observe the monolithic champion variant in each contention context. It turns out that the lock-free solutions outperform the lock-based solutions already if two threads are working concurrently in all observed cases. Hence, our transformation components switch to these solutions as soon as contention is detected.

6.1 Implementation Deviations

In Section 4, we defined the adaptive locking protocol for promoting and demoting monitors. However, our implementation was integrated into OpenJDK 8 that follows a slightly different locking scheme. First locks are biased towards one thread as before. When it turns out that the lock was not exclusively used by a single thread, a local safepoint is issued to rebias the lock towards another thread or to revoke the bias. OpenJDK supports bulk rebiasing where every instance of a class gets its lock bias revoked.

In OpenJDK, it can happen that a lock is allocated on the stack, e.g., of an interpreter. When the code needs to be optimized by the JIT-compiler such stack allocated locks get inflated when on-stack-replacement (OSR) is performed to replace the current stack frame with a new native stack frame. This can lead to bulk revocation of all biases of all currently held locks, not due to contention, but due to migration from the interpreter frame to JIT-compiled frames. Of course, this is not an inherent problem; it was simply easier exploiting the given environment. A locking protocol could indeed be implemented without false lock promotion.

As a work around to not affect the benchmarks, the benchmarks have a warming up phase that makes sure everything gets JIT-compiled and that no OSR will disrupt the results.

Finally, we did not use transactional memory. We were only interested in high performance in this paper, not in transactional composition of concurrent components. Combining the benefits of transactional composition with other synchronization mechanisms using transformations is possible but evaluation of this is outside the scope of this paper.

6.2 Components and their Variants

Three different components – queues, sets, and ordered sets were implemented. OpenJDK already comes with implementations of these in the concurrency package: ConcurrentLinkedQueue, ConcurrentHashMap and ConcurrentSkipList, all implemented by Doug Lea. We added transformation components of these and evaluated them experimentally in Section 6.3.

6.2.1 *Queue*

Concurrent queues are used in many applications. Therefore, we chose to evaluate the performance of a transforming concurrent queue as one of our candidates. Initially, it assumes little contention using a coarse-grained lock and a normal LinkedList implementation. At the first blocking operation for the lock, it transforms into a lock-free queue, i.e., a ConcurrentLinkedQueue. A transformation back to the normal LinkedList is optionally supported using invalidatable references combined with random or accurate probing.

The lock-based solution uses a double-linked list for performance improvements. By using a cyclic linked list with a sentinel header node connecting the endpoints, branches can be omitted to check for null endpoints.

The lock-free queue is based on [23] and uses CAS for synchronization. It is only single-ended because maintaining double-ended linked lists is difficult (although not impossible). The core idea is to CAS the next references of the linked list as linearization point, and then CAS the tail lazily using helper methods. Nodes are logically removed by changing the element of an internal node to null. It is then lazily removed physically from the linked list.

6.2.2 *Set*

The second data structure we tried is sets. In particular, we focused on hash tables. The ConcurrentHashMap from the Java concurrency package, splits multiple locks over multiple buckets. The size of the table is scaled up by the number of *expected* threads (which is 16 unless otherwise stated) to make cache interference and synchronization less common.

Although memory footprint was not a constraint of this paper and is not shown in benchmark results, keep in mind that they use significantly more memory. The hash table is split into different segments, each individually functioning like a hash table. The segment is locked when inserting and deleting. Therefore, this implementation is not lock-free but more fine-grained in its locking scheme for scalable performance. When a thread performs contains() on a segment, it tries a bounded number of times without locking and then resorts to locking the segment.

6.2.3 *OrderedSet*

The last example is ordered sets. For the single thread optimized implementation, we used normal red black trees with a coarse grained lock (synchronized TreeSet).

There is an inherent difficulty of making lock-free algorithms for trees since they have two children that may mutate arbitrarily. Therefore, the tree-like skip list was picked instead for the concurrent case (ConcurrentSkipList). The ConcurrentSkipListMap is lock-free and based on CAS for linearization points. It basically works as a multi-level lock-free linked list where the number of levels picked for each node inserted is picked from a random distribution rather than deterministically balanced. The concurrent skip list inserts dummy marker nodes to denote a node has been logically deleted and are then lazily deleted physically.

The transforming variant starts as a red black tree, and then transforms into a ConcurrentSkipList for improved concurrency at signs of actual contention.

