UNIVERSITY OF TARTU

FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

Institute of Computer Science

Computer Science

Kristjan Kelt

Immutable data types in concurrent programming on basis of Clojure language

Bachelor thesis (6 EAP)

Supervisor: Oleg Batrashev

Author:	Kristjan Kelt	"….." May 2013
Supervisor:	Oleg Batrashev	"….." May 2013

Approved for defence

Professor: ………………………………… "….." May 2013

TARTU 2013

Table of Contents

1 Introduction....4
2 Overview of the multi threading problems....6
  2.1 Shared memory....6
  2.2 Coordination between threads....6
  2.3 Coordination induced performance problems....7
3 Current state of Java concurrent programming....9
  3.1 Java platform....9
  3.2 How concurrency is added into Java....9
  3.3 Mutual exclusion in Java....10
  3.4 Enforcing mutual exclusion....11
  3.5 Concurrency improvements in Java version 5....12
  3.6 Summary of concurrent programming in Java....13
4 Concurrency in Clojure....14
  4.1 Clojure introduction....14
  4.2 Concurrent programming in Clojure....14
  4.3 Immutable data types....15
    4.3.1 Immutability....15
    4.3.2 Immutable data types in Java....16
      4.3.2.1 Drawbacks....17
    4.3.3 Clojure collection data types....17
    4.3.4 Persistent linked list....18
    4.3.5 Persistent vector....18
      4.3.5.1 Binary trie or a digital search tree....19
      4.3.5.2 2k tries....23
      4.3.5.3 Performance improvements....24
    4.3.6 Persistent vector performance measurement....24
      4.3.6.1 Performance tests....25
      4.3.6.2 Test results....25
    4.3.7 Suggestions for persistent vector additional performance improvements....30
      4.3.7.1 Broaden immutability definition....30
      4.3.7.2 Change control using AtomicBoolean....31
      4.3.7.3 Identify different threads....31
      4.3.7.4 Test results....33
      4.3.7.5 Immutable list building with mutable companion....34
      4.3.7.6 Test results....35
  4.4 Sharing data between threads....37
    4.4.1 Actor model....37
    4.4.2 Agents....38
    4.4.3 Software transactional memory....38
      4.4.3.1 Compare and swap....39
      4.4.3.2 STM comparison with CAS....39
      4.4.3.3 STM comparison with locks....41
  4.5 Summary of Clojure concurrent programming....43
5 Summary....44

1 Introduction

While we may perceive the world as sequential because there appears to be a certain order of events that is fixed in time, the world around us is parallel by nature and everything happens simultaneously. The computers have long time imitated this naive world view but in a few recent years the processors that support running multiple calculations in parallel have become more and more prevailing. Processors with multiple cores are now even found in the mobile phones that decade ago used simple microprocessors.

Very often we are concerned more about processes interference with each other, while sharing common resources, than about their parallel nature. This what concurrency is about, indeed there is a lot of concurrency in the real world. Roads can only fit a fixed number of cars on the single lane and similarly a cashier in the super market can attend only one person at time and the queue is needed to organize the access to this resource. Beside being parallel, the world is highly concurrent. The similar limitations do apply to the computer systems.

The limits set by the processor architecture are explored in Section 2 "Overview of the multi threading problems". It appears that cascade of the problems is inherited from the shared memory model where the main memory is shared between different computing cores and the only method to the programmer to exchange information between different running threads is to write and read known memory addresses.

Running programs concurrently is not a new paradigm and have been widely in use among the general public since emerging of the modern operation systems with the graphical user interfaces. The concurrency in the single core system can provide apparent responsiveness – both the part of the program doing long running background calculations and the part of the program showing user interface can make a progress. Therefore it can not be surprising that the new high level programming language Java that appeared in 1995 incorporated a set of concurrency features. With modern multi core processors the programming systems with concurrency support can take advantage of the added parallelism relatively naturally. Legacy of the Java in concurrent programming is evaluated in Section 3 "Current state of Java concurrent programming".

Over the time it became more on more clear that the method of solving concurrency problems in Java is relatively error prone. While the programs appeared to be working to the programmers during the test phase, the programs could dramatically become completely halt after running several years in the production.

This understanding has lead to search of new paradigms for programming concurrent problems and to the reevaluation of the older paradigms that have not become widely popular. One of such one man quests has lead to the emergence of the new programming language Clojure. Clojure combines an old Lisp programming language with the new thinking about solving concurrent problems on top of the Java ecosystem.

