Autom Softw Eng
DOI 10.1007/s10515-017-0219-0

Self-adaptive concurrent components

Erik Österlund[1] · **Welf Löwe**[2]

Received: 15 July 2016 / Accepted: 27 July 2017

Abstract Selecting the optimum component implementation variant is sometimes difficult since it depends on the component's usage context at runtime, e.g., on the concurrency level of the application using the component, call sequences to the component, actual parameters, the hardware available etc. A conservative selection of implementation variants leads to suboptimal performance, e.g., if a component is conservatively implemented as thread-safe while during the actual execution it is only accessed from a single thread. In general, an optimal component implementation variant cannot be determined before runtime and a single optimal variant might not even exist since the usage contexts can change significantly over the runtime. We introduce self-adaptive concurrent components that automatically and dynamically change not only their internal representation and operation implementation variants but also their synchronization mechanism based on a possibly changing usage context. The most suitable variant is selected at runtime rather than at compile time. The decision is revised if the usage context changes, e.g., if a single-threaded context changes to a highly contended concurrent context. As a consequence, programmers can focus on the semantics of their systems and, e.g., conservatively use thread-safe components to ensure consistency of their data, while deferring implementation and optimization decisions to context-aware runtime optimizations. We demonstrate the effect on

This research was supported by the Swedish Research Council Under Grant 2011-6185.

✉ Erik Österlund
erik.osterlund@oracle.com

Welf Löwe
Welf.Lowe@lnu.se

[1] Oracle, Stockholm, Sweden

[2] Software Technology Labs, Department of Computer Science, Linnaeus University, 351 06 Växjö, Sweden

Published online: 22 August 2017

performance with self-adaptive concurrent queues, sets, and ordered sets. In all three cases, experimental evaluation shows close to optimal performance regardless of actual contention.

Keywords Context-aware composition · Self-adaptive components · Concurrent context

1 Introduction

A software component exposes its functionality via an application programming interface (API) while hiding its implementation details. Selecting the right implementation variant of a component for a given task can be tedious and time consuming. The component implementation performing best in the worst case may perform worse in the actual executions. Yet programmers tend conservatively optimize for the worst case, e.g., they use the component implementations that scale best, even though scalability is not an issue and, hence suboptimal, in the actual executions.

Components that can run safely in concurrent, i.e., contended, contexts[1] have become increasingly important since the rise of symmetric multiprocessing (SMP) architectures. We refer to them as *concurrent components*. A common problem programmers face when implementing concurrent components is that the level of runtime contention of a component, i.e., the concurrency a component is exposed to, is unknown at development time. If multiple (increasingly deep) layers of APIs depend on one another, it becomes increasingly difficult to know which level of contention a component will be exposed to in the end. Yet again, often a conservative approach is taken. A thread-safe component is picked if there is a possibility that an API is used in a concurrent context in some possible deployment, e.g., by a multi-threaded application.

However, if the actual contention was known, alternative component implementation variants may be preferable. If contention is high, it may be beneficial to use an implementation based on lock-free or wait-free data structures rather than an implementation using mutual exclusion locks. Likewise, if contention is low or synchronization is not needed, implementations that block or disregard concurrency would be more beneficial. The contention is a property that can change throughout program executions and the respectively optimal component implementation does so as well.

Since optimizing single thread performance in hardware got increasingly difficult the last decade, both hardware and operating system (OS) vendors added more and more synchronization features, exposed at different levels of an application stack, making the choice of synchronization mechanisms more difficult than ever for programmers. The same component implementation might perform well in one deployment environment and bad in a different one. With emerging cloud platforms, that environment might not even be known until runtime.

[1] The notions "context" and "context-aware" are overloaded: "context" often refers to the CPU state such as registers and stack variables or the calling context. Context-aware composition and self-adaptive components use a generalized notion of a calling context and the present paper focuses on the contention context, i.e., the number of concurrent threads calling functions of a component.

A lot of research has focused on improving the synchronization mechanisms to cope with varying contention. Adaptive spin locks (Pizlo et al. 2011) and biased locking (Russell and Detlefs 2006) partially addresses this issue by adapting to and, hence, providing higher performance for contended and uncontended contexts, respectively.

However, the nature of mutually exclusive locking tends to imply scalability bottlenecks. Specialized lock-free component implementations (Michael and Scott 1996; Kogan and Petrank 2011; Herlihy et al. 2008; Fomitchev and Ruppert 2004) provide even better concurrent performance, by limiting contention to actual data conflicts, carefully hand tuning the algorithms to spatially distribute those data conflicts, as well as even allowing some data conflicts not violating the consistency of the components. These hand-crafted highly specialized concurrent component implementations tend to scale best when available. However, due to the use of atomic instructions and stricter memory models required for concurrent consistency, they are suboptimal in uncontended contexts.

Even more research has been focused on universal "silver bullet" constructions like transactional memory (TM) (Herlihy and Moss 1993) that would automatically turn sequential components into scalable concurrent components without mutual exclusion. First it was implemented in software (STM) (Herlihy et al. 2003; Saha et al. 2006; Felber et al. 2008) to provide ease of use and scalable concurrent performance. Then it was implemented in hardware (HTM) (Hammond et al. 2004; Ananian et al. 2005) to get constants down, but it lost some ease of use such as transactional composition. Hybrid variant (Damron et al. 2006) combined the transactional composition of STM with the performance of HTM.

Existing sequential components were optimized for sequential contexts and never had concurrency in mind. They did not minimize data dependencies and, e.g., deliberately rely on an updated size variable for each operation. They did not consider making disjoint memory accesses whenever possible, which is required for good concurrent performance. Therefore, after being automatically transformed using TM, non-essential data dependencies have to be removed manually to scale well in concurrent contexts. However, the removal of those data dependencies, e.g., by adding re-computations of a size variable for each operation, makes them then perform worse in sequential contexts. Even in the concurrent contexts they were optimized for, transactional memory does not allow certain non-essential data conflicts to happen without aborting in the way that specialized lock-free data structures do. Hence, they can not compete with lock-free data structures in the concurrent contexts.

The dream of not having to manually pick component variants based on assumed contention traces back to old components like Vector in the Java class library. All methods were "synchronized" so that programmers could assume thread-safety always. Unfortunately, the approach had bad uncontended performance, and increasingly bad concurrent performance as number of cores grew, due to the use of mutual exclusion locks. Therefore, the dream was abandoned in later generations of the class library, and the responsibility of picking the appropriate component variant became the burden of programmers again.

This paper rejects the idea of a single "silver bullet" synchronization mechanism that performs optimally on all levels of contention. Instead we suggest uniting the

different mechanisms with an architectural solution, *self-adaptive concurrent components*, that allows them to coexist and complement each other and combines their individual strengths. They regard contention as a context attribute and automatically transform at runtime between different component implementation variants. This way they provide superior uncontended performance and superior contended performance at the same time, while exposing a single component API for programmers. Self-adaptive concurrent components relieve the programmers from the burden of finding the optimal solution for each actual context; this is done automatically behind their interface. They encapsulate variants of its operation implementations (algorithms), state representations (data structures) and synchronization mechanism behind a well-defined interface and

1. Switch between different algorithms and representations without changing its functional behavior. Observed operation sequences at runtime determine and transform to the expectedly best-fit algorithmic and data representation variants. This idea was first introduced in (Österlund and Löwe 2013).
2. Switch between different synchronization mechanisms seamlessly without violating consistency. The default is optimistic biased locking that is essentially for free when used by a single thread. Contention sensors that do not have any noticeable performance impact automatically sense changes in concurrent contention. At signs of higher contention, the components adapt 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 idea was first introduced in (Österlund and Löwe 2014).

The focus of the present paper is on self-adaptive *concurrent* components. Hence, it discusses how to switch correctly between different algorithms and representations in a concurrent context and how to switch the synchronization mechanism, which is only interesting in a concurrent context.

In our experiments, we evaluate self-adaptive concurrent components extending Java concurrency data structures. We run them on our own modified OpenJDK and HotSpot Java Virtual Machine (JVM) showing that these concurrent components perform (almost) as well as the best known component for each contention context.

The paper is organized as follows: Sect. 2 introduces self-adaptive components using context-aware composition that at runtime selects the presumably optimal algorithms and data representations for each usage context. Section 3 introduces three standard synchronization mechanisms: locks, lock-free algorithms, and TM. Section 4 introduces self-adaptive *concurrent* components using context-aware composition also based on contention as an additional usage context attribute. Section 5 shows how contention can be monitored efficiently, Sect. 6 discusses how to consistently invalidate an outdated component variant, and Sect. 7 shows how the actual component transformation can be implemented efficiently. Section 8 introduces the Java concurrency data structures used in the evaluation, highlights some implementation details and finally describes the evaluation and the evaluation results. Section 9 discusses the related work and Sect. 10 concludes the paper and points out directions of future work.

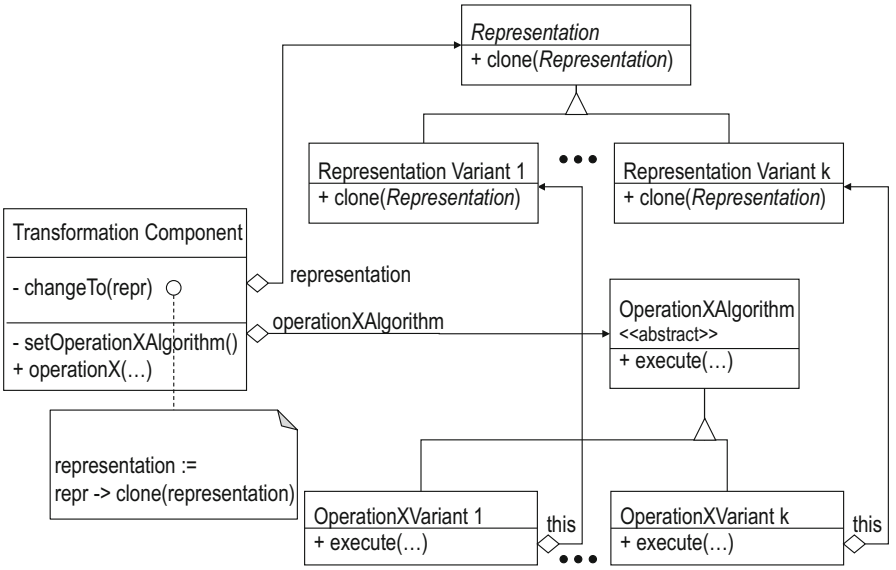

Fig. 1 Transformation Component. UML diagram of the conceptual design pattern. Implementation design could be different to optimize performance

2 Self-adaptive components

Self-adaptive components (or dynamically transforming data structures) as suggested by Österlund and Löwe (2013) build on the previous work of Andersson et al. (2008), Kessler and Löwe (2012) introducing context-aware composition and of Löwe et al. (1999) suggesting transformation components as a general design pattern for data structures with changeable representation and algorithm variants.

2.1 Transformation components

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.

Transformation components follow a general design pattern for data structures with changeable representation and algorithm variants depicted in Fig. 1. It uses a combination of the well-known *bridge* and *strategy* design patterns (Gamma et al. 1995).

A transformation component holds 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 the data representation variant.

State migration between data representation variants allow a component to transform its data representation. Therefore, each representation variant implements a Representation clone(Representation) method accepting an instance of a previous (outdated, unknown) representation variant as a parameter and returning the new actual representation variant instance.

The changeTo() operation of the transformation component 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 *o* the component also holds references to the current algorithm variant implementing *o* on the current representation variant. Algorithm variants are classes specializing an abstract operation class. The abstract class provides an execute() method implemented (differently) by all algorithm subclasses. Calls to an operation are delegated to the current algorithm variant.

In general, one representation variant might have several algorithm variants per operation *o* and for each operation, a setOperation...Algorithm() method sets the expectedly best algorithm for (a sequence of) calls to *o*. Then even a selection of the algorithms adapted to the actual call context is possible and beneficial as shown by Andersson et al. (2008). Without loss of generality, we assume here that the representation variant determines the algorithm variant for each operation and that the *component variants* comprise both the representation and its associated set of algorithm variants.

In the transformation component pattern, the transition from one to another component variant is triggered by an explicit call to the changeTo() or an setOperation...Algorithm() method in the application code. Context-aware composition, in contrast, automatically selects between different component variants based on the actual usage context. Finding the expectedly best (algorithm and data representation) variant, requires the designer to define an optimization goal, e.g., minimize execution time, and formal context attributes, e.g., problem size or number of processors and amount of memory available, which are expected to have an impact on the goal. Formal context attributes, i.e., the actual context, must be evaluable before each call to an operation *o*; the property to optimize must be evaluable after each call. Context-aware composition finds the expectedly best variant automatically using profiling, analysis, and/or machine learning. Using profiling or analysis or a combination thereof, the fitness of variants can be assessed for different *selected* actual contexts. Using machine learning, dispatchers can be trained selecting the expectedly best fitting variants of operations and representation implementation for *each* actual context.

Note that best-fit-learning can be performed offline at design or deployment time, or even online using feedback from online monitoring during program execution (Abbas et al. 2010; Kirchner et al. 2015a, b).

2.2 Self-adaptive components as generalized transformation components

Implementing context-aware composition in the operations of transformation components as an explicit dispatch makes the components *self-adaptive*. Then a call to an operation of a self-adaptive component:

1. evaluates the actual context attributes,
2. decides, based on the actual context, whether a new component variant should be selected and, if so, adapts the component using appropriate calls to the changeTo() or setOperation...Algorithm() method,
3. calls the execute() method of the presumably best-fit algorithm on the presumably best-fit data representation,
4. before returning a result and only in the case of online profiling and learning, evaluates and captures the performance of the operation on the current component variant for future, maybe revised decisions of best-fit component variants.