6.3 Experimental Setup

All benchmarks were run on a MacBook Pro with 4 Intel i7 CPU cores, 8 hyperthreads, 256 KB L2 cache (per core), 6 MB L3 cache (shared), 2 x 8 GB DDR3 1600 MHz RAM, running on Mac OS X 10.9.1. Benchmarks were run on our own modified OpenJDK 8.

Threads 1 to 4 get exclusive CPU cores. Threads 5 to 8 get hyperthreads. Threads 9 to 16 have no such benefits.

By default biased locking would not get activated until the benchmark is over. Therefore it was forced to start immediately. Escape analysis was disabled because using static analysis to elide locks in the benchmarks would be unfair. In practice, our transformation components would benefit from such optimizations. The JVM arguments were -XX:+UseBiasedLocking -XX:BiasedLockingStartupDelay=0 -XX:-DoEscapeAnalysis.

Time is measured as the net of the wall time across all threads, invariant of lock spinning and similar optimizations. The elements in all benchmarks are random integers.

Note that for the sequential scenario, the single thread favoured implementations would win even more if the size() method was run since they only return the loaded value of a field. Concurrent components have to traverse the data structure and calculate the total size. This makes transformation even more motivated. Yet, we chose not to show such benchmarks because the gain is proportional to the size of the data structure. Such a biased benchmark is of little value. To be more objective, we focus on the constants.

6.4 Case 1: Concurrent Queues

In order to evaluate the performance of our queue, a micro benchmark was used. A number of threads p pick operations enqueue and dequeue according to a random distribution and execute them n times. The results for one thread are shown in Figure 2 and the corresponding benchmark results for multiple threads are shown in Figure 3.

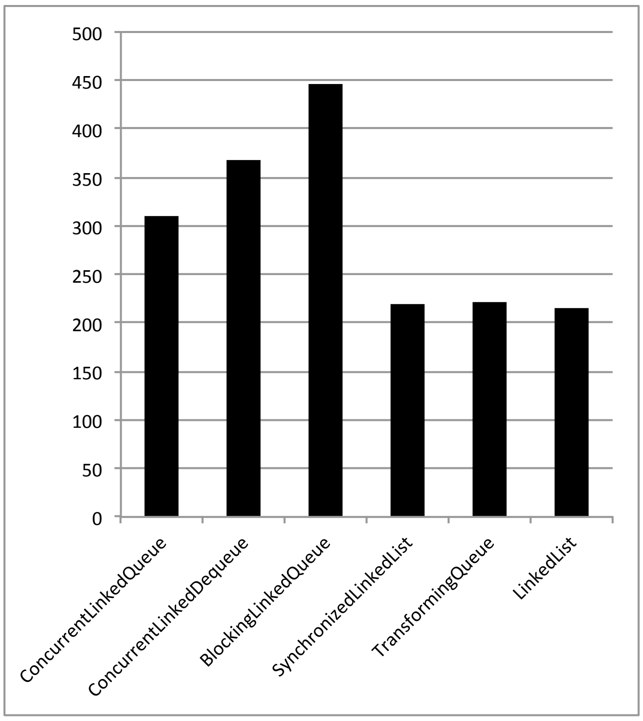

Figure 2: Time for 10,000,000 randomly distributed enqeue/dequeue operations on a single thread in μs. Average size of the data structures is 1000 elements.

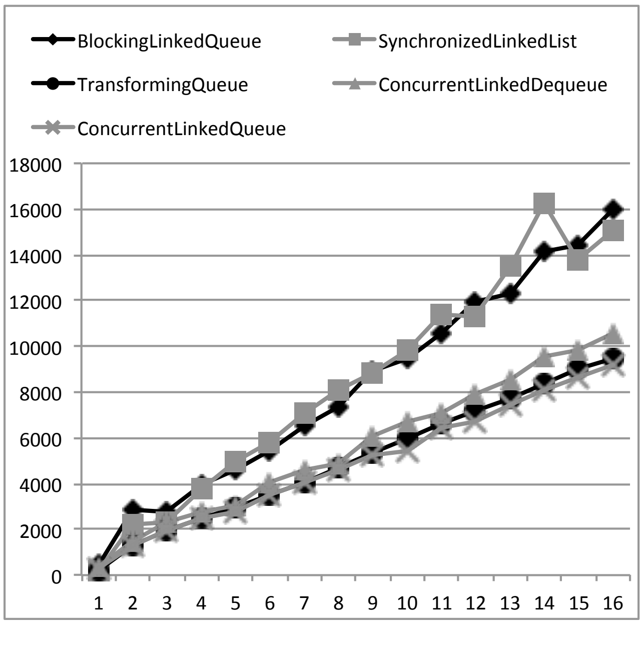

Figure 3: Net time for a total of 10,000,000 randomly distributed enqeue/dequeue operations across all threads in μs. Average size of the data structures is 1000 elements.