While Clojure Lisp like syntax features are not explored in this work, Section 4 "Concurrency in Clojure" evaluates the concepts introduced by this language. The main focus of this work lies on the immutability of the data structures. The internal structure and the performance of one of the Clojures most novel data types Persistent Vector is explored in the sections Section 4.3.5, Section 4.3.6 and Section 4.3.7. In addition small set of performance improvements are suggested and discussed.

Overview of the data sharing methods in Clojure is given in Section 4.4.2 “Agents” and Section 4.4.3 “Software transactional memory”.

2 Overview of the multi threading problems

2.1 Shared memory

The root of the concurrent programming difficulty is shared memory hardware architecture and the need for communication between different threads. This means that when two separate threads need to share some information, it can be only done by one thread writing to the known memory address and another thread reading from that address (for example by periodically checking given address).

The problem arises when composition of processor instructions must have certain logical integrity and order. In single processor and single processor core systems it is possible that the processor time is given to another thread in between two dependent instructions, in multi-core or multiprocessor system it is in addition possible that two processing units want to read or write the depending memory addresses simultaneously. The program is not sequential anymore and multiple reads and writes appear in seemingly random order.

It is even more complicated when we take into account that the modern processor do have multiple level caches before the main memory and due processor memory model[1] different processor cores may now see the changes in more relaxed order or may not see at all [1].

Higher level programming languages can make the situation even more complicated for the programmer because to gain more performance the compiler can change the execution order of the statements [2] and the statements appearing to write into memory[2] actually keep the values in processor registers or write into memory only after the end of the calculation ([3] Section 17.3, [4])

2.2 Coordination between threads

Therefore it becomes apparent that there should be some form of coordination between different threads when there is risk that reads or writes of different threads may overlap.

Process to accomplish this is called mutual exclusion what will ensure that no two threads are in the critical section, that is, in the code block that accesses shared memory (or any shared resource in general). General method to enforce mutual exclusion that may involve changing multiple memory locations is to use locks that are requested before entering critical section and released after leaving it but modern processors support special instructions that guarantee secure writing of values that fit into one processor architecture specific word (32 bit or 64 bit accordingly) within one instruction. ([5], [6])

Locks are very general solution and are not related to any specific memory address. One lock can be used to guard different locations or two locks can guard overlapping sections. It is programmer or higher level software architecture responsibility to guarantee that the locks are used properly – that critical section is correctly determined, that locks are taken before entering and that locks are released correctly.

It may be possible that actual shared memory is accessed without acquiring a lock or by acquiring a

1 Processor memory model defines what writes to the shared memory addresses may be seen by the reads executed by the other threads. [1]

2 When the source code of the higher level programming language is inspected.

wrong lock. It may be that locks are not released properly (for example after software exception). [7]

There is also specific problem that is related to the lock taking order – deadlocking. It is possible that two critical sections depend on each other and take locks in opposite order ending in situation where both of them can not proceed because they are waiting after each other (see Figure 1). [7]

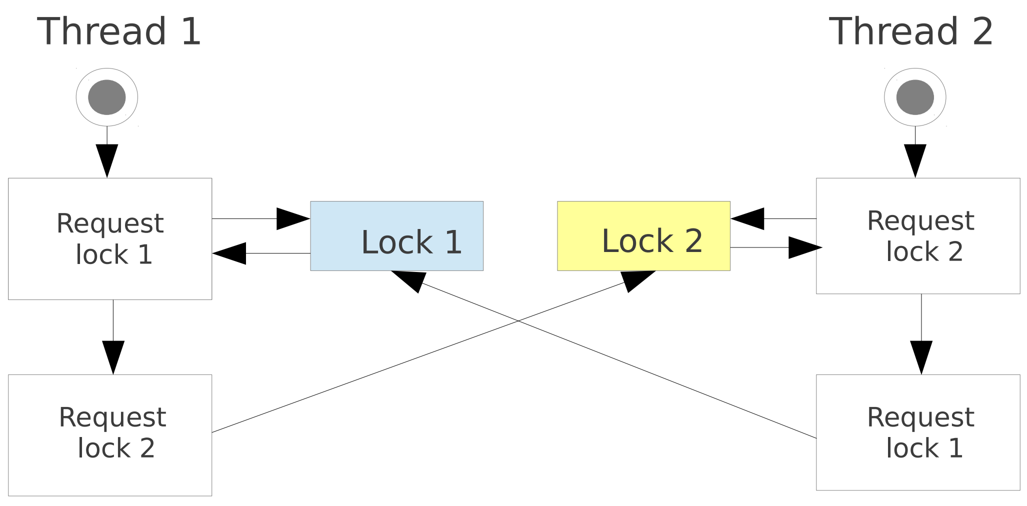

Figure 1: Deadlock

It is possible to reduce the risk of deadlock by carefully arranging the lock taking order but this method is again prone to human error.