For integrating a set of *existing* variants implementing the same functional behavior of a component to a self-adaptive component and for replacing these variants with this self-adaptive component in an *existing* environment, certain potential pitfalls ought to be regarded. Below we distinguish and discuss issues in sequential and concurrent environments and how to handle them.

2.2.1 Self-adaptive components in sequential environments

If the internal data representation variant was exposed outside its component and then adapted to a new variant, there could be outdated representations referenced in the heap. Normally this does not happen as the internal representation should not be exposed. Instead all accesses should go through the component interface. Iterators, however, are an exception that needs to be explained more in detail.

Iterators have a reference to the component's internal representation. A transformation triggered during (read) iteration interleaved with any write could lead to inconsistencies. Data structures usually (for good reasons) forbid arbitrary writes during iteration. However, the iterator itself may allow writes to the underlying internal representation. Care must then be taken to handle this case by, e.g., (1) distinguishing an external iterator object (similar to the component's façade object itself) referencing an internal one, and abandoning the internal iterator when its internal data representation is outdated and then constructing a new internal iterator after a transformation, (2) not allowing the iterator access to the internal representation but always accessing through the façade object, or (3) deferring transformation until after iteration.

It may also happen that a variant has side effects either directly or by invoking callback methods in the environment outside the self-adaptive component. It must be assured that all side effects and callbacks with side effects and exactly those happen in the right order in all variants even if not needed in all of them. For instance, a constructor with side effects might never be executed since the variant is not used or, due to transformation, it might possibly be executed several times, or it is executed in a different order with other methods with side effects. Also, a callback with side effects might not occur in a variant but it does in another, or two callbacks with side effects might occur in different orders in different variants.

Especially, in the presence of callbacks, e.g., to `equals` or `hashCode` methods in Java, this cannot be guaranteed without knowing the usage environment. Then, side effect free callbacks ought to be explicitly required in preconditions; they can be enforced by conservative static analysis. It is worth noting that there should not

be a problem if the implementation of those methods follows the specification for `java.lang.Object` (Oracle 2016) requiring that a contract must be followed such that subsequent invocations of `equals` or `hashCode` are consistent.

2.2.2 Self-adaptive components in concurrent environments

Also for concurrent components, all operations may potentially trigger a transformation between variants and perform (lightweight) bookkeeping of usage histories in order to make more informed transformation decisions in the future. This means that even a read operation could cause a mutation while the corresponding read in any of the existing variants (typically) does not. The usage context of a variant might assume that reading operations do not need to be synchronized because there is no mutation. In order for self-adaptive components to work correctly in existing concurrent usage contexts this assumption ought to be regarded.

The solution to this problem could be (1) synchronizing even reading operations, or (2) accepting that the self-adaptive component would make a suboptimal transformation decisions due to inconsistent bookkeeping. The latter is fine as long as it does not happen too often and the amortized time over all operations is shorter. The consistency of the actual data structure is never violated.

Another issue arises from using a variant as a monitor object to synchronize with. The monitor object used has to be the self-adaptive component façade object encapsulating the internal variants, as this is what is exposed to the outside world. This potentially requires code transformations in both the usage environment and the variants where references to `this` object need to be redirected to the façade object.

3 Synchronization mechanism variants

For concurrent execution environments, synchronization mechanisms assure the atomicity of thread access to critical resources. Orthogonally to the component variants, there are different variants of these mechanisms too. 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), and (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.

3.1 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, i.e., locking the whole component instance at once (coarse grained) or some of its objects individually (fine grained), can be varied to achieve scalability.

Problems connected to locks 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 around the complete sequential algorithms implementing an API are usually very efficient. 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 if it is not provable statically that a lock is always held by a single thread exclusively, the lock implementation may optimistically assume so using biased locking (Russell and Detlefs 2006; Pizlo et al. 2011). Therefore, it installs the presumed owner thread the first time it is locked using CAS, cf. Algorithm 1, with a dummy thread id `NO_OWNER` as the expected value indicating that no other thread has claimed to own the lock before. If the owner stays the same, locking and unlocking is performed with normal loads and stores without need for atomics, and is very fast.

```
1  def CAS(expected_val, new_val, address)
2    atomic begin
3      actual_val = *address
4      if (actual_val == expected_val)
5        *address = new_val
6        return expected_val
7      else
8        return actual_val
9      end
10   end
11 end
```

Algorithm 1 The atomic compare-and-swap (CAS) operation compares the contents of a memory address to an expected given value and, if and only if they are the same, modifies the contents of that memory address to a given new value. The caller is notified of the success by either getting returned the expected value (success) or the actual contents of the memory address (failure)

Concurrent Context In concurrent execution contexts, locks suffer from problems predicted by Amdahl's law (Amdahl 1967). 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, with a negative impact on programming and verification efficiency. 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 (almost) sequential execution contexts but may not be the best candidate for a (highly) concurrent one.

3.2 Lock-free synchronization

Lock-free algorithms (Michael and Scott 1996; Kogan and Petrank 2011; Herlihy et al. 2008; Fomitchev and Ruppert 2004) were introduced to deal with the scalability 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 actual data conflicts and, thus, somehow mitigates the problems of Amdahl's law by allowing more work to be done concurrently.

Besides the inherent complexity of these algorithms, not all sequential data structures have lock-free implementations yet. Also, a truly lock-free data structure 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 (Sundell 2005) or hazard pointers (Michael 2004). 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 concurrent component variants.

Sequential Context Lock-free algorithms are typically slower in (almost) sequential contexts 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 data conflicts. In the example of the size attribute, this leads to an $O(n)$ 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.

Beyond this principle issue of lock-free algorithms, their performance greatly depends 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. At least that is true for the officially supported aligned atomics. Misaligned atomics, although not officially supported, typically works still to support legacy software, and reverts back to the older behaviour whenever the memory access crosses two cache lines.

The cost also typically depends on whether the CAS is contended or not. And when it is contended, the cost typically depends on the locality of the thread causing the contention, e.g., if it has shared L2 cache or not.

A CAS requires sequentially consistent 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. Some newer hardware can speculate over fences and continue executing as store buffers are being flushed, but

rolling back in case cache coherence detects that the speculation was unsafe and changes the observable outcome.

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

In other architectures such as POWER, which is weakly consistent, the cost is higher. Here, enforcing the sequentially consistent 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 more lightweight `isync` fence to serialize the pipeline. In the JVM, the lightweight fence is conservatively replaced by another heavy weight `sync`, to accommodate reordering between a potentially latent store conditional and subsequent memory accesses after the specification, which, according to the specification, should not be possible. Nevertheless, the OpenJDK implementation of CAS on POWER uses the more conservative fencing than, e.g., C++11 to accommodate hypothetical broken implementations of POWER.

On POWER, the CAS can also spuriously fail due to context switching between the load link and store conditional. This is called a weak CAS—it admits false negatives but never false positives. Normally, the stronger behaviour is desired by users of CAS that do not care about such particularities. Therefore, generic CAS implementations typically guarantee strong CAS semantics by checking for false negatives and then retrying in a loop.

Lock-free algorithms can also rely on volatile memory accesses (other than CAS) with acquire-release semantics whose performance may similarly depend on which architecture it runs on. Machines with a total store order (TSO) typically do this cheaply, whereas machines with relaxed memory ordering (RMO) need some kind of lightweight fencing like `lwsync` on POWER.

Concurrent context is where the merits of lock-free programming become visible. Lock-free components commit changes at linearization points, cf. (Herlihy and Wing 1990), using CAS. Optionally they lazily, i.e. asynchronously on any subsequent operations, update references that are not necessary for consistency, but for improving time complexity. Therefore, only true data conflicts violating the consistency of the component require operations to be restarted and delay the execution significantly. Other data conflicts not necessary for maintaining the consistency can be tolerated and handled lazily, i.e., without restarting the operation. This makes lock-free algorithms very scalable in terms of performance and typically the best option if available in contended contexts.

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 their sequential counterparts in the absence of concurrency. However, when there is contention, this class of components provides the most fine-grained synchronization fine-tuned by clever implementers.

3.3 Transactional memory

Analogous to database transactions, *Transactional Memory* (TM) (Herlihy and Moss 1993) 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 thread safe algorithm. Some constructions can even turn them into lock-free (Shavit and Touitou 1997) and wait-free (Moir 1997) algorithms without any deadlocks. Despite this theoretical benefit, TM has yet to become widely 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 Software TM (STM) (Herlihy et al. 2003; Saha et al. 2006; Felber et al. 2008) 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. Saha et al. (2006) evaluated all these trade-offs.

Hardware TM (HTM) (Hammond et al. 2004; Ananian et al. 2005) 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 that may or may not be available. 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), which, in turn, causes rollback (abort and restart the transaction) 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 (Damron et al. 2006).

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 (Damron et al. 2006).

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, and it does not matter for the consistency of the component and the correctness 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.

4 Self-adaptive concurrent components

As discussed in the previous section, 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 the application runtime behavior and the runtime environment. The application could be sequential or concurrent and its runtime behavior includes the operations invoked, their read to write ratio to the data representation, the number of threads, and their actual contention. The runtime environment 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 socket system or multiple socket system, the number of hardware threads per core, the fairness of the scheduler, the fairness of the potential locking protocol (unfairness potentially yields higher performance since translation lookaside buffer (TLB) caches are already populated when reacquiring the lock), the availability (and stability) 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 these contexts, our approach is to understand the synchronization mechanism as yet another component variant self-adapted to the actual context dynamically. We refer to components adapting (among others also) their synchronization mechanism to the actual context as *self-adaptive concurrent components*. Based on the changing contention context, we strive to select the best performing component variant including the best synchronization mechanism and the algorithm variant with the lowest time complexity.

The contention attribute is orthogonal to other context attributes, e.g. size of actual parameters and actual operation call sequences. However, contention is regarded the most important dynamically changing context attribute. The rationale for this is that whenever contention builds up on locks, the system experiences a massive slow down due to the concurrency bottleneck that typically by far outweighs all other decisions, similar to how thrashing should be avoided at (almost) all costs. Getting the right fit for a given contention level gets the highest priority. Therefore, a straightforward layered approach is used for the contention (first layer) and the other context attributes (second layer).

A lock-free component variant is used whenever there is a lot of contention (first layer). In practice, there are not many lock-free variants of one and the same lock-free component. However, the number of processors used for storing the data representation instances of the component and for algorithms operating on it can still vary based on the number of processors actually available (second layer) according to Kessler and Löwe (2012).

If there is little contention a component variant protected by locks is used (first layer). If there are different sequential or parallel component variants, then transformations can happen to the (lock-based) component variant that has a presumably superior time complexity (second layer), either instantly with the first operation invoked (Ander-

sson et al. 2008; Kessler and Löwe 2012) or in a deferred way using biasing (Österlund and Löwe 2013).

Of course, a more complex biasing could find tricky corner cases by simultaneously employing contention and other context attributes, like size of data, application profile, time complexity of operations used, hardware, and scheduling, to find more accurate break-even points for what contention level should trigger transformation to a lock-free component variant with potentially worse time complexity. This is less straightforward and potentially costly for the common case as the cost for the logic deciding the presumably best fitting solution increases. It only improves corner cases where manual attention perhaps should be advised anyway for optimal performance. It is therefore deferred to future work.

For the implementation of the first level adaptation of the synchronization variant, we propose a single *contention manager* object for each self-adaptive concurrent component *object*. 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 presumably more appropriate component variant is instantiated. The contention manager receives input from the specific active implementation variants in completely different ways. The contention manager itself is invariant of how it gets its information.

For the contention manager, we need to address three issues. We need to (i) assess the contention context and decide whether changes in the contention should trigger a change of the component variant or not. If so, we need to (ii) invalidate the current component variant and, finally, we need to (iii) transform to the new component variants safely. Obviously, the steps (i)–(iii) need to be performed online and they contribute to the runtime overhead of self-adaptive concurrent components. Hence, efficiency of context assessment and transformation is an important issue. We will address (i)–(iii) in the subsequent three sections.

5 Continuous context feedback (i)

Continuous context feedback for contention, issue (i), is provided by a *contention sensor*. The contention sensor is implemented differently for the three different synchronization mechanisms. It provides a general interface, in essence hooks to code in the contention manager, for 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 was necessary. Conversely, a message indicating that a biased lock was successfully acquired local to one thread would never lead to any change. Hence, such a 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.

5.1 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. (2011) in JikesRVM and Russell and Detlefs (2006) for HotSpot (OpenJDK), 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 locking, 2) lightweight spin lock, 3) heavyweight blocking lock. Each level has its own type of lock that needs to be installed. The ideas apply for any managed runtime, but we will focus specifically on HotSpot, where our algorithms were implemented.

Note that our contention sensor just needs to understand that different locks are installed corresponding to different contention level. In a way, the installation of different locks based on contention is a self-adaptation of the lock component, which is below our general architecture for self-adaptive components done on the level of the runtime environment of our components.

5.1.1 Contention level 1: biased locking

A lock is biased against the first thread that acquired it. This owner thread can acquire and release the lock cheaper with normal loads and stores, avoiding heavier atomic instructions (Russell and Detlefs 2006). The owner thread is established using an atomic CAS operation to install a thread id in the object header. Once the owner has been established, subsequent acquire and release operations are almost for free. It also avoids allocating a lock data structure, by fitting all fields in the object header instead.