We observe that the sequential queue without thread safety performs as well as the variant with biased coarse grained locks in absence of concurrency. We note that the performance of using synchronized vs not using it is negligible since the biased lock is exclusively owned by one thread. The BlockingLinkedQueue uses the ReentrantLock class which does not have a biased locking scheme; this explains the poor performance on one thread. The conditionally lock-free queue ConcurrentLinkedQueue is based on [23]. Note that the x86_64 architecture has a very fast CAS implementation. Still, the performance is considerably worse than the single thread optimized double linked list. This is partially due to branches not being necessary in doubly linked lists with a sentinel node, while a single linked list has special cases if the queue is empty. The ConcurrentLinkedDequeue, a lock-free double-linked list, performs not as good as the ConcurrentLinkedQueue but better than the BlockingLinkedQueue.

Finally, the TransformingQueue optimistically starts as a synchronized linked list. When the lock is inflated (level 3), it transforms to a concurrent linked queue. Since the overhead of checking whether transformation is necessary is triggered only upon lock inflation, we perform optimally on one thread as seen in Figure 2.

As the number of contending threads grows (Figure 3), the concurrent linked queue shows the best performance. For the enqueue and dequeue operations that are $O(1)$, the synchronization overheads are completely dominating performance. Locks require two CAS instructions - one to lock and one to unlock. The concurrent linked queue, however, requires only one and a half CAS on average for each operation, exploiting a lazy updating trick for tail references. Since it both has higher concurrency (by not having to do everything sequentially) and uses less synchronization overhead, it always has better performance.

Once again, unsurprisingly, we see that the TransformingQueue follows the ConcurrentLinkedQueue closely. Therefore the transforming component shows the optimal performance on all levels of contention.

The first part of this case evaluated performance when transforming from a lock-based to a lock-free component. The second part evaluates the costs of possible transformations back from a lock-free to a lock-based component. We quantified the costs of using invalidatable references in conjunction with accurate probing and random probing. In the concurrent contexts, the TransformingQueue shows a performance loss of 1.9% and of 1.5% on average due to accurate probing and random probing, resp. This is an insignificant cost and will hence not be investigated more in subsequent experiments.

6.5 Case 2: Concurrent Sets

We measured the results of running a benchmark with the operations add (15%), remove (15%) and contains (80%), randomly distributed on a number of threads. The results of the single threaded version context are presented in Figure 4 and the multi threaded version results in Figure 5.

The ConcurrentHashMap is the Java concurrency hash table. The HashSet is simply a normal hash set (although with some logic to turn large buckets into trees rather than linked lists at certain thresholds for better worst time complexity. The SynchronizedHashSet is simply a HashSet wrapped in synchronized blocks. The TransformingHashSet starts as a (memory efficient) hash set and then transforms into a concurrent hash map when the lock gets inflated.

In the single threaded benchmark (Figure 4) the observed performance differences are not massive. This is under-

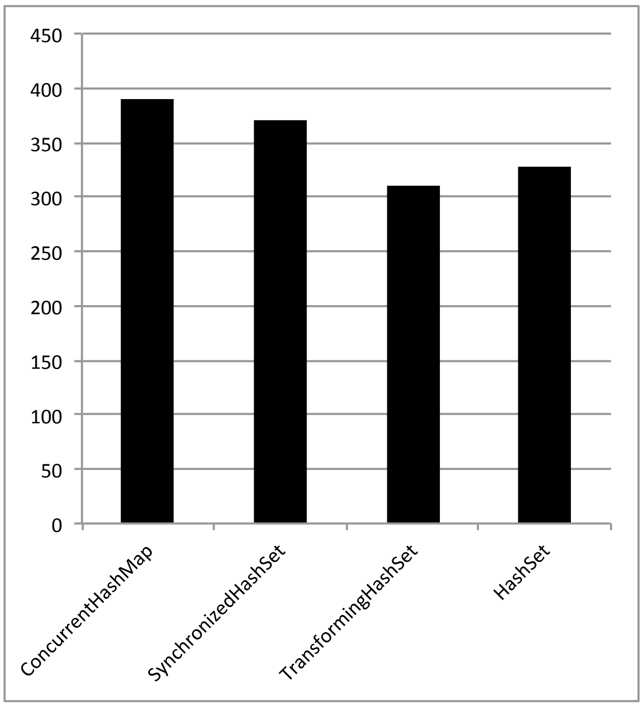

Figure 4: Time for 10,000,000 randomly distributed add (15%), remove (15%) and contains (70%) operations on a single thread in μs. Average size of the data structures is 1000 elements.