2.3 Coordination induced performance problems

Acquiring but also releasing a lock always involves additional overhead especially because to guarantee the memory consistency in tiered memory architecture the processor must be forced to flush cache buffers to guarantee that all the changes are written into memory. In addition the compiler should arrange the code consequently that the values expected to be in memory are not cached in processor registers. When this is not done, one of the threads may never see the changes made by other.

When thread tries to request a lock that is already held by another thread then it will be forced to wait. This situation is called lock contention [6] and longer the critical section is the more likely it will cause it. [8] This problem is especially harmful for systems with multiple processors or multiple processor cores because it creates a point where program can not proceed in parallel due all the cores waiting after single section of the code.

Another problem is that depending on the lock implementation, it may result in context switch[3] for waiting threads further degrading the performance. [10]

It is important to note that when two threads are only reading memory then they can do this

3 A process to store the state of the current running thread and replace it with another one. [9]

simultaneously without error[4]. This allows to use read/write locks where only thread to write into memory will have to wait till all previously taken locks are released and where it is the only lock that will block others to proceed before releasing it. [11] When most of the accesses are read only then using read/write locks can provide noticeable performance gains. [7]

4 It must be still ensured that the writes are synchronized properly before the reading.

3 Current state of Java concurrent programming

3.1 Java platform

Java is one of the popular general-purpose object oriented languages whose history dates back into early 1990s when it was developed in Sun Microsystems and it was introduced to public in 1995. ([12], [13], [14])

Java was designed as platform rather than language with main objective to allow programs once compiled to run on every device that has support for Java. [15] Therefore code written for Java platform is not directly compiled into platform specific machine code but into intermediate so called byte code that is then executed by Java Virtual Machine by interpreting the code or by using just in time (JIT) compilation techniques. [16]

Java Virtual Machine does not limit its support only to the Java language and there are now at least few vigorous relatively new languages that support other programming paradigms on top of the Java platform. For example Groovy adds scripting capability with dynamic data types, Processing adds visualization and animation domain specific language and Scala and Clojure add support for the functional programming paradigm. [17]

Java platform provides wide range of standard class libraries that provide beside core features like string manipulation, collection classes, a platform independent access to platform specific resources such as file access, network access but also multithreading. ([3], [18])

Java libraries are accessible to other programming languages on Java platform and other programing languages can make their own libraries and features available over the Java platform. [16]

Java is a garbage collected language meaning that the memory management (allocation and releasing) is provided by the Java runtime. In addition Java does not allow unbounded memory access – bounds of the arrays are verified and while the objects are created inside the heap memory, there is no direct access to the object pointers. [3]

3.2 How concurrency is added into Java

While being high level object oriented language, concurrency support in Java relies on very low level concurrency constructs that are added directly into language.

Java supports general concept of shared memory model where different threads can access the same memory addresses concurrently. Thread are added into Java as *Thread* object. There are two ways to create a new thread.

1. Create a class that extends *Thread* class. This subclass should override the *run* method[5]. An instance of the subclass can then be allocated and started.
2. Create a new *Thread* instance with parameter to class that implements the *Runnable* interface *run* method.

5 Programmer can in addition theoretically override the *start* method and call the *start* method of the *Thread* class but this is not correct programming practice. [7]

New threads are started by calling *start* method. This will create a new subroutine that is executed independently in the new thread. Thread objects do not provide a direct mechanism for differnet threads to communicate or exchange data. For example it would be possible for existing thread to create a new *Thread* instance, start a new thread and then call custom methods on this instance but those methods would be still executed within the calling thread.

Data sharing must be done by using objects that are known by both of the communicating threads. For example the new thread instance could contain a object variable that could be used for communication.

Due its multi platform nature, Java can not rely directly on the memory model provided by the underlaying hardware architecture. Therefore it defines its own memory model that can provide the same guarantees on every platform where Java is supported. [3] Java does not provide declarative definitions of related atomic variables and critical sections must be marked imperatively by the programmer. Except for few special cases the Javas support for mutual exclusion is managed using locks either internally by the runtime or explicitly by the programmer. ([18], [7])

Java language model of the concurrency means that concurrent programming in Java is open to the most of the problems inherited from the low level hardware architecture. [19]

3.3 Mutual exclusion in Java

The initial way to specify the critical sections in the Java code was by using the special *synchronized* keyword in the method signature or in the header of the anonymous code block for more granular control. When a thread enters a code block that is marked as *synchronized*, Java runtime request a lock before executing the code inside the block and releases it after leaving the block by either naturally or due runtime error. In addition Java runtime guarantees that all the relevant processor registers and caches are written into memory. Locks are identified by its target object what must be specified for the anonymous code block or is automatically method owner object[6] for the methods.