Biased locking is a speculative optimization that makes locking fast when used by a single thread. Therefore, it is optimized for very low contention levels. If it turns out that multiple threads need to use the lock, then the bias must be revoked. This is done by stopping the thread holding the bias using the same synchronization mechanism, referred to as safepoint, as used by a garbage collector (GC) to stop threads for GC pauses (all managed threads, referred to as mutators, need to be stopped during certain critical GC operations called GC pauses). Once inside of the safepoint, the lock is reacquired as a lightweight spin lock of level 2.

In detail, for revoking the biased lock of level 1 and promoting the lock to level 2 heuristics keep track of the cost of individual lock bias revocations. When the cost

exceeds a certain threshold, a bulk re-bias operation is performed for all locks on objects of a certain type. This is done by maintaining bias epochs as counters for each type that must match a corresponding counter for each lock on objects of that type. A bulk re-bias operation issues a safepoint, changes the epoch counter of a type, and reclaims all locks currently held. Once the threads start running again, the epochs no longer match between the locks and their corresponding type. This way, the locks that were not held in the safepoint can be re-biased with new owners at subsequent acquire operations. This allows keeping more objects at level 1 and reduces spurious promotions to level 2.

5.1.2 Contention level 2: lightweight spin lock

Once a lock gets promoted to a lightweight spin lock, some contention can be assumed. In HotSpot these locks are implemented by having threads allocate some 8 bit aligned stack space for their lock, loading the object header, writing it to the stack, and installing the stack slot address into the object header using CAS. If the CAS succeeds, then the lock is acquired. Otherwise it failed. This type of stack allocated lock is still lightweight in that it does not need any heavyweight OS lock.

When locking fails, the lock can either spin and wait for a while before trying to lock again, assuming that a small critical section is concurrently executing in another thread that will release the lock soon. The naïve approach would be to just spin in a loop until the lock is released. But that has a number of problems.

Doing spinning right is very difficult and can either speed up a program or make it slower. An optimized spinning implementation in locks goes through the whole stack: from the application layer (disregarded here), over the VM layer and the OS layer, all the way to the hardware layer:

VM	VMs can adaptively change the length of the spinning loop before retrying by learning from history. If it finds that spinning takes too long, it promotes the lock to level 3.
OS	Waiting for a lock to be released by spinning is only a good idea if it is believed that the lock will be released soon. If a thread in a critical section was to be preempted by the OS or blockingly waiting for IO, then suddenly the spinning will waste many cycles in vain. Solaris allows to check cheaply from userspace if the owner of a lock is running on the processor or not. If it is not running, then spinning can be immediately abandoned.
Hardware	Modern processors typically have more than one hardware thread per core to allow sharing execution units like arithmetic logic units (ALU) within the core between different concurrently executing threads, to better exploit thread level parallelism (TLP) when instruction level parallelism (ILP) can not be exploited. On Intel chips it is normal to expect two hardware threads per core, and on SPARC chips 8 hardware threads per core. However, the hardware has no notion of priority once threads have been scheduled to run. So if one thread is in a critical section performing actual work, and a concurrent spinning thread is waiting for acquiring

the lock, then the spinning thread in the same core will use the resources of the core and slow down the one thread performing actual work in the critical section.
To combat this, Intel has a special pause instruction that is similar to a `nop` but also flushes the pipeline with its speculations, uses less power, and lets the other thread in the core (which might be running the critical section) use the core better.
SPARC goes yet another step further and addresses the issue directly. The header word can be loaded with a special load, and then a special `mwait` register is set to wait a certain amount of time, or until the address that was loaded is changed. Meanwhile the hardware is performing very cheap spinning, while boosting the ILP of the other threads, and possibly the threads executing the critical sections. When the lock is released, the spinning threads will wake up and start executing again.

These spinning particularities may motivate performing transformations to new component variants already at this lock contention level, if the environment is not supporting spinning well enough throughout the technology stack. Especially, in environments where the JVM is not the only heavy process running on the same machine contending for cycles. With emerging deployment models where many JVMs are running on the same machine, spinning might be even more destructive than it usually is because of intra JVM interference, and further motivate transforming to a lock-free component earlier.

5.1.3 Contention level 3: heavyweight blocking lock

If bounded spinning on a lightweight lock was not enough, the lock is inflated to a heavyweight lock of level 3 able to block to the OS. A monitor object is allocated and installed into the header word (inflation) using CAS.

The heavyweight lock will also try spinning in a similar fashion as level 2, but once that fails, it will block the thread so that other threads may run instead. It is in particular this blocking event that contention sensors use to signal that locking probably is not the best fitting strategy any longer and that the application should transform to a better one.

The contention sensors need to monitor 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. In practice, we chose to only monitor blocking events. because it occurs when the contention becomes noticeable. Also it makes sure that the rest of the locking machinery has untouched performance having an extra code path when monitoring has insignificant cost as the cost of blocking far exceeds that of the monitoring code.

5.2 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 an idea similar to 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, even lock-free component variants could be transformed back to the locking variant optimized for a single-thread 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.

Alternatively, the lock-free component creates a probe object that is garbage on creation (no references to it). Hence, the probe gets freed, i.e., finalized, by the garbage collector at the next GC cycle. Its finalizer, however, checks the ownership of a lock-free component. If the component indeed was exclusively owned by a single thread, the probe did not sense any contention.

Checking 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 *inaccurate probing* is acceptable. 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 the optimization.

The accurate and inaccurate probing techniques are similar to biased locking; they work well when a single instance is either contended or not. However, when contention changes rapidly, this technique may detect the change to lower contention undesirably late as it needs to wait until the next garbage collection cycle, which could take arbitrarily long. Then it could be beneficial with *active probing*. This scheme calls an introspection probe in the prologue of each operation of the data structure. For a contended data structure, the cost of cache coherence traffic dominates. Therefore, the extra time spent for performing local computations for probing is arguably cheap.

Our active probing mechanism maintains a `ContentionStatus` object in the lock-free data structure. It has a high resolution timestamp describing when it was created, and a pointer to the `owner` thread that created it. These two fields are set at creation, and do not change thereafter. Another field, `last_check`, tracks the when this probe was checked for contention the last time. If a `RESOLUTION` amount of time has not yet passed since the last check, the monitoring routine exits. This is

the common path of the monitoring routine. The RESOLUTION constant is picked so that the overheads of monitoring are kept minimal. We used 1024 nanoseconds in our experiments. The last field, contended, describes whether other threads have observed a ContentionStatus. When MONITORING_INTERVAL time has elapsed, the current ContentionStatus is examined. If it was not found to be contended, an exception is thrown to trigger invalidation and transformation to something more suitable. Otherwise, a new ContentionStatus is installed using CAS. In our experiments, a MONITORING_INTERVAL of 1 ms was used. The procedure is described with pseudo code in Algorithm 2.

```
def active_probe(data_structure)
  status = data_structure.
     contention_status.load();
  if !status.contended && status.owner
     != Thread.currentThread()
    # Checking the current thread is
       cheap; on
    # x86_64, it is stored in the r15
       register in
    # HotSpot. This path is also
       triggered at worst
    # once per thread and
       MONITORING_INTERVAL
    status.contended = true;
  end
  # On x86_64, nanoTime translates into
     a rdtscp
  # instruction. It serializes execution
     and flushes
  # e.g. store buffers. But so do the
     atomic
  # instructions in the lock-free data
     structure to
  # ensure TSO ordering of stores.
     Therefore these
  # costs piggybag on eachother in the
     common case.
  timestamp = System.nanoTime() & ~(
     RESOLUTION - 1);
  check_elapsed = timestamp - status.
     last_check;
  if check_elapsed <= RESOLUTION
    return  # The common case ends here.
  end
  status.last_checked = timestamp;
```

```
22    probe_elapsed = timestamp - status.
          timestamp;
23    if probe_elapsed < MONITORING_INTERVAL
          )
24      return
25    end
26    if !status.contended && check_elapsed
          < MONITORING_INTERVAL)
27      # Uncontended monitor. Trigger
            transformation
28      throw new TransformationException()
29    end
30      # Ongoing contention. Install a new
31      # ContentionStatus object tracking
            the next
32      # monitoring interval.
33      new_status = new ContentionStatus()
34      data_structure.contention_status.CAS
            (status, new_status)
35    end
36  end
```

Algorithm 2 Code for performing active probing on lock-free data structures

5.3 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.

6 Safe component invalidation (ii)

Knowing that a particular component variant is not suitable is not enough; we also need to address issue (ii) and invalidate the current unsuitable component variant safely and efficiently. This *component variant invalidation* invalidates a component variant in the sense that no operation (read or write) may complete or harm the consistency of the component once invalidation has finished. We provide safe component variant invalidation mechanisms separately 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 and, 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. This read-iterator is used in the methods, Representation clone(Representation) copying the state of an invalidated representation to a new representation variant.

6.1 Locks

A general and natural solution to invalidating a lock-based component variant is making lock invalidation a part of the locking protocol. An invalidated lock will make subsequent lock and unlock operations throw an exception taking the execution back to the abstract component wrapper.

Lock invalidation is best done on inflated locks for two reasons. First, we do not care about taking a slow path to the invalidated locks as lock invalidation is rare and happens only when transformation is due. Second, we only invalidate locks due to even higher contention when they have been inflated already.

```
#callback from contention manager
def invalidate_lock(object)
  monitor = get_monitor(object)
  monitor.owner = Thread.current() |
     INVALIDATED
  throw InvalidatedException
end

# slow path if biased locking fails
def slow_lock(object)
  monitor = get_monitor(object)
  if try_spin_lock(monitor)
    # low contention
    return
  end
  # even spin locking fails
  blockingly_lock(monitor)
  # signal high contention to the
     contention manager
  # while still holding the inflated
     blocking lock
  call_contention_manager(object)
end

def blockingly_lock(monitor)
  loop
    # we try to acquire the lock
    prev_owner =
        CAS(0, Thread.current(), &
           monitor.owner)
```

```
27      if prev_owner == 0
28        # success, lock acquired
29        return
30      elsif (prev_owner & INVALIDATED) ==
          INVALIDATED
31        # lock already invalidated
32        throw InvalidatedException
33      else
34        # park and wait for other thread
            to wake us up
35        block(monitor)
36        # then we can retry to acquire the
            lock
37      end
38    end
39  end
```

Algorithm 3 Lock invalidation makes subsequent lock and unlock calls throw an exception.

The pseudo code shown in Algorithm 3 sketches the implementation ideas. When a lock is inflated (line 16) from a spin lock appropriate for contention level 2 to a blocking lock for contention level 3, it can be changed safely, especially invalidated, once the current thread is the owner established with CAS (lines 24–29). After the inflated blocking lock is acquired while holding the lock, we call the contention manager (line 19). The self-adapting concurrent component manager will then trigger transformation and invalidate the lock by calling the invalidation callback (lines 1–6). It changes the owner from its own thread to a sentinel value corresponding to the real owner tagged with a single extra bit that denotes that the monitor has been invalidated (line 4).

After invalidation, the faster JIT-compiled locking paths of other threads may optimistically try biased locking or spinning to get the lock, but they always fail to acquire the lock and ultimately all threads take the slow path because the owner field is never cleared. In that slow path, they will ultimately see that the lock has been invalidated because the owner field is tagged (line 30–32). By now all threads will fail to enter the critical section, and instead compete for taking the transformation lock in the abstract component wrapper to transform to a more suitable component variant.

6.2 Lock-free

For discussing lock-free algorithms, we first need to differentiate linearizing from non-linearizing CAS operations. A linearizing CAS atomically commits the change made by an operation at a linearization point (Herlihy and Wing 1990) 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, non-linearizing CAS operations that update the component lazily and do not hamper the consistency of the component.

Any non-linearizing CAS operation does not need to be visible at all as the consistency of the data structure is not harmed by their succeeding or failing according to the algorithm specification.

Operations of lock-free components are typically *linearizable* and we need to make sure that state changes of succeeding operations are copied to and that failing operations are repeated on the new component variant.

Linearizability is commonly defined in the following way:

Definition 1 All linearizable function calls have a *linearization point* at some instant between their invocation and their response. All functions appear to occur instantly at their linearization point, behaving as specified by the sequential definition.

Sequential consistency (Lamport 1979) ensures that the observable events in program order are observed in the same order on all processors. Linearizability is strictly stronger than sequential consistency. Each operation has a linearization point where the operation logically either completely succeeds or completely fails, hence, making the whole operation logically appear to happen instantaneously. Other processors will never observe partially completed operations from other threads that violate consistency. For writing operations, this linearization point is typically an atomic sequentially consistent CAS operation (cf. Algorithm 1) that atomically either writes a new value, hence, publishes the result of the whole operation, or fails, depending on whether the previous value is as expected, as required for consistency. Similarly, for reading operations, the linearization point is typically a load.

We exploit the linearization points of the lock-free component variants to invalidate them. All the linearization points go through operations on atomic references. We provide our own atomic reference class, that has special load and CAS operations used as linearization points for lock-free data structures for all operations. These special atomic references, can be irreversibly invalidated. Once an atomic reference has been invalidated, any subsequent linearizing operation will throw an exception that takes control back to the abstract component wrapper to handle the fact that the variant has been invalidated, by restarting the whole operation in the new component variant once transformation has finished. We assume lock-free data structures to be composed of object nodes linked together with invalidatable atomic references. Values are captured in value objects linked from (reference) object nodes with atomic references too.

By design, all atomic references transitively reachable from the root (data structure wrapper node) to the leaves (references to the elements) are at all points in time completely encapsulated within the lock-free data structure and are never exposed outside of the data structure. That is, no method on the data structure returns or otherwise exposes any of the internal nodes used by the lock-free component variant outside of itself. This is important for encapsulating the lock-free machinery from the outside world.