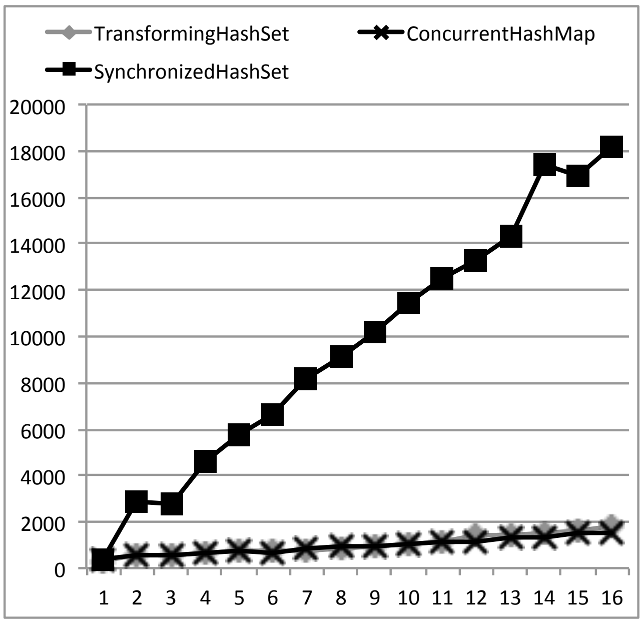

Figure 5: Net time for a total of 10,000,000 randomly distributed add (15%), remove (15%) and contains (70%) operations across all threads in μs. Average size of the data structures is 1000 elements.

standable since a concurrent hash set is essentially a larger hash set using fine grained locking. Still it is slower than the single thread optimized variants, and also has significantly larger memory overheads. The TransformingHashSet does not treeify large buckets and does not indirectly use a HashMap, and is hence slightly cheaper in performance compared to HashSet on average.

For locking, the ConcurrentHashMap variant uses the ReentrantLock class which is slower than native biased locking. This is why it is slower than the synchronized hash set in the uncontended case. It also has a slightly more expensive hash function that improves hashing with respect to both segments and buckets in segments.

In the concurrent benchmark, the improved concurrency of the concurrent hash set becomes obvious. Since 70% of the operations are contains(), which rarely needs locking, and then only for a single segment, the concurrency is increased. Even the add and remove operations need to lock only one (randomly distributed) segment.

The transforming variant once again follows closely the lower bound and hence performs best regardless of the contention, while in this case also keeping lower memory footprints when there is no contention.

6.6 Case 3: Concurrent Ordered Sets

The same benchmark as for the normal sets was run and results are shown in Figure 6 for the uncontended and in Figure 7 for the contended case.

For the uncontended benchmark (Figure 6), the red black tree with its synchronized counterpart is faster than the concurrent skip list. The explanation is most likely that the biased locks do not need atomic CAS instructions once the bias has been established, while the concurrent skip list performs CAS all the time. Moreover, a lazy deletion scheme is used for the ConcurrentSkipList: a special marker node is created and installed in the list to signify deletion has occurred. This requires memory to be allocated when deleting and some extra traversal to be done, ignoring the deletion marker nodes. Also, the balancing of the concurrent skip list is random and possibly suboptimal while the red black tree follows rules with a worst case depth of the tree.

Finally we note that the transforming variant performs as well as the red black tree. This is not surprising because red black tree is the initial variant picked.

In the concurrent case (Figure 7) the point of using the concurrent skip list becomes obvious. The coarse grained lock solution is much slower because it allows no concurrency, while the concurrent skip list allows all threads to perform lookups and updates on disjoint data concurrently.

As before, the transforming variant consistently follows the lower bound by transforming to a concurrent skip list when the lock becomes inflated and the system identifies the concurrency bottleneck.

7. RELATED WORK

A number of approaches to STM [13, 27, 12, 6] were attempted as universal constructions but never saw any success in practice. Similar ideas were presented for HTM [2, 11] and hybrid TM [5, 4]. HTM became more popular with the introduction of the Intel Haswell processors employing HTM. However, they have bounded write buffers and their availability can not be guaranteed.