Java uses reentrant type of locks for synchronized blocks meaning that the locks taken by the same thread do not cause thread to block on the lock when called recursively.

While being relatively straightforward to apply and eliminating risk of some programming errors, the synchronization mechanism provides very limited control over it. Because locks are induced automatically around the synchronized block, it is not possible to leave the scope of the lock open. For example it does not give programmer option to take the lock in one method of the object and release it in the another. This kind of synchronization can be done only outside of the control of the objects. This means that atomic[7] operations that contain more that one method call on the object must be synchronized separately by the calling code. [7]

For two threads the already taken lock will always require a wait regardless of the nature of the code – *synchronized* keyword does not have a separate lock type for read only access. This is notably noticeable on the multi processor (-core) systems where two threads could potentially read and process data in parallel. In addition it is not possible to check beforehand if the lock is already taken, it is not possible to set timeout for wait if thread gets blocked by lock.

These problems were addressed to certain extent in Java version 5 by introduction of the Lock interface

6 Every object in Java contains an internal monitor that is used when synchronization is applied on the object.

7 Two operations are atomic when their side effects are not visible separately.

that allows to control locks programatically. Unfortunately the synchronization and Lock interface can not interface with each other. This effectively creates two separate incompatible methods to apply mutual exclusion in Java. [7]

3.4 Enforcing mutual exclusion

One of the object oriented programming design principles is to use encapsulation to prohibit uncontrolled manipulation of the objects state. Object state can only modified via method calls that can ensure that object remains in the consistent state after every method call. ([20] Section 1.6, [21] Chapter 4)

Enforcing this integrity is natural in sequential programming because every method call can assume that the object was left in the consistent state after prior method calls. *Ad hoc* incorporation of the low level multi threading operations into object oriented language like is done in Java breaks this assumption because now different thread accessing the object can see the object state in the middle of the method call. It can be argued that object oriented programming and concurrent programming interfere with each other. [22]

In this paradigm the programmers first task is to identify if the class is supposed to be shared between different threads. A class can be designed not to follow any thread safety principles. For example many standard classes in Java like collection classes[8], dates, are not thread-safe and it is their user responsibility to guarantee that those classes are used properly.

Java does not provide a simple method to avoid using not-thread safe objects unsafely by multiple threads. For example it is possible (not prohibited by compiler or runtime) to execute the code on the Figure 2 by multiple threads. While this code may work, it is not guaranteed to work and it is possible that thread executing the *run* method does not see the change to the *stopCrawling* variable[9].

```
public class LinkCrawler implements Runnable {

      private boolean stopCrawling = false;

      public void stopCrawling() {
             stopCrawling = true;
      }

      public void run() {
            while (!stopCrawling) {
                  crawlMoreLinks();
            }
      }
}
```

Figure 2: Not thread-safe link crawler can be called by different threads.

8 Except concurrent collections in different package and the indigenous synchronized *Vector* class.

9 This code can be fixed by marking the *stopCrawling* variable with the *volotile* keyword.

Applying the locks inherits the problems from the shared memory model. Figure 3 shows how deadlocks can occur in Java code. On their own both methods in the class apply locks correctly but when one thread is calling the addNumber method and another thread is calling the removeNumber method then there is a risk that both threads will deadlock by unfortunate timing. This code can be fixed by using the same lock taking order in both methods.

```
public class  BigNumbers {

      private final List<BigInteger> numbers =
                                   new ArrayList<BigInteger>();

      private BigInteger summary;

      public void addNumber(BigInteger newNumber) {

            synchronized (numbers) {
                  synchronized (summary) {
                        numbers.add(newNumber);
                        summary =  summary.add(newNumber);
                  }
            }

      }

      public void removeNumber(BigInteger existingNumber) {

            synchronized (summary) {
                  synchronized (numbers) {
                        numbers.remove(existingNumber);
                        summary =  summary.subtract(existingNumber);
                  }
            }

      }
}
```

Figure 3: Example of deadlock in Java.

Another set of problems can be identified as check and act misuse. [23] For example it could be first checked if a synchronized collection contains a element and then when it does not, element will be added into collection. While both method calls are thread safe separately, they will not form a atomic operation without additional synchronization.

This indicates a wider problem related to the programming with locks – locks do not compose. It is hard to combine two separately atomic method calls into a new atomic operation.