Invalidating an atomic reference (cf. Algorithm 4) is itself a lock-free and linearizable operation. A CAS tries to install an `INVALIDATED` sentinel value. This is the linearization point of the operation—it either fails and retries or it succeeds and the reference is invalidated. It continues in a loop until the atomic reference is invalidated in a lock-free fashion.

```
1  def invalidate(atomic_ref)
2    loop
3      val = load(atomic_ref)
4      if val == INVALIDATED
5        return # Already invalidated
6      # CAS is a linearization point,
         atomic_ref is
7      # either successfully invalidated or
         not
8      elsif CAS(val, INVALIDATED,
         atomic_ref) == val
9        atomic_ref.remembered_value = val
10       return
11     end
12   end
13 end
```

Algorithm 4 Invalidating atomic references.

When a reference has been invalidated, the value that was there before the invalidation is remembered and exposed through a special `load_remembered()` API operation (cf. Algorithm 5) that can be used in the clone(Representation) method to read the state of the reference, used only for the purpose of iterating through the elements to construct the new component variant, once the whole data structure has been invalidated.

```
1 def load_remembered(atomic_ref)
2   atomic_ref.remembered_value
3 end
```

Algorithm 5 Special load to get the value after invalidation.

Invalidation means that all subsequent load and linearizing CAS operations (cf. Algorithm 6) on the atomic reference will throw exceptions and fail their operations.

Note that all loads used by the data structure operations, whether linearization points or not, will throw an exception if the atomic reference has been invalidated, because the `INVALIDATED` sentinel value can never be the expected value. Therefore, CAS operations, whether linearizing (explicitly checking for invalidation) or not, will never succeed after invalidation: a CAS can only succeed if the current value is also expected. The `INVALIDATED` sentinel value is an encapsulated secret of the atomic reference class; it is never exposed to the outside world. As a result, it is impossible even for a non-linearizing CAS to ever succeed after invalidation. Therefore, invalidated atomic references are immutable.

The reason we still distinguish between linearizing and non-linearizing CAS is that even though they are similar in that they fail after invalidation, it is semantically subtly different to report a failed CAS and throw an invalidation exception. An algorithm

could have a linearizing CAS in a loop that expects some value that was not loaded from the atomic reference, e.g., sentinel values known by the algorithm or NULL. Such an algorithm would get stuck in an infinite loop after invalidation unless a linearizing CAS is used to explicitly break the loop after invalidation, rather than just reporting failure to CAS. Typically, for CAS operations not used at linearization points, e.g., to lazily update tail pointers in lock-free linked lists, do not even check the success of their CAS; it does not matter if the CAS succeeded or not. Then it is more appropriate to use a non-linearizing CAS to save a few instructions.

```
1  def load(atomic_ref)
2    val = load(atomic_ref)
3    if val == INVALIDATED
4      throw InvalidatedException
5    else
6      return val
7    end
8  end
9
10 def linearizing_cas(expected_val,
     new_val, atomic_ref)
11   val = CAS(expected_val, new_val,
       atomic_ref)
12   if val == INVALIDATED
13     throw InvalidatedException
14   elsif val == expected_val
15     return true
16   else
17     return false
18   end
19 end
```

Algorithm 6 Pseudo code for load and linearizing CAS (atomic write) for atomic references, to be used by lock-free components.

Before transforming a lock-free component into another variant, we need to ensure that all operations performed on the lock-free component variant will be blocked by invalidated references. Therefore, all contained atomic references must be invalidated. The data structures of lock-free components consist of objects (nodes) accessed and connected via atomic references (edges). Invalidation is done by tracing through the lock-free component's data structures and invalidating all reachable atomic references.

6.2.1 Consistency of lock-free invalidation

To guarantee consistency of the lock-free component once variant transformation was decided, we must guarantee that, once tracing (and, hence, invalidation) has been

started, operations either successfully reach a linearization point, or fail and retry in the new component variant when transformation has finished. We must also guarantee that tracing (invalidation) will terminate, and that once it has terminated, it holds that (1) all objects reachable from the data structure root node have been invalidated, and (2) all subsequent operations on the invalidated data structure will fail and instead run the operation on the new component variant once transformation has finished.

If the data structure was immutable or there were no concurrently executing threads (serial tracing), tracing through such a data structure object graph in a consistent way would be easy. Any depth-first search (DFS) or breadth-first search (BFS) algorithm would do.

Guaranteeing consistency of the tracing while concurrent threads are executing is more tricky. Tracing through the data structure to invalidate atomic references concurrently to other threads executing an algorithm works analogously to GC tracing through all live objects concurrently to mutators. In GC, on-the-fly marking advances a wavefront of visited objects. Instead of marking found objects as live to separate them from the garbage objects in GC, they are now marked as invalidated. This guarantees a stable snapshot of the transitive closure of the data structure. Because of the strong similarity to on-the-fly marking, we reuse Dijkstra's tricoloring scheme (Dijkstra et al. 1978) for reasoning about this. It describes a framework of reasoning where nodes can be in one of three states (colors), at any time during tracing.

White nodes have not been visited by the tracing algorithm yet, *gray nodes* have been noted by the tracing algorithm, but their references have not been processed yet, i.e., in our case, their edges haven't been invalidated. *Black nodes* have been visited by the tracing algorithm and their references have been processed and, hence, they don't need to be visited by the tracing algorithm again.

Our tracing and invalidation algorithm can then be described like this: The first node to be traced is the lock-free component variant itself. It is shaded gray and pushed to a stack. In a loop, the stack top node is popped and visited until the stack is empty. A node is visited iff it is not already black, i.e., visited before. For each node visited, all its atomic references are invalidated and, iff the referents are white, they are shaded gray and pushed to the stack (black referents do not need to be visited again and gray referents will eventually be visited as they are on the stack already). Finally, it is shaded black.

A GC tracing algorithm is complete if, upon termination, all reachable objects are eventually visited (shaded black) and, hence, separated from garbage objects (shaded white) that can be collected. This is a challenge if tracing and mutations of the object graph happen concurrently. Note that our tracing for invalidating references of the lock-free data structure also happens concurrently to mutating accesses to this data structure. It was found by Pirinen (1998) that any concurrent tracing GC algorithm that guarantees GC-completeness, enforces one of two invariants on the concurrent execution to guarantee consistency of the tracing:

- The strong tricolor invariant: There is no edge from a black node to a white node.
- The weak tricolor invariant: All white nodes pointed to by a black node are also reachable from a gray node through a chain of white nodes.

Algorithms enforcing the strong tricolor invariant disallow new edges from black nodes to white nodes to guarantee GC-completeness, while the weak tricolor invariant allows new edges from black nodes to gray nodes as long as those white object are somehow reachable later in the tracing.

Our algorithm enforces the strong tricolor invariant. If concurrent threads try to write any edge at all from a black node with the linearizing CAS (cf. Algorithm 6), then it will fail to commit and an invalidation exception would be thrown. Therefore, the scenario where an edge from a black node to a white node is concurrently written, can trivially not happen because black nodes have become immutable.

Our tracing algorithm for invalidating all reachable nodes in a lock-free data structure is complete in the sense that the tracing, once terminated, has shaded all reachable nodes black regardless of concurrent mutation (attempts) to the data structure. This can be shown analogously to already established and since long understood theoretical frameworks for concurrent tracing by a GC. In particular, Dijkstra et al. (1978) proved that their tracing algorithm could guarantee GC-completeness, provided that the strong tricolor invariant is enforced by GC and mutators. Barriers (mutator code run when writing references to the heap, referred to as actions in the paper) for mutators were described that would enforce the invariant, and it was then proven that this invariant on the concurrent execution makes tracing correct and complete. It was later mechanically verified (Hawblitzel and Petrank 2009; Gammie et al. 2015) that barriers of practical GC algorithms really do enforce these tricolor invariants, and then proved by extension that some example tracing algorithms are GC-complete and correct.

For the interested reader, there are formal proofs of invalidation termination, invalidation completeness, and data structure consistency in the Appendix A. These proof are sketched below.

The *termination* of invalidation is guaranteed iff the data structure instance is not concurrently extended endlessly, as reference cycles are broken by not pushing black and gray referents on the stack again. The trick in proving termination is to recognize that the atomic references are completely encapsulated within the lock-free component variant. Therefore, all subsequent operations after the root has been invalidated will fail, and only latent operations that started before invalidation has started need to be dealt with. This is formalized in Section A. I.

The proof of *invalidation completeness* is analogous to the proofs of GC-completeness of the GC tracing algorithms: (1) black nodes cannot change as it requires changing outgoing atomic references that are all invalidated, (2) gray nodes still can change while being on the stack or being visited but, changing competes with attempts to invalidate them in an endless loop and eventually all outgoing references are invalidated and cannot change, (3) a wavefront of gray then black coloring will eventually color all nodes black making the whole data structure immutable. This is formalized in Section A. II.

The *consistency* of the whole data structure instance is guaranteed by the design of the lock-free data structure type. Operations and algorithms are designed to be correct if reads and writes fail at liniarization points; they just don't make progress then. Once invalidation has been initiated, they all will fail eventually. Black nodes are guaranteed to block all subsequent linearization points by concurrent threads. All their operations that have failed, i.e., didn't logically finished atomically, will wait for

the transformation to complete and then eventually retry on a new component variant. This is formalized in Section A. III.

6.2.2 Lock-free invalidation implementation

While the principles of invalidation have been discussed before, we still need to fit them into the object-oriented framework of self-adaptive components. Lock-free components are built using our own IAtomicReference<T> that can be invalidated. It has special get() and load() methods (the implementation principles are described in Algorithm 6) that will throw an exception when invalidated to block linearization points as previously described.

The generic type argument T is required to be a subclass of Object. An internal atomic reference assumes the Object type, and load() casts the internal load from Object to T. A special internal class Invalidated<T> represent invalidated references, and load() 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.

An invalidate() method in IAtomicReference<T> invalidates a reference by installing an Invalidated<T> object containing the original value using CAS in a loop that terminates when CAS succeeds. To invalidate a complete lock-free component variant, it is traversed as described above using invalidate() calls 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.

Like for the other synchronization mechanisms, 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 the special T loadRemembered() method (cf. Algorithm 5) implemented in IAtomicReference<T> used to implement the clone(Representation) methods for changing the representation variants.

6.3 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 itself wrapped in a transaction. If the flag checking operation's transaction failed, it checks if the invalid flag is set. If so it throws an exception like in the locking approach. Unlike the locking invalidation, we unfortunately 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.

7 Safe component transformation (iii)

For all synchronization mechanisms, we can sense inappropriate component variants due to changing contention (i) and safely invalidate them (ii). Finally, this section addresses issue (iii) and discusses how to transform components safely.

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 transformation lock owned by the contention manager. Alternatively, we could employ the lock-free GC copying algorithm by Österlund and Löwe (2015). However, we argue that if contention is so heavy that this lock would become a bottleneck itself, then

1. Choosing to transform away from a locking component variant is probably a good decision anyway. The lock of the component is the actual bottleneck that is eventually removed by the transformation.
2. Choosing to transform away from a lock-free component variant is probably a bad decision in the first place and the contention manager should simply not trigger this transformation.

To safely transform to a new component variant, a special read-only transformation iterator is used in the Representation clone(Representation) methods to iterate over all the elements of the old component variant and add them one-by-one to the new component variant.

The read-only transformation iterator has its own implementation for each component variant, depending on what measures were taken to invalidate the state of the data structure. The read-only transformation iterator must be able to read this invalidated state.

7.1 Lock-based component variants

For the locking, clone may simply use a normal iterator without a lock for the read-only transformation iterator. Apart from the transformation lock, no other lock is necessary to read the state since, the data structure is immutable once invalidated.

7.2 Lock-free component variants

Once a lock-free component variant has been invalidated, any normal access to the invalidated atomic references will throw an exception. Therefore, the read-only transformation iterator walks through the data structure accessing the state using the special loadRemembered() method instead of the normal load (that would throw exceptions), in order to read the invalidated state.

7.3 Transactional memory component variants

Assuming write-buffered TM, the read-only transformation iterators and, hence, the clone methods are implemented by using plain memory read operations.

8 Evaluation

All benchmarks were run on a 2 socket Intel® Xeon® CPU E5-2665 with 2.40 GHz, 16 cores, 32 hyperthreads, 256 KiB L2 cache (per core), 20 MiB L3 cache (shared), 8 x 4 GiB DIMM 1600 MHz, running on Linux kernel version 4.4.0-31. Benchmarks were run on our own modified OpenJDK 9.