Some researchers argue that the best approach is to build a *manually* fine tuned library of reusable concurrent com-

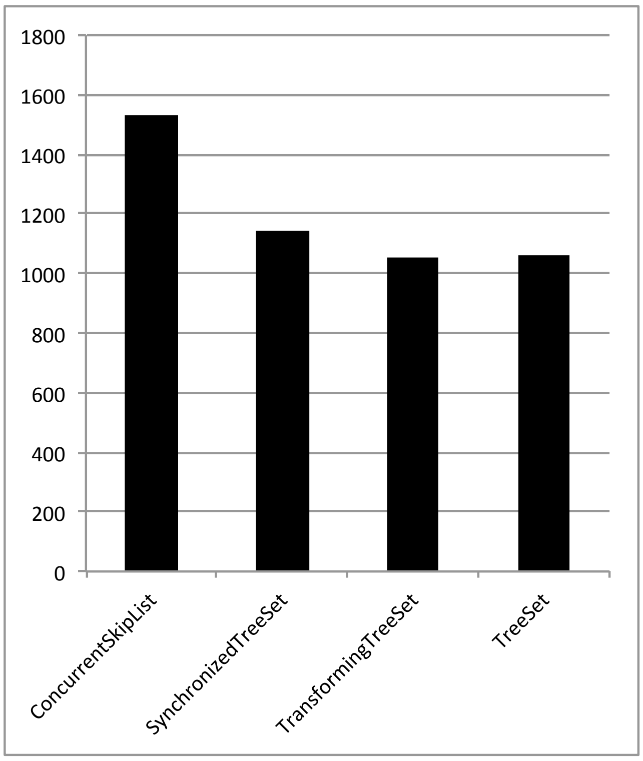

Figure 6: Time for 10,000,000 randomly distributed enqeue/dequeue operations on a single thread in μs.

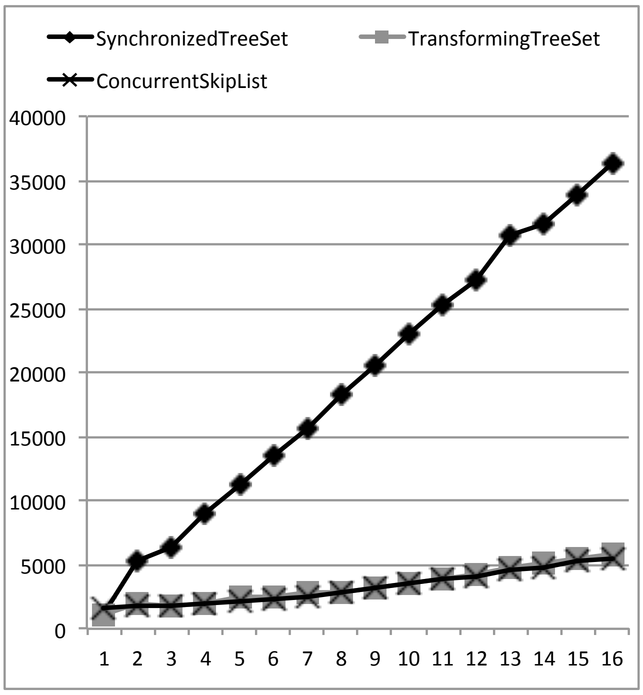

Figure 7: Net time for a total of 10,000,000 randomly distributed enqeue/dequeue operations across all threads in μs.

ponents without locks. Many lock-free data structures have been implemented [23, 14, 9, 21]. Recently with the publication of methodologies for constructing wait-free data structures [19] it became straight forward to also make previously lock-free components wait-free [18, 30]. However, their progress guarantees depend on the progress of the underlying memory management.

Felber et al. [8] presented an adaptively fine tuning synchronization strategy in word-based STMs, yielding better STM performance than before by adapting to the context.

Pizlo et al. [26] presented an adaptive locking protocol, Fable, for monitors in Jikes RVM. This provided better performance by adapting and learning from the context.

These two papers only adapt the synchronization details, not the components exploiting synchronization mechanisms.

Dynamically transforming data structures were independently introduced in [25] and [31]. Their concept is closely related to context aware composition [3, 16]: offline and online machine learning approaches find the best implementation variants in different contexts and transform to these variants at runtime. The addition of this paper is the ability to efficiently sense contention level as a new context attribute and to safely and efficiently transform to the component variant fitting best in each such context.