3.5 Concurrency improvements in Java version 5

Java version 5 introduced beside many new welcomed language features a more strict Java memory model. More importantly complete set of new application programming interfaces was introduced to solve some more common concurrency programming problems. [7]

The main focus was on providing better thread management by introducing easier and more convenient methods to execute threads and to help better signaling between threads. For example a *ExecutorService* interface along with the new *Callable* and *Future* interfaces and existing *Runnable* interface provides more practical method for task or calculation execution instead of direct use of the *Thread* class, classes *CountDownLatch* and *CyclicBarrier* simplify the signaling between the threads. Direct use of *Thread* class and *Object wait*/*notify* methods is since Java version 5 highly discouraged. [7]

While new *Lock* interface was introduced in addition to existing locking mechanism with the *synchronized* keyword, it did not provide principal changes into Javas mutual exclusion. It provided better control and performance improvements especially by providing *ReadWriteLock* interface.

One very important addition was introduction of the atomic wrapper classes that added support for lock-free thread-safe programming with single variables. Wrappers are provided for primitives like boolean, int, long and arrays and object references. In essence, these classes extend the notion of volatile variable but provide also an atomic conditional update operations ([24], [18], [7])

Java 5 also introduced improved collection framework with fast but not thread safe collection classes that could be made thread-safe with synchronized wrapper classes and set of thread safe concurrent collections that were designed mostly for the performance. Unfortunately collection classes are still open for check and act concurrency bugs. [23]

3.6 Summary of concurrent programming in Java

Probably one of the most critical problem in Java for concurrent programming is that the mutual exclusion is solely programmer responsibility because language does not prohibit incorrect usage. While this makes it possible to fine tune the performance, it also opens possibility for very wide range of the programming errors.

While Java provides its own universal memory model over platform specific memory models, it only specifies what is guaranteed to work. This leaves open possibility that programs would defectively work in the development environment but fail inside the production environment or when moved from one production environment to another that is using different hardware platform or different JVM implementation. [25] For example not thread safe changes to the variable may be seen by the other threads but this is not guaranteed to work unless variable is marked as volatile or both reads and writes are properly synchronized.

Improved concurrency API does offers much more flexible control over applying mutual exclusion but does not bring improvement into difficulty of combining multiple atomic operations.

4 Concurrency in Clojure

4.1 Clojure introduction

Clojure is a functional Lisp inspired general purpose compiled programming language created by Rich Hickey. ([26], [27]) Clojure is created on top of the Java platform[10] and its core functionality is implemented in Java and then incorporated into language using Clojures powerful macro system. Clojure language possesses dynamic typing but allows to include type hints what are considered by the compiler during compile time to optimize the code execution. Clojures main focus is on providing strong platform for the concurrent programming.

Current work covers only Clojures JVM version aspects, especially its Java implementation side.

4.2 Concurrent programming in Clojure

Clojure provides very strict scope for variables or more precisely references because Clojure variables do not support primitive data types. Being functional language, most of the time it is natural to use stack confinement provided by the local bindings. [7] Local binding can be viewed as analog to the variables defined within a method. Their scope begins from their definition and ends after leaving the code block they were defined.

In addition to the local bindings *vars* define the thread local global binding. Every *var* accessed first time by every thread will initialize its own copy of the *var* that can be modified within the thread. Changes to *vars* are thread local and are not synchronized between other threads. In Java similar functionality can be archived with the *ThreadLocal* reference type. [7]

Clojure enforces that sharing data between thread is deliberate. There are two different forms of data sharing in Clojure. Data can be shared either synchronously or asynchronously.

1. Asynchronous changes are done using agents and
2. synchronous changes are done either by atoms or using references (*refs* shortly) managed by the software transactional memory. Software transactional memory allows to coordinate updating multiple references atomically. Update to atoms and refs can be retried when not successful due memory contention (see Section 4.4.3 for further details).

Figure 4 shows comparison between different Clojure reference types.

10 Clojure is supports now in addition Common Language Runtime and JavaScript engines.

	Local binding	Var	Agent	Atom	Ref
Stack confined	X				
Thread confined		X			
Shared			X	X	X
Asynchronous			X		
Synchronous	X	X		X	X
Coordinated					X
Retriable				X	X

Figure 4: Clojure reference types.

4.3 Immutable data types

4.3.1 Immutability

It is common to think that object oriented programming provides a good model of the real world. It feels natural that properly designed objects in the program represent objects in the real life, they are defined by their internal state only changeable through methods that characterize them and protect the invariant of the guarded state and allow it change over time. Objects state is mutable because this represents how they behave in the real world.

Lets take for example a black box and lets try to model it using these principles. We create a class Box that represents the box and we add a internal property that represents the color. We add a method to change the color and one to query the color, we add a constructor to create new boxes with specific color.