By default biased locking would not get activated until too late. 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.
```

Each data series in each chart has executed in a separate JVM to make sure the JIT-compiled code of each data structure is not negatively affected by another data structure. Each data series first starts with ten warmup iterations, followed by ten subsequent recorded iterations.

Each data point in the charts represents the geometric mean of those recorded iterations, and has error bars representing the standard deviation of the recorded iterations. The data points all represent throughput as operations per millisecond, displayed in the primary (left) y-axis. Each chart states how many operations were executed per data point, but the number of threads used to drive the work varies. A thin dashed line represents the level of concurrency and its value is read on the secondary y-axis on the right.

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. Lock-free 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.

8.1 Components and their variants

Three different components were implemented: queues, sets, and ordered sets. OpenJDK already comes with concurrent 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.

8.1.1 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 protected by a lock. 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, accurate or active probing.

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

The lock-free queue is based on (Michael and Scott 1996) 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. They are then lazily removed physically from the linked list.

8.1.2 Set

The second component we analyze implements 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 8 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 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.

Our TransformingHashSet starts as a basic hash table (without tree buckets like HashMap) with a lock. It transforms to a ConcurrentHashMap representation when there is contention.

8.1.3 Ordered set

The last example component implements 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 ConcurrentSkipList is lock-free and based on CAS for linearization points. It basically works like 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 that are then lazily deleted physically.

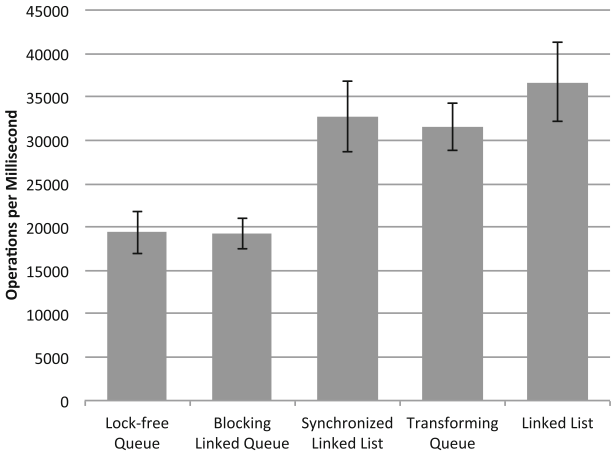

Fig. 2 Throughput for 10,000,000 randomly distributed enqueue/dequeue operations on a single thread. Average size of the data structures is 1000 elements

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

8.2 Case 1: concurrent queues

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

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 that does not have a biased locking scheme, and requires atomic updates to a counter; this explains the poor performance on one thread. The Lock-free queue is based on the queue described by Michael and Scott (1996). Its performance is considerably worse than the single thread optimized double linked list when there is no contention. 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 use of CAS also makes it slower.

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, it performs well on one thread as seen in Fig. 2.

As contention grows with the number of concurrent threads, the Lock-free queue shows the best performance (Fig. 3). The enqueue and dequeue operations that are

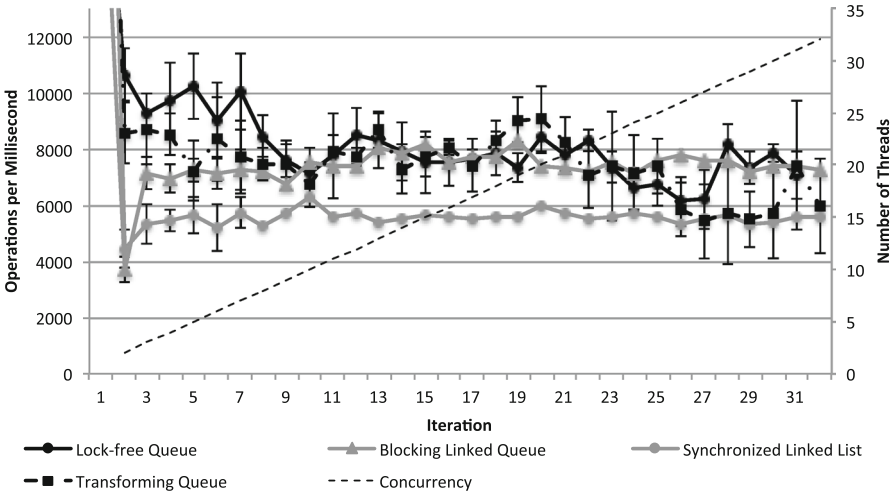

Fig. 3 Throughput for a total of 10,000,000 randomly distributed enqueue/dequeue operations for 1 . . . 32 threads. Average size of the data structures is 1000 elements

$O(1)$ and the synchronization overheads are dominating performance. Locks require two CAS instructions: one to lock and one to unlock. The Lock-free 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 higher throughput when there is contention.

Once again, unsurprisingly, we see that the TransformingQueue follows the Lock-free queue closely. It shows high performance on all levels of contention.

So far, random probing was used as strategy for transforming back to the uncontended component variant. This strategy works well when components turn out to be either contended or not. We quantified the costs of using atomic references that can be invalidated 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, respectively. This is an insignificant cost and will hence not be investigated more in subsequent experiments.

However, as can be seen in Fig. 4, random probing can be suboptimal when contention rapidly oscilates, as a GC cycle may not have passed in time to update the contention level and subsequently transform back to an uncontended component variant.

The first four iterations are uncontended, and the throughput of the TransformingQueue is on par with the best available uncontended component variant. For the next four iterations, there are two threads contending, which triggers a transformation to a lock-free component variant, that continues maintaining throughput on par with the best available contended component variant. However, the next four iterations are uncontended again, but random probing does not catch the change of contention in time. As a result, the throughput is now suboptimal for a while.

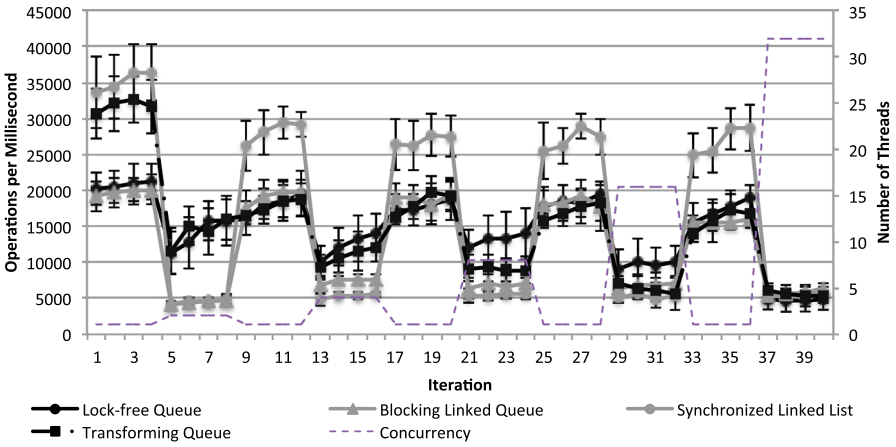

Fig. 4 Throughput for a total of 1,000,000 randomly distributed enqueue/dequeue operations for oscillating levels of contention. Average size of the data structures is 1000 elements

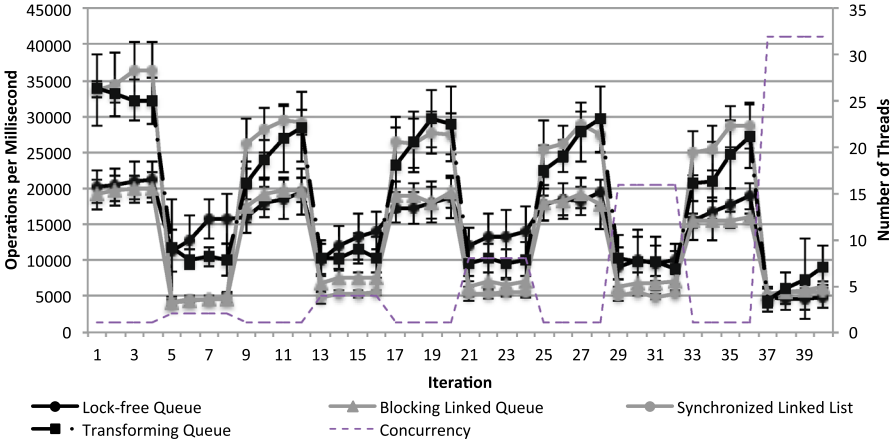

Fig. 5 Throughput for a total of 1,000,000 randomly distributed enqueue/dequeue operations for oscillating levels of contention. Average size of the data structures is 1000 elements

The active probing strategy deals well with this, as can be seen in Fig. 5. Similar to the previous example in Fig. 4, the first four iterations are uncontended, and the performance of our TransformingQueue is on par with the best single threaded variant. For the next four iterations, there is contention between two threads. This triggers a transformation to a lock-free queue which is on par with the best contended queue. And then, for the next four iterations, contention stops again. This is discovered by active probing after approximately one millisecond of there consistently being no contention, and then a transformation is triggered back to the uncontended component variant. Then throughput is restored to similar levels as the best uncontended component variant again. The monitoring costs for the contended component variant were so small they are not noticeable.

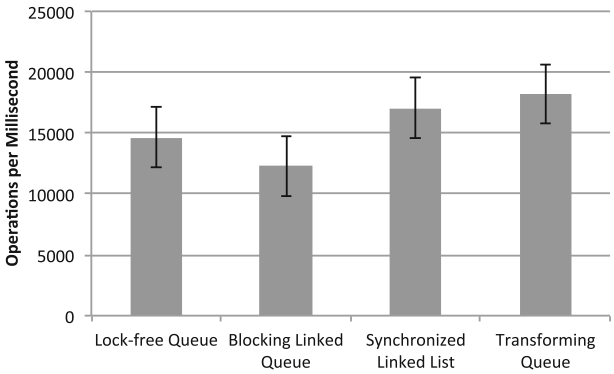

Fig. 6 Average throughput over time with oscilating contention from Fig. 5

In fact, the average throughput over time of the TransformingQueue was higher than for any of the other component variants as documented in Fig. 6. The reason for this is that the other variants demonstrated suboptimal throughput in every other phase, whereas the TransformingQueue maintains a high performance in every phase.

The time spent for transformations to lock-free component variants with active probing was 71 μs on average with a standard deviation of 35. Similarly, the time spent for transformations back into the lock-based component variants was 421 μs on average with a standard deviation of 89. This transformation cost is cheap considering there are tens of thousands of operations executed per millisecond.

8.3 Case 2: concurrent sets

We measured the results of running a benchmark with the operations add (33%), remove (33%) and contains (33%), randomly distributed on a number of threads. The results of the single threaded version context are presented in Fig. 7 and the results of a multi threaded version, with linearly increasing levels of contention, are presented in Fig. 8.

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 a simple hash map wrapped in synchronized blocks, but without the logic for "treefying" buckets, explaining its slightly better performance. 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 (Fig. 7) the observed throughput differences are not large. This is understandable 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 larger memory overheads. The TransformingHashSet does not turn large buckets into trees and does not indirectly use a HashMap, and is hence slightly lower in performance compared to HashSet but outperforms the ConcurrentHashMap on average.

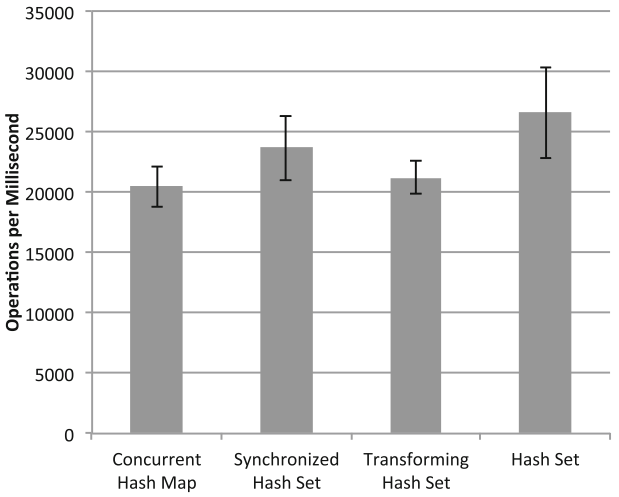

Fig. 7 Throughput for 10,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations on a single thread. Average size of the data structures is 1000 elements

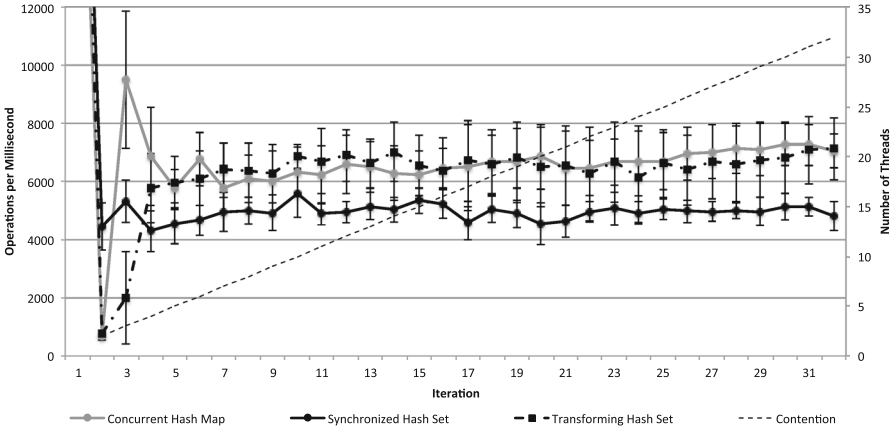

Fig. 8 Throughput for 10,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations for 1 . . . 32 threads. Average size of the data structures is 1000 elements

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 33% of the operations are contains(), which rarely need 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.

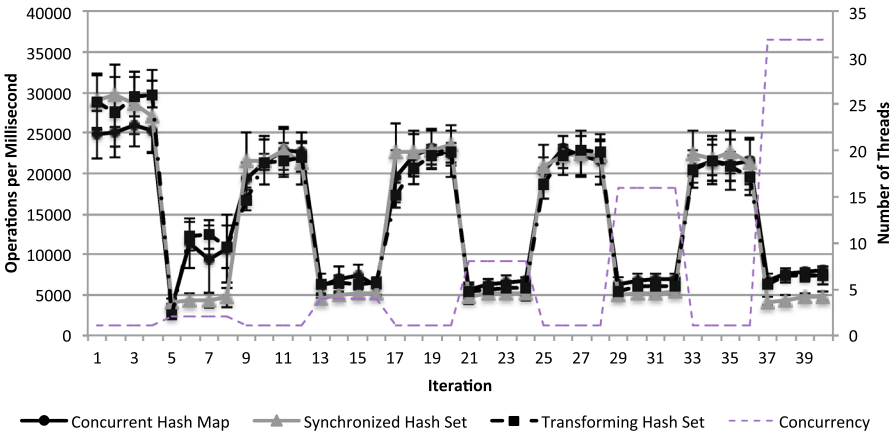

Fig. 9 Throughput for 1,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations on oscillating contention levels. Average size of the data structures is 1000 elements

Even more obvious than in the previous experiment withs queues, the transforming set variant once again follows closely the implementation optimized for concurrency. Again, performs well regardless of the contention, while in this case also keeping lower memory footprints when there is no contention.

So far, the set experiments used increasing contention and random probing. The results of running oscillating contention with random probing can be seen in Fig. 9.

The first four iterations are uncontended, and throughput is on par with the best available uncontended component variant. For the next four iterations, there are two threads contending, which triggers a transformation to a lock-free component variant, which still maintains throughput on par with the best available contended component variant. The next four iterations are uncontended again, but random probing can not catch the change of contention in time. Fortunately, the ConcurrentHashMap still seems to be performant.

However, for the sake of completeness, the active probing strategy can be seen in Fig. 10. It shows very similar performance characteristics as random probing and seems to be doing consistently well in all contention contexts.

Note also that the transforming hash set recovers the improved uncontended performance better than the synchronized hash set after contended periods. The reason is that the biased lock of the synchronized hash set remains tainted when contention vanishes. The transforming hash set on the other hand transforms to a new synchronized hash set with a new lock that successfully becomes biased.

Finally, the average throughput over time of the transforming hash set was higher than for the two other component variants as shown in Fig. 11 as it performs well in low and high contention phases.

The time spent for transformations to lock-free component variants was 203 µs on average with a standard deviation of 131. Similarly, the time spent for transformations back to lock-based component variants was 82 µs on average with a standard deviation

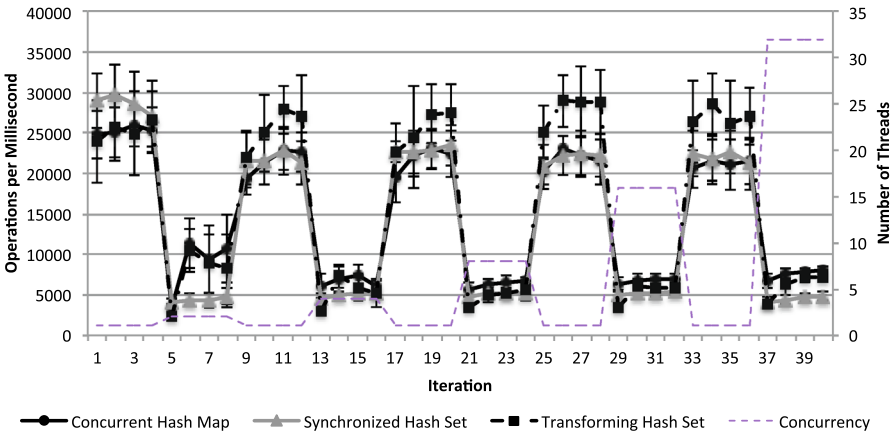

Fig. 10 Throughput for 1,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations on oscillating contention levels. Average size of the data structures is 1000 elements

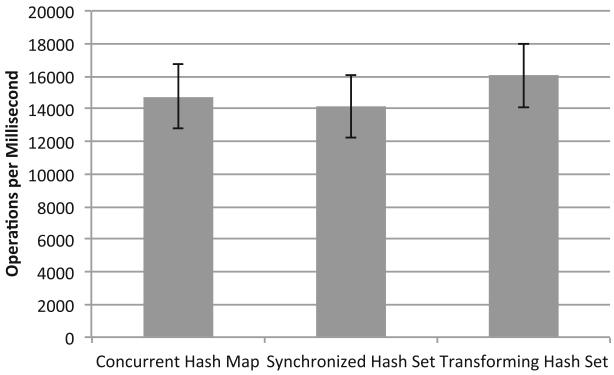

Fig. 11 Average throughput over time with oscilating contention from Fig. 10

of 35. This is cheap considering there are tens of thousands of operations executed per millisecond.

8.4 Case 3: concurrent ordered sets

The same benchmark as for the normal sets was run and results are shown in Fig. 12 for the uncontended and in Fig. 13 for the contended case.

For the uncontended benchmark (Fig. 12), 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

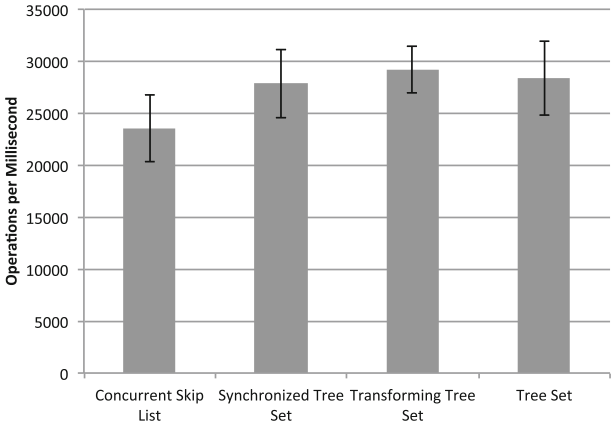

Fig. 12 Throughput for 10,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations on a single thread

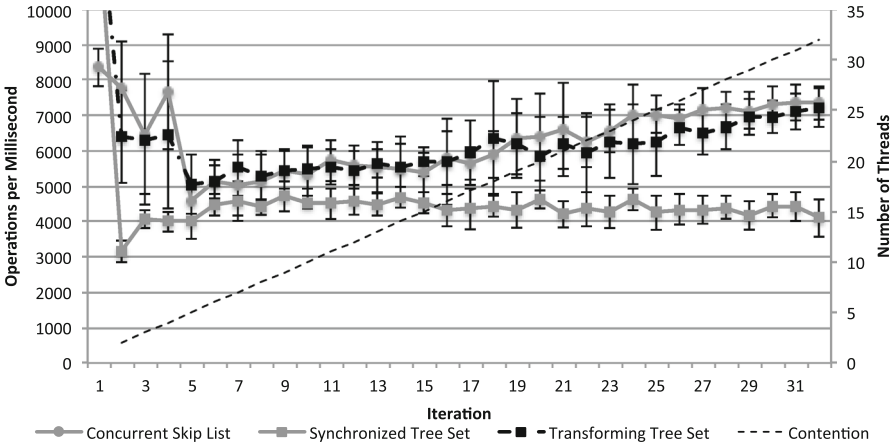

Fig. 13 Throughput for 10,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations for 1 . . . 32 threads

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. This also explains the large error bar for the concurrent skip list.

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 (Fig. 13) 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 performance of the transforming variant consistently follows the performance of the concurrent skip list achieved by transforming to a concurrent

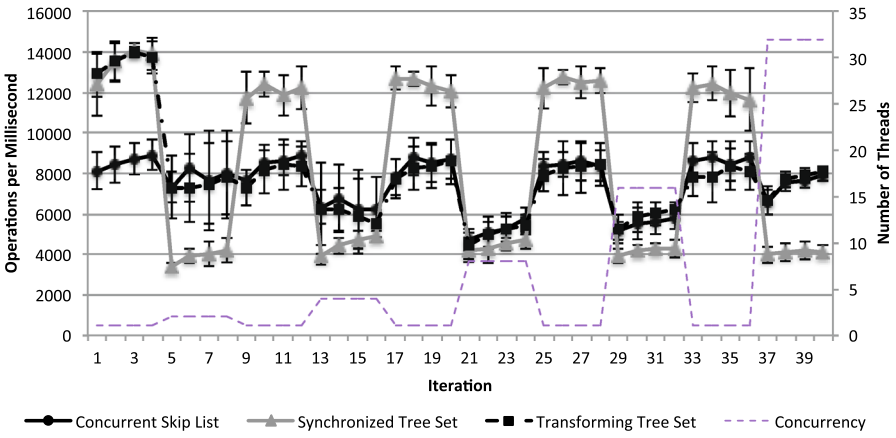

Fig. 14 Throughput for 1,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations on oscilating contention levels. Average size of the data structures is 1000 elements