Dig et al. [7] presented a static analysis that could refactor sequential into thread safe data structures and Kjulstad et al. [17] presented a similar analysis for converting mutable objects to immutable objects. We believe these static approaches are complementary to our dynamic approach.

8. CONCLUSION

A framework for building concurrent transformation components was introduced. It bridges the gap between the best performing sequential component variants and the best performing concurrent component variants by automatically transforming between them dynamically and transparently for the usage context. These component variants can be implemented using completely different synchronization mechanisms, e.g. lock-based, lock-free or TM.

Contention sensors sense contention changing at runtime and contention managers eagerly transform to concurrency friendly component variants as actual contention increases. They may revise the decision and eventually transform back to sequential component variants if contention stops.

The approach was evaluated using transforming queues, sets and ordered sets exposed to different contention levels. The transforming components have performance on par with the best solutions for each contention context.

The main merit of this paper is that programmers do not have to think about picking the right component for the right contention and can focus on other tasks instead, knowing that the component is thread-safe yet always performs as well as possible. It automates a design and optimization task which, in general, neither a component designer nor a component user nor a static compiler can *effectively* make.

9. ACKNOWLEDGMENTS

This research was supported by the Swedish Research Council under grant 2011-6185. We would also like to thank the anonymous reviewers for their insightful feedback that helped to improve this paper.

10. REFERENCES

[1] G. M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485. ACM, 1967.

[2] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 316–327. IEEE, 2005.

[3] J. Andersson, M. Ericsson, C. W. Keßler, and W. Löwe. Profile-guided composition. In *Software Composition*, pages 157–164, 2008.

[4] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11):336–346, Oct. 2006.

[5] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ACM Sigplan Notices*, volume 41, pages 336–346. ACM, 2006.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In *Distributed Computing*, pages 194–208. Springer, 2006.

[7] D. Dig, J. Marrero, and M. D. Ernst. Refactoring sequential java code for concurrency via concurrent libraries. In *Proceedings of the 31st International Conference on Software Engineering*, pages 397–407. IEEE Computer Society, 2009.

[8] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[9] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 50–59. ACM, 2004.

[10] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[11] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ACM SIGARCH Computer Architecture News*, volume 32, page 102. IEEE Computer Society, 2004.

[12] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[13] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[14] M. Herlihy, N. Shavit, and M. Tzafrir. Hopscotch hashing. In *Distributed Computing*, pages 350–364. Springer, 2008.

[15] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[16] C. Kessler and W. Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, 2012.

[17] F. Kjolstad, D. Dig, G. Acevedo, and M. Snir. Transformation for class immutability. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 61–70. ACM, 2011.

[18] A. Kogan and E. Petrank. Wait-free queues with multiple enqueuers and dequeuers. *ACM SIGPLAN Notices*, 46(8):223–234, 2011.

[19] A. Kogan and E. Petrank. A methodology for creating fast wait-free data structures. In *ACM SIGPLAN Notices*, volume 47, pages 141–150. ACM, 2012.

[20] W. Löwe, R. Neumann, M. Trapp, and W. Zimmermann. Robust dynamic exchange of implementation aspects. In *TOOLS (29)*, pages 351–360, 1999.

[21] M. 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*, pages 73–82. ACM, 2002.

[22] M. M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *Parallel and Distributed Systems, IEEE Transactions on*, 15(6):491–504, 2004.

[23] 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*, pages 267–275. ACM, 1996.

[24] M. Moir. Transparent support for wait-free transactions. In *Distributed Algorithms*, pages 305–319. Springer, 1997.

[25] E. Osterlund and W. Lowe. Dynamically transforming data structures. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 410–420. IEEE, 2013.

[26] F. Pizlo, D. Frampton, and A. L. Hosking. Fine-grained adaptive biased locking. In *Proceedings of the 9th International Conference on Principles and Practice of Programming in Java*, pages 171–181. ACM, 2011.

[27] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.

[28] N. Shavit and D. Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[29] H. Sundell. Wait-free reference counting and memory management. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 24b–24b. IEEE, 2005.

[30] S. Timnat, A. Braginsky, A. Kogan, and E. Petrank. Wait-free linked-lists. In *Principles of Distributed Systems*, pages 330–344. Springer, 2012.

[31] G. Xu. Coco: Sound and adaptive replacement of java collections. In *27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*, pages 1–26. Springer Berlin Heidelberg.