But there is a problem. When we decide to paint the box blue, we would eventually end up with a completely new box. But there are references. There might be pictures of the still black box. There are memories. The then black box does not suddenly turn into blue. In this regard our model is actually very different from the real world.

We could remodel our box the way that the method to change the box color does actually not change the internal property of the object but creates and returns a new object that has new value for the color and the object state never changes after creation.

Now when we obtain the newly repainted box back to the original reference the existence of the box with the old color would disappear. Unless there were other references (see Figure 5 for illustrative usage).

```
Box blackBox = new Box("Black");

List<Box> historyRecords = new ArrayList<Box>(Arrays.asList(blackBox));

Box blueBox = box.paint("blue");  // blackBox is still "black"
```

Figure 5: Illustrative usage of the immutable Box class.

In theory we can classify data types according to this difference as mutable and immutable. Data types are mutable when it is possible to change their inner state after creation and data types are immutable when this is not possible. [21]

4.3.2 Immutable data types in Java

The garbage collecting makes operation with the immutable types very easy and guarantee of invariability makes it possible to regard instances of even complex immutable class as values similarly to primitive types.

There are many other benefits designing classes as immutable. Immutable classes can be easier to design, implement, and use than mutable classes with the same purpose. They can be less prone to error and are more secure and are easier to understand and argue about because their state is always defined after creation. [21]

Following guidelines should be followed in Java when designing a immutable class.

1. Immutable class should not provide any methods that can change its state.
2. It should not allow subclassing (because deriving class can break this immutability contract).
3. All fields should be marked as *private* and *final* and it should be ensured that class does have exclusive access to its mutable components (reference to mutable components should not leak out of the class).
4. Immutable class must be properly constructed by not letting reference to this escape during construction.

Due value like behavior it is safe to pass a reference to immutable object to a method or return it from the method without worrying about the uncontrolled state changes. One example of this usage is using immutable objects as map keys.

In addition it gives immutable classes a very strong advantage in concurrent programming – because their state can not be changed, they are inherently thread-safe and can be freely shared between threads without worrying that two different threads will fail to change their internal state properly.

Distinction between mutable and immutable data types is not something new inside the Java community. In fact most core Java classes like primitive type wrappers, String class, BigInteger etc. are designed (or at least intended[11]) to be immutable.

11 BigInteger can be extended and is not therefore correctly immutable. [21]

4.3.2.1 Drawbacks

Immutable data types are not completely free from drawbacks. Every distinct state requires a separate instance for it. While creating a lot of objects is not a concern anymore like it was with the earlier Java versions [7], it still will not perform like simple method call and it will add additional stress to the garbage collection. Especially creation of bigger objects inside multi step operation where eventually only last result is needed can perform considerably worse than the same operation with similar mutable object. There are few methods for overcoming or reducing these problems.

Immutability makes sharing inner state among derivative objects easy and this is one way of reducing need for copying. For example Java *String* method *substring* creates new object that reuses the underlaying buffer and has only new values for start index and length [specify] or *BigInteger* negate method creates new object that has the opposite sign value but shares the bit array. In fact the *BigInteger* class could go even further and split the bit array into multiple junks and create only copies of junks when change is needed in the particular junk and share the rest among different instances.

Another method is to guess what multistep operations may be useful for the class under design. This will allow to apply multiple operations internally without creating new objects each time. Finally a mutable companion class can be designed that allows to perform multiple operations over mutable dataset and then convert itself into immutable instance that can be freely shared. Java *String* and *StringBuilder* are two of such companion classes for example.

Despite well understood and established framework for immutable data type creation, the support for them in Java is still limited. Especially the collection framework consist only mutable variants of the collection classes and a number classes like *Date* were unfortunately designed as mutable.

4.3.3 Clojure collection data types

Clojures approach to concurrent programming relies heavily on immutable data types. As a result the availability of basic immutable collection data types that just do not create full copies after each modification becomes unavoidable for it success. Such data types are called persistent immutable data types in Clojure to underline that they always preserve the previous version of themselves after they are modified. [27]

Clojure applies three earlier mentioned design principles to collections. First it creates set of data structures that on creation of extended copies do not create full copies but share as much data as possible, secondly it provides in class solutions for many commonly occurring problems and third it provides mutable companion classes that are fast to mutate and fast to convert into immutable counterparts. [27]

The availability of the simple and complex immutable data types creates a foundation for the Clojures concurrency approach.

In the following two Clojures immutable collection types like persistent linked list and persistent vector are described in more detail. Clojure offers also an immutable hash map implementation but this data structure is not covered in details in this work.