skip list when the lock becomes inflated and the system identifies the concurrency bottleneck.

These ordered set experiments used increasing contention and random probing. The results of running these components under oscillating contention with random probing can be seen in Fig. 14.

The first four iterations are uncontended, and the transforming variant performs on par with the best available variant for the respective contention contexts. For the next four iterations, there are two threads contending, which triggers a transformation to a lock-free component variant, that still maintains throughput on par with the best contended component variant available. The next four iterations are uncontended again, but random probing can not catch the change of contention in time. The throughput becomes suboptimal.

The active probing strategy can be seen in Fig. 15, and it remedies the situation a little bit. It is doing reasonably well in all contention contexts.

Again, the average throughput over time of the transforming tree set was higher than for the other component variants, cf. Fig. 11 achieved by being competitive in all phases with their different contention levels (Fig. 16).

The time spent for transformations to lock-free component variants was 154 µs on average with a standard deviation of 79. Similarly, the time spent for transformations back into the lock-based component variants was 103 µs on average with a standard deviation of 37. This transformation cost is cheap considering there are tens of thousands of operations executed per millisecond.

9 Related work

While general components are reusable in many contexts, the performance costs of this generality are sometimes unacceptable. Hence, specializing the components to specific usage contexts is desirable. Component specialization and the selection of or

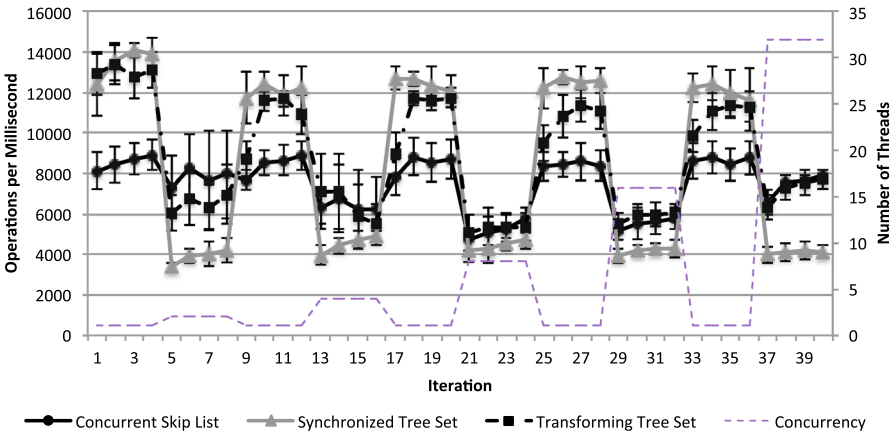

Fig. 15 Throughput for 1,000,000 randomly distributed add (33%), remove (33%) and contains (33%) operations on oscillating contention levels. Average size of the data structures is 1000 elements

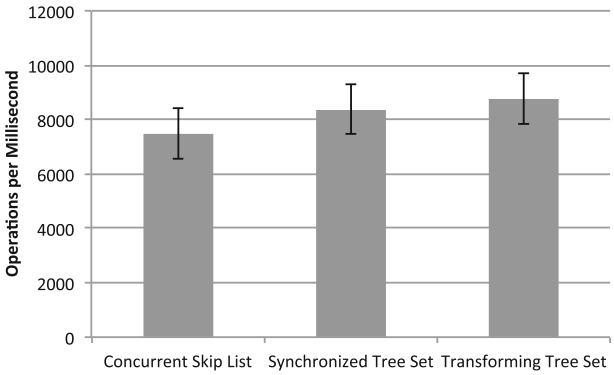

Fig. 16 Average throughput over time with oscilating contention from Fig. 15

the transformation to the best-fitting special variant has been a great concern in the composition community for many years.

9.1 Variant specialization and adaptation in sequential contexts

The semantics of certain high level languages don't allow programmers to be precise in their choice of data structures. JavaScript, for instance, has an associative array used as both an array with indices and as an associative collection with key/value pairs. These more abstract data structures implemented as part of programming languages also benefit from component sepcialization and a dynamic selection of the internal implementation depending on how it is used (Schonberg et al. 1979).

For object-oriented languages, Schultz et al. (1999) propose different approaches to automatically specialize applications using advices from the developers. Löwe et al. (1999) assume that the programmer will select specialized algorithms and data repre-

sentation variants dynamically. Their design of such a data structure uses bridge and strategy design patterns for exchanging the representation implementation and the algorithms, repectively. However, the specialization selection operations are assumed to be public at the data structures' interface and specializations are triggered explicitly by calls form application code.

Svahnberg et al. (2005) present a taxonomy of techniques for variability realization and specialization. Within this framework, self-adaptive (concurrent) components constitutes a variability realization technique for variant binding during runtime.

Self-adaptive (concurrent) components may be considered as a generalization of the dispatch mechanism in object-oriented languages. Context Oriented Programming (COP) (von Löwis et al. 2007) offers generic, language level, mechanisms suitable for implementing context-aware optimizations at runtime.

Autotuning in domain-specific library generators achieves adaptive optimizations. Profiling data gathered during off-line training is used to tune key parameters, such as loop blocking factors to adapt to, e.g., cache sizes. Examples include ATLAS (Whaley et al. 2001) for linear algebra computations, SPIRAL (Moura et al. 2000) and FFTW (Frigo and Johnson 2005) for Fast Fourier Transforms (FFT) and signal processing computations.

Li et al. (2004) implement a library generator using dynamic tuning to adapt to the target machine. A number of context attributes such as the size and the distribution of the input and hardware environment properties such as cache size are input to a machine-learning algorithm. The resulting dispatcher is able to select the most appropriate algorithm for a context. In contrast to our approach, auto-tuning is domain specific and data representations are not changed dynamically.

There are several approaches to automatic optimization of algorithm selection, resource allocation, or scheduling at runtime aiming at parallel target machines. Again, these are often for specific domains, and dynamic selection of the data representation is not considered. We refer to (Kessler and Löwe 2012) for a detailed discussion of approaches for parallel machines.

Context-aware composition (Andersson et al. 2008; Kessler and Löwe 2012) is the principle approach used for self-adaptive (concurrent) components. Using online or offline profiling and different machine-learning approaches, the implementation of operations as well as data structures could be adapted to meet different usage contexts. Österlund and Löwe (2013) extended context-aware composition by regarding call sequences of operations as context attributes. Not always optimizing individual operation calls avoids oscillating transformations between variants that create more costs than benefits. Therefore, this approach can also decide for changing to a variant, which only amortizes over a number of operations.

Sound and adaptive replacement of Java collections data structures was introduced by Xu (2013). Their concept is closely related to context-aware composition: offline and online machine learning approaches find the best implementation variants in different contexts and transform to these variants at runtime. Xu (2013) focuses on the automated generation of code for transforming data structures while Österlund and Löwe (2013) focus more on avoiding oscillation.

9.2 Variant specialization and adaptation in concurrent contexts

Dig et al. (2009) presented a static analysis that could refactor sequential into thread safe data structures and Kjolstad et al. (2011) presented a similar analysis for converting mutable objects to immutable objects.

A number of approaches to STM (Herlihy and Moss 1993; Saha et al. 2006; Herlihy et al. 2003; Dice et al. 2006) were attempted as universal static transformations for turning any sequential algorithm into an algorithm running in a concurrent context. They have yet to be adopted practice. Similar ideas were presented for HTM (Ananian et al. 2005; Hammond et al. 2004) and hybrid TM (Damron et al. 2006). 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.

All these static approaches are orthogonal and complementary to our approach that dynamically transforms *instances* of these functionally equivalent data structures with different synchronisation behaviour.

Some researchers argue that the optimal approach is to build a *manually* fine tuned library of reusable concurrent components without locks. Many such lock-free data structures have been implemented (Michael and Scott 1996; Herlihy et al. 2008; Fomitchev and Ruppert 2004; Michael 2002). With the publication of methodologies for constructing wait-free data structures (Kogan and Petrank 2012) it became straight forward to also make previously lock-free components wait-free (Kogan and Petrank 2011; Timnat et al. 2012). However, their performance is suboptimal, constants may be higher, and the progress guarantees depend on the progress of the underlying memory management. Our concurrent queues used in the evaluation build on the work of Michael and Scott (1996), Kogan and Petrank (2011). While their data structures have good concurrent performance, they are suboptimal in less contended contexts.

Felber et al. (2008) 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. (2011) 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 component implementation variants exploiting synchronization mechanisms.

The idea of self-adaptive concurrent components was first introduced by Österlund and Löwe (2014). The present paper puts it into the context of general self-adaptive components and adds architecture, design, and implementation details and gives evidence for transformation consistency.

10 Conclusions and future work

Self-adaptive components relieve programmers from deciding which component variant is fitting best in the actual usage context. Context includes the hardware and OS environment, the number of unused cores/processors, the thread contention of the component operations, the sequences of component operation calls, the size and distribution of operation parameters etc. Component variants include different data structure

representations, algorithms, synchronization mechanisms, processor allocations and schedules etc.

Self-adaptive components select a presumably best fitting variant automatically behind the visible interface of the component. This way they can even outperform carefully manually selected component variants, as it is often impossible to know the execution contexts at compile time and which component variant becomes optimal at runtime. The execution context and, hence, optimal component may even change throughout the execution of a program such that any static decision gets suboptimal.

In the present paper, we focus on the self-adaptive *concurrent* components and on the thread contention context determining the respectively optimal component synchronization mechanism variant. We introduced a framework for building such components. It bridges the gap between the best performing component variants in a sequential context and the best performing component variants in a concurrent context by automatically transforming between them dynamically and transparently for the application. These component variants can be implemented using completely different synchronization mechanisms, e.g., lock-based, lock-free or TM. Contention managers sense contention changing at runtime and 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 eases.

The approach was evaluated using transforming queues, sets and ordered sets exposed to different contention levels. The transforming components show performance on par with the best solutions for each contention context, and even better performance than all alternative variants when the contention levels oscillate.

The main merit of self-adaptive *concurrent* components 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 (almost) 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* perform.

Open issues addressed by future work are discussed below.

Deferred transformation using GC The cost of transforming from one component variant to another is typically $\Omega(n)$, which is expensive. If an operation with the current variant is $O(n)$ and the operation in the potential target variant is $O(1)$, then it may be suitable to perform the transformation immediately. When the difference between the operations on different variants is just a constant, eager transformation could be counter-productive. Instead transformation should amortize over a sequence or concurrent calls of several operations and, hence, be deferred if such a sequence/-contention can be expected in the future. Currently, we use observed sequences and the actual contention to heuristically predict the future usage context.

Knowing when it is worth to transform is crucial and future work will improve deferred transformation: we could flag that a component benefits from changing its variant but, not perform the transformation immediately. Instead, it could be done when it is more suitable, piggybacking on a copying garbage collector. Since it has to go through all objects, copying them from from-space to to-space, it would be possible to let the transformation happen during this transition. A pure copying GC performs an

identity transformation that replicates the object graph as it is, without changing it. It could as well perform a semantic identity transformation of object sub-graphs that are representations of self-adaptive components. This way, transformation would become cheap as part of garbage collection happening anyway, and remove the problem of oscillating transformations completely.