4.3.4 Persistent linked list

One of the very basic collection data types that can be straightforwardly made immutable is linked list (represented by IPersistentList interface in Clojure and implemented in PersistentList class). Linked list is a list where the items are added to the beginning of the list. Only thing necessary for add operation implementation is to create a new separate instance for every new element. This instance must then contain data and reference to the previous instance on what top it was added (see Figure 6 lists a, b, c, d and e can be individually modified and can share internal structures). [27]

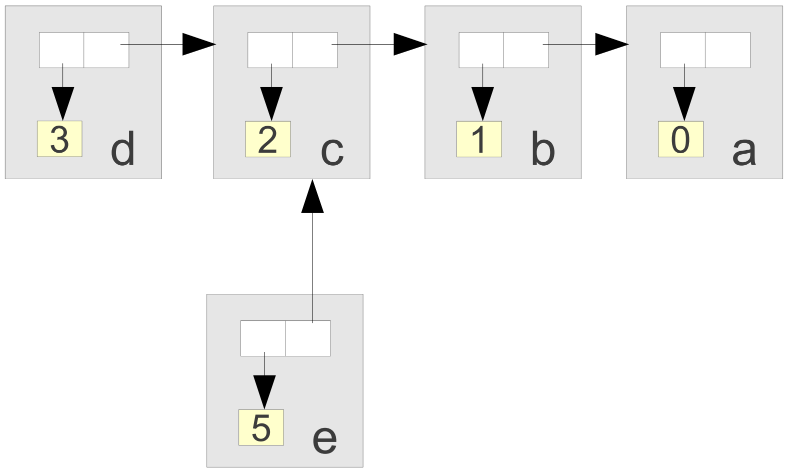

Figure 6: Immutable linked list

Immutable linked list provides O(1) time complexity for adding to the beginning of the list, reading from the beginning of the list and for removing from the beginning of the list. Linked list is not universally usable data structure because it can not be efficiently traversed by the order items were added into it. Still it is clearly a good candidate for a stack and is therefore used by Clojure to represent its code.

This data structure does obviously not provide a good performance for index lookups and as a result this functionality is not provided directly in Clojure – linked lists must be converted into indexed lists (named vectors in Clojure) to be accessed by the index.

4.3.5 Persistent vector

One of the most interesting collection data structures in Clojure is Clojures persistent vector (represented by IPersistentVector interface in Clojure and implementedin PersistentVector class)., a immutable indexed list implementation. This data structure differentiates Clojure from the previously existed (functional) languages.

Persistent vector provides add to the end of the list, look up by index and update by index by very attractive O(~1) time complexity while being immutable. Notation O(~1) means here “near O(1)” because the time complexity is only approximatively O(1) for practical applications as it is explained subsequently.

4.3.5.1 Binary trie or a digital search tree

To explain Clojures vector, it makes sense start explaining from simpler data structure – a binary trie.

It is possible to implement a indexed list as a binary tree where all the elements in the list are kept as tree leafs and the element index in binary form supplies a path in the tree from the root to the leaf node. The index lookup must start from the root node. When the first bit (when reading from left to right) has value 0 then left branch should be taken and when bit has value 1 then the right branch should be taken as the next root node. When the next root node is the leaf node then the element is found. All leaves in this tree must have the same depth (see Figure 7). ([28], [29])

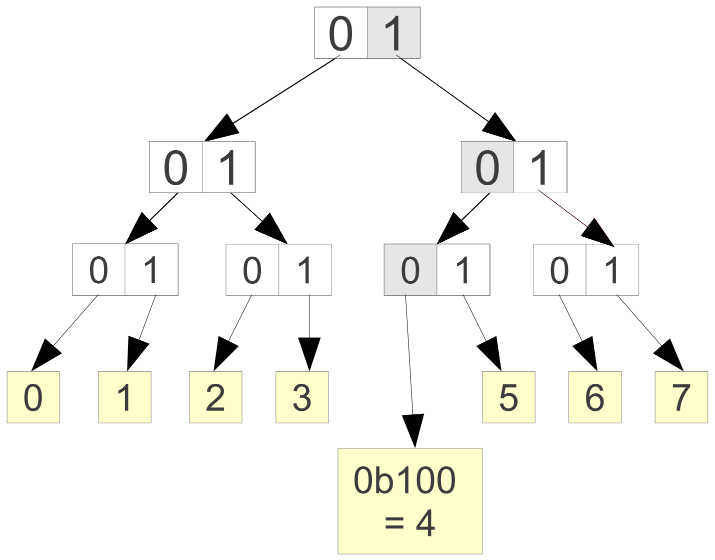

Figure 7: Binary trie

It is easy to see that this data structure allows the lookups by index with O($\log_2$M), where M is current list size rounded up to nearest power of 2, time complexity what for pratical applications like around milion to 10 million records will still require relatively low count of operations (for 100000 17, for 1000000 20). The update (replacing the element) by index has the same complexity.