A GC could also help with keeping the constant time required by the self-adaptive component framework down. More specifically, there is a need for a level of indirection between the component façade and the implementation variant and dynamic binding/dispatch to get to the actual implementation variant. This costs was measured as a 10% overhead in the worst case of an add in an array implementation variant of an ArrayList component (Österlund and Löwe 2013). This is a high cost for performance portability, especially, if the programmer already knows this is what's optimal in special cases. A GC could forward pointers to the current component versions, i.e., letting references point directly to the actual representation as part of garbage collection and, hence, reduce even these indirection and dispatch costs.

Adding off-line intelligence to the on-line dispatcher The current solution optimizes based on contention (and other context attributes) that are observed on-line. However, sometimes static analysis can quite accurately predict contention (and other context attributes) if not over the complete run of a program then at least over program phases or as long as certain conditions hold. Merging static conservative but inaccurate analyses and with dynamic optimistic but accurate assessment could allow to automatically move some of the online decisions to more efficient offline decisions without loosing accuracy.

Experimental evaluation Measurements show a 5.19% increase in performance (on average) on the DaCapo multi processor benchmark suite when just replacing a single data structure ArrayList with a self-adaptive *locking* component (Österlund and Löwe 2013). DaCapo is a standard benchmark; for the applications contained, data structure implementations were probably statically selected with care but they cannot adapt to dynamically changing usage contexts. More work needs to be invested in evaluation:

- It would be interesting to combine the sequential ArrayList variants using locks with a lock-free variant and evaluate the overall performance gain when the self-adaptive concurrent component varies over algorithm, data structure and synchronization mechanism. Also, the combination of contention and other context attributes, e.g., available cores/processors, should be evaluated more in detail.
- In general, more self-adaptive concurrent components need to be implemented and performance evaluated in realistic benchmarks and real world application contexts.

Other optimization goals The current solution looks mainly at one single optimization goal, the runtime execution time. Maybe there could be other conflicting goals to optimize for, such as memory usage and energy consumption. Future work could describe how to manage these potentially conflicting goals.

Acknowledgements We would also like to thank the anonymous reviewers of ASE'13 and ASE'14 for their insightful feedback that helped to improve this paper.

Appendix A: Lock-free invalidation correctness proof

A. I Invalidation termination

To prove *termination* of invalidation, we require three lemmata:

Lemma 1 *Every node visited by the tracing is unique.*

Proof Reference cycles are broken by not pushing black referents on the stack again. That is, visited (black) nodes are not visited again. Multiple pointers to the same referent are not followed by not pushing gray referents on the stack again. That is, (gray) nodes are only occurring once on the stack. □

Lemma 2 *There is a finite set of nodes to be visited.*

Proof At the point when the root node has been successfully invalidated, referred to as *root invalidation time*, there may be at most t concurrently executing operations to mutate the shape of the data structure, where t is the number of threads operating on the data structure. At root invalidation time, let the snapshot of reachable nodes (with propagation paths from the root) be the set of nodes N. Let the nodes created by the t potentially concurrently executing threads be the set of nodes L. The L nodes may be committed by creating edges from gray and white nodes that the concurrent invalidation has not reached yet.

All operations started subsequent to root invalidation time, will always fail, because all atomic references linking the data structure together are completely encapsulated in the lock-free component variant. Therefore the only way of performing the operation is to go through the root node and its atomic references, that have been invalidated. Therefore, only the finite L nodes due to latent operations may be added after root invalidation time.

During concurrent tracing, the latent operations that had already started at root invalidation time, could either be removing nodes in N with white or gray predecessor so that they are no longer transitively reachable, or adding nodes from L to gray or white nodes in the data structure so that they become reachable. This is a race between the latent operations of the t mutating threads and the tracer. If the mutators are faster than the tracer, they have the potential to change a few edges, but that does not block the tracer. Conversely, if the tracer is faster at coloring nodes black, then the latent mutator operations get aborted.

Therefore, in the worst case, the nodes to be visited by the invalidation is at most $N \cup L$, which is finite. Therefore, the data structure can not be extended endlessly concurrent to invalidation. □

Lemma 3 *The tracing progresses for each visited node.*

Proof At root invalidation time, the set of black (visited) nodes, V, is a set of size 1 containing only the root node r, and then the set of unvisited nodes, U is in the worst case $N \cup L \setminus \{r\}$. U will contain gray nodes immediately reachable from V and white nodes. Each time a unique gray node from U is visited, it is moved from U to V. Therefore, U monotonically decreases for each visit, until either U is empty or contains only unreachable (white) objects. □

Theorem 1 *Invalidation of a component variant will terminate.*

Proof Since it was proven that invalidation visits unique (cf. 1) and finitely many (cf. Lemma 2) nodes, and the tracing progresses through those nodes (cf. Lemma 3), it follows that the invalidation will terminate when there are eventually no more nodes to visit. □

A. II Invalidation completeness

To prove *completeness* of invalidation, we require three more lemmata:

Lemma 4 *No edge from a black node may ever change.*

Proof References out of black objects are invalidated and therefore immutable. Any CAS invocation, linearizing or not, needs to expect the INVALIDATED sentinel value for its CAS to succeed, which is impossible; the INVALIDATED sentinel value is privately encapsulated in the atomic reference class. □

This invariant is strictly stronger than then the *strong tricoloring invariant*, cf. Lemma 5, which is sufficient to prove completeness. However, Lemma 4 makes the proof easier and easier to follow.

Lemma 5 [Strong tricoloring invariant] *There is no edge from a black node to a white node.*

Proof A node is colored black (and moved to V) once all immediately reachable nodes in U have been shaded gray. At that point, the edges may refer to either gray or black nodes. Mutating threads cannot subsequently add new (white) referents to black nodes as they are invalidated and therefore immutable. □

Lemma 6 *Every black node is part of the transitive closure C of the root r.*

Proof It can be proven by induction that there exists a path of predecessors from every black node that goes all the way back to the root r.

Base case At the root invalidation point, the set of black (visited) nodes, V, consists of only r. The root r is by itself part of the transitive closure of R by definition. Also all successor nodes of r in U are gray. Once a node has been colored black (and moved to V), its out edges will not change (cf. Lemma 4), and therefore those gray objects will permanently have r as a predecessor.

General case A gray node $u \in U$ being visited at a point in time has at least one black predecessor node $v \in V$. All immediately reachable successor nodes from u in U are shaded gray during that visit, then u is shaded black and moved to V. Therefore, every such visited node u added to V was at some point (when colored black) directly reachable from a predecessor black node $v \in V$. Because of Lemma 4, that predecessor will remain the same throughout the tracing and u will remain to be the successor of v throughout the tracing, too. Therefore, Lemma 6 is proven by induction. □

Theorem 2 *Upon termination of invalidation, the whole data structure is invalidated.*

Proof Tracing terminates when U is either empty or contains only white nodes. Because of Lemma 5, there are no edges from V (only black nodes) to any node in U. Therefore, upon termination, the nodes in U are not reachable from any of the nodes in V. Together with Lemma 6, the remaining nodes in V are the transitive closure C of r, i.e., $C = V$. Since all nodes in V are black, hence, invalidated, it holds that all reachable nodes are invalidated and, hence, the whole data structure is invalidated. □

A. III Component consistency

The *consistency* of the whole data structure instance is guaranteed by the design of the linearizable lock-free data structure types. Operations and algorithms are designed to be consistent if reads and writes fail at liniarization points; they just do not make progress then, like a failed transaction. Linearizability is a consistency property that is up to implementors of lock-free components to prove. Once proven, consistency of the data structure is guaranteed. Either operations succeed, or *fail safely*. By safely we mean that they do not have side effects that compromise the consistency of the data structure. Any operation that fails safely, will wait for the component to transform into a new component variant, and then restart the operation there.

Lemma 7 *In a concurrently invalidating component variant, all subsequent operations after root invalidation time will fail safely.*

Proof All subsequent linearizable operations that start after root invalidation time will need to access the data structure root edges to perform any meaningful operation on the data structure, as the linearization points are all defined in the atomic references. The root node is black, and therefore its successor edges will all throw exceptions once accessed, causing any such operation to immediately fail safely. □

Lemma 8 *In a concurrently invalidating component variant, all latent operations that started before root invalidation time will either fail or succeed safely.*

Proof There could be up to t latent operations that had already started at root invalidation time, where t is the number of threads concurrently accessing the data structure. Such operations could consist of either:

- Reading elements (following edges)
- Adding edges to new nodes
- Removing edges to old nodes

In all of those scenarios, the operations could at any point before the linearization point come across an invalidated reference, in which case the operation is invalidated. But since a linearization point had not been reached, this fails safely by definition.

However, any such operations could also make it to the linearization point. The linearization may succeed if performed on edges of white or gray nodes (ahead of

invalidation), in which case the operation is made completely observable at that linearization point (according to definition).

The operation could fail at an invalidation point either due to conflicts with other latent operations, or due to conflicts with invalidation. If it is a conflict with other latent operations, then the operation will be retried (without violating any consistency due to linearizability). When linearization points fail due to accessing invalidated references (the invalidation tracer caught up), the operation will fail safely (without retrying in the old component variant) due to the exception thrown, effectively blocking the linearization point. □

Lemma 9 *In a concurrently invalidating component variant, any latent operation that succeeds at a linearization point in the source component variant c_1, will have its successful operation reflected in the target component variant c_2.*

Proof A successful linearization point during concurrent tracing can only be reached when the linearization point is reading or writing edges of a white or gray node. That is, it is ahead of the invalidation tracer and operating on nodes the tracer has not yet reached. Tracing will eventually reach all transitively reachable objects (cf. Theorem 2), and therefore the successful operations will eventually have their operations reflected in the invalidated data structure and the corresponding state changes gets transformed to c_2. □

Lemma 10 *In a concurrently invalidated component variant, all latent operations that started after invalidation terminated, will fail safely.*

Proof After invalidation has terminated, the transitive closure of the root r is invalidated (cf. Theorem 2). Therefore, all atomic references transitively reachable from r are invalidated, and any subsequent linearization point will fail safely. □

Theorem 3 *All operations on a component transforming from component variant c_1 to c_2 will finish in c_1 before invalidation and in c_2 after invalidation.*

Proof We consider three cases: (i) the invalidation has not yet started, (ii) it has started but not terminated, and (iii) the invalidation has finished.

Case (i): all operations will finish in c_1 because no atomic references in the component variant have been invalidated and hence will work as normal.

Case (ii): Lemma 10 proved that the remaining latent operations that started before invalidation finished, but had not yet finished after invalidation finished, will fail safely, and restart their operation in c_2. Transformation eventually shifts all operations to the new component variant.

Case (iii): Lemma 7 proved that all subsequent operations started after invalidation finished will, fail safely and start operating on c_2. □

Theorem 4 *Operations on components undergoing transformation preserve consistency such that all operations, invariantly of the stage of the transformation, have a total ordering that is consistently observed across all threads and homomorphic to a sequential execution with the same component-specified semantics preserved.*

Proof We distinguish the cases of Theorem again.

Case (i): all operations work as usual because no atomic reference has been invalidated yet. Their operations finish in linearization points with a total ordering that can be projected to consistent sequential executions, as required by linearizability that the lock-free component variants guarantee.

Case (ii a): all operations started *after* root invalidation time will fail safely and restart their operations in the target component variant c_2 (cf. Lemma 7). When an operation fails safely, it did not commit its operation at its linearization point, and, hence, linearizability guarantees that it has not left any traces of half-completed operations that compromises the consistency of the component, i.e., it restarts in the target component variant c_2 once transformation has finished, without leaving any trace in c_1. All potential target components c_2 guarantee consistency, i.e., we only transform between component variants that are thread safe on their own, disregarding transformation. Therefore, operations that fail safely in c_1 and are restarted in c_2 have a total order homomorphic to consistent sequential execution with corresponding component-specified semantics.

Case (ii b): latent operations were started *before* root invalidation time but did not yet finish. It was proven in Lemma 7 that those operations either complete successfully in the source component variant c_1 or fail safely and restart in c_2 after transformation. As already argued, operations that fail safely and restart in c_2 after transformation guarantee the desired consistency. Lemma 9 proved that those operations that succeed in c_1 will have their side effects observable in c_2 after transformation. The latent operations may compete with each other as usual, and get a total ordering in the linearization points. Once an operation succeeds at a linearization point, it is ahead of the invalidation wavefront. Once the invalidation tracing catches up, it will be as if those latent operations that succeeded had already succeeded at root invalidation time, and any potential state change due to such successful latent operations will be reflected in the invalidated component variant once the invalidation is completed. This invalidated component variant c_1 is then transformed to be semantically equivalent in c_2, and, hence, the effect of those successfully completed latent operations in c_1 have successfully transferred to c_2 and maintained the same total ordering that they had in c_1. Again, this total ordering is homomorhic to a consistent sequential execution with intended data structure specified semantics, because it has the same cloned state as the valid execution in c_1 that committed at linearization points that according to linearizability guarantee those properties. It is as if invalidation of the whole data structure happened instantaneously right after the last latent operation succeeded at its linearization point.

Case (iii): all new operations that start after invalidation terminated, or latent operations that did not finish before invalidation finished, will retry in the target component variant c_2 (cf. Theorem 4) once transformation has finished. The total ordering of those operations can, hence, be guaranteed if such guarantees hold in c_2 and, again, we only use component variants that can guarantee total ordering. Therefore, those operations will have such a total ordering and be homomorphic to a consistent sequential execution with corresponding component-specified semantics. □

References

Abbas, N., Andersson, J., Löwe, W.: Autonomic software product lines (ASPL). In: Proceedings of the Fourth European Conference on Software Architecture: Companion Volume, ACM, New York, NY, USA, ECSA '10, pp. 324–331 (2010). doi:10.1145/1842752.1842812

Amdahl, G.M.: Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, Spring Joint Computer Conference, ACM, AFIPS '67 (Spring), pp. 483–485 (1967). doi:10.1145/1465482.1465560

Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on, IEEE, pp. 316–327 (2005)

Andersson, J., Ericsson, M., Keßler, C.W., Löwe, W.: Profile-guided composition. In: Software Composition, pp. 157–164 (2008)

Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ACM, New York, NY, USA, ASPLOS XII, pp. 336–346 (2006). doi:10.1145/1168857.1168900

Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: Distributed Computing, Springer, pp. 194–208 (2006)

Dig, D., Marrero, J., Ernst, M.D.: Refactoring sequential java code for concurrency via concurrent libraries. In: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, pp. 397–407 (2009)

Dijkstra, E.W., Lamport, L., Martin, A.J., Scholten, C.S., Steffens, E.F.M.: On-the-fly garbage collection: an exercise in cooperation. Commun. ACM **21**(11), 966–975 (1978). doi:10.1145/359642.359655

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

Fomitchev, M., Ruppert, E.: Lock-free linked lists and skip lists. In: Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Distributed Computing, ACM, pp. 50–59 (2004)

Frigo, M., Johnson, S.G.: The design and implementation of FFTW3. Proc. IEEE **93**(2), 216–231 (2005). Special issue "Program Generation, Optimization, and Platform Adaptation"

Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns—Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)

Gammie, P., Hosking, A.L., Engelhardt, K.: Relaxing safely: verified on-the-fly garbage collection for x86-TSO. SIGPLAN Not. **50**(6), 99–109 (2015). doi:10.1145/2813885.2738006

Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. In: ACM SIGARCH Computer Architecture News, IEEE Computer Society, vol. **32**, pp. 102 (2004)

Hawblitzel, C., Petrank, E.: Automated verification of practical garbage collectors. SIGPLAN Not. **44**(1), 441–453 (2009). doi:10.1145/1594834.1480935

Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures, vol. **21**. ACM (1993)

Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. TOPLAS **12**(3), 463–492 (1990)

Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the Twenty-Second Annual Symposium on Principles of Distributed Computing, ACM, pp. 92–101 (2003)

Herlihy, M., Shavit, N., Tzafrir, M.: Hopscotch hashing. In: Distributed Computing, Springer, pp. 350–364 (2008)

Kessler, C., Löwe, W.: Optimized composition of performance-aware parallel components. Concurr. Comput. Pract. Exp. **24**(5), 481–498 (2012). doi:10.1002/cpe.1844

Kirchner, J., Heberle, A., Löwe, W.: Evaluation of the employment of machine learning approaches and strategies for service recommendation. In: Service Oriented and Cloud Computing—4th European Conference, ESOCC 2015, Taormina, Italy, Sept 15–17, 2015. Proceedings, pp. 95–109 (2015a)

Kirchner, J., Heberle, A., Löwe, W.: Service recommendation using machine learning methods based on measured consumer experiences within a service market. Int. J. Adv. Intell. Syst. **8**(3&4), 347–373 (2015b)

Kjolstad, F., Dig, D., Acevedo, G., Snir, M.: Transformation for class immutability. In: Proceedings of the 33rd International Conference on Software Engineering, ACM, pp. 61–70 (2011)
Kogan, A., Petrank, E.: Wait-free queues with multiple enqueuers and dequeuers. ACM SIGPLAN Not. **46**(8), 223–234 (2011)
Kogan, A., Petrank, E.: A methodology for creating fast wait-free data structures. ACM SIGPLAN Not. ACM **47**, 141–150 (2012)
Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. **C–28**(9), 690–691 (1979). doi:10.1109/TC.1979.1675439
Li, X., Garzarán, M.J., Padua, D.: A dynamically tuned sorting library. In: Proceedings of the International Symposium on Code Generation and Optimization (CGO'04), IEEE Computer Society, pp. 111ff (2004)
Löwe, W., Neumann, R., Trapp, M., Zimmermann, W.: Robust dynamic exchange of implementation aspects. In: TOOLS (29), pp. 351–360 (1999)
Michael, M.M.: 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, pp. 73–82 (2002)
Michael, M.M.: Hazard pointers: safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. **15**(6), 491–504 (2004)
Michael, M.M., Scott, M.L.: 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, pp. 267–275 (1996)
Moir, M.: Transparent support for wait-free transactions. In: Distributed Algorithms, Springer, pp. 305–319 (1997)
Moura, J.M.F., Johnson, J., Johnson, R.W., Padua, D., Prasanna, V.K., Püschel, M., Veloso, M.: SPIRAL: Automatic implementation of signal processing algorithms. In: High Performance Embedded Computing (HPEC) (2000)
Oracle: Object (Java Platform SE 8). (2016). https://docs.oracle.com/javase/8/docs/api/java/lang/Object.html. Online, Accessed 13 July 2016
Österlund, E., Löwe, W.: Dynamically transforming data structures. In: Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, IEEE, pp. 410–420 (2013)
Österlund, E., Löwe, W.: Concurrent transformation components using contention context sensors. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering, ACM, New York, NY, USA, ASE '14, pp. 223–234 (2014). doi:10.1145/2642937.2642995
Österlund, E., Löwe, W.: Concurrent compaction using a field pinning protocol. SIGPLAN Not. **50**(11), 56–69 (2015). doi:10.1145/2887746.2754177
Pirinen, P.P.: Barrier techniques for incremental tracing. In: Proceedings of the 1st International Symposium on Memory Management, ACM, New York, NY, USA, ISMM '98, pp. 20–25 (1998). doi:10.1145/286860.286863
Pizlo, F., Frampton, D., Hosking, A.L.: Fine-grained adaptive biased locking. In: Proceedings of the 9th International Conference on Principles and Practice of Programming in Java, ACM, pp. 171–181 (2011)
Russell, K., Detlefs, D.: Eliminating synchronization-related atomic operations with biased locking and bulk rebiasing. In: Proceedings of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications, ACM, New York, NY, USA, OOPSLA '06, pp. 263–272 (2006). doi:10.1145/1167473.1167496
Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Minh, C.C., Hertzberg, B.: 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, ACM, pp. 187–197 (2006)
Schonberg, E., Schwartz, J., Sharir, M.: Automatic data structure selection in SETL. In: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages, ACM, pp. 197–210 (1979)
Schultz, U.P., Lawall, J.L., Consel, C., Muller, G.: Towards automatic specialization of Java programs. In: Proceedings of the 13th European Conference on Object-Oriented Programming (ECOOP'99), Springer, pp. 367–390 (1999)
Shavit, N., Touitou, D.: Software transactional memory. Distrib. Comput. **10**(2), 99–116 (1997)
Sundell, H.: Wait-free reference counting and memory management. In: Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International, IEEE, pp. 24b (2005)

Svahnberg, M., van Gurp, J., Bosch, J.: A taxonomy of variability realization techniques. Softw. Pract. Exp. **35**(8), 705–754 (2005). doi:10.1002/spe.652

Timnat, S., Braginsky, A., Kogan, A., Petrank, E.: Wait-free linked-lists. In: Principles of Distributed Systems, Springer, pp. 330–344 (2012)

von Löwis, M., Denker, M., Nierstrasz, O.: Context-oriented programming: beyond layers. In: Proceedings of the International Conference on Dynamic Languages (ICDL'07), ACM, pp. 143–156 (2007). doi:10.1145/1352678.1352688

Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimizations of software and the ATLAS project. Parallel Comput. **27**(1–2), 3–35 (2001). http://citeseer.ist.psu.edu/article/whaley00automated.html

Xu, G.: Coco: sound and adaptive replacement of java collections. In: 27th European Conference, Montpellier, France, July 1–5, 2013. Proceedings, Springer, Berlin, pp. 1–26 (2013)