There are following possibilities when adding a new element into tree to next position.

1. When there is free place on the right most right leaf node then the element can be added directly into tree (see Figure 8).
2. If this is not possible then the next upper root node must be examined for free place. When there is free position then the new branch can be created and the new element can be added as left node of the left most path (see Figure 9).
3. When tree is full then the new level must be created by adding a new root node. The old tree comes as a left branch of the new root node and new element is added as left node of the left most path from the root node right branch (see Figure 10).

The binary trie is on its own a mutable data structure but it is easy to see an efficient method to make it immutable. When adding a new elements to the end of the list represented by the binary trie, only the path to the last root element is nessesary to recreate. Everything that remains to the left side from the path can be reused by the new tree (see figures Figure 11 and Figure 12).

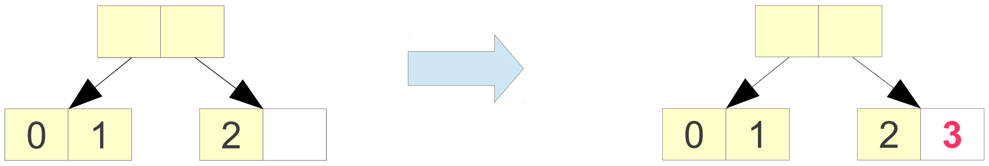

Figure 8: Adding element to the end of the list, option 1

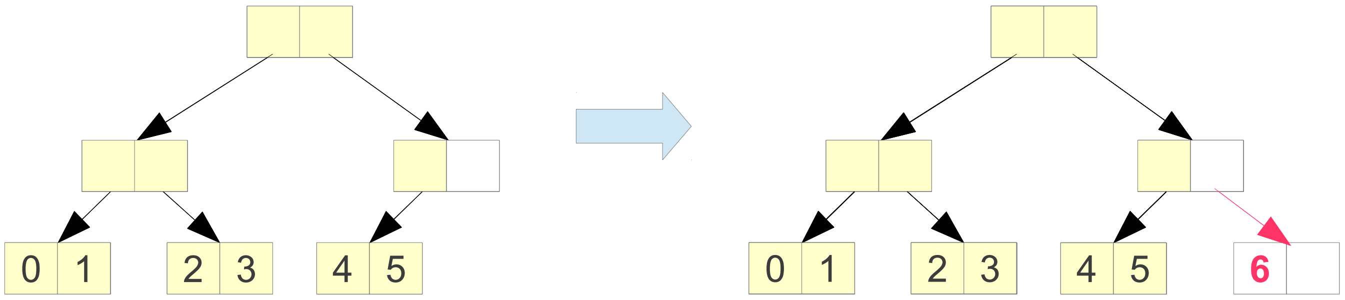

Figure 9: Adding elements to the end of the list, option 2

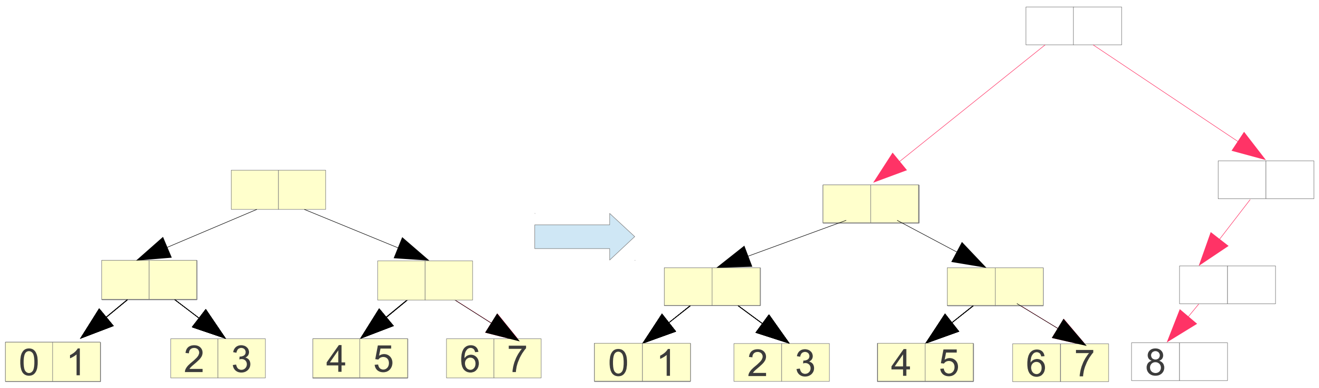

Figure 10: Adding element to the end of the list, option 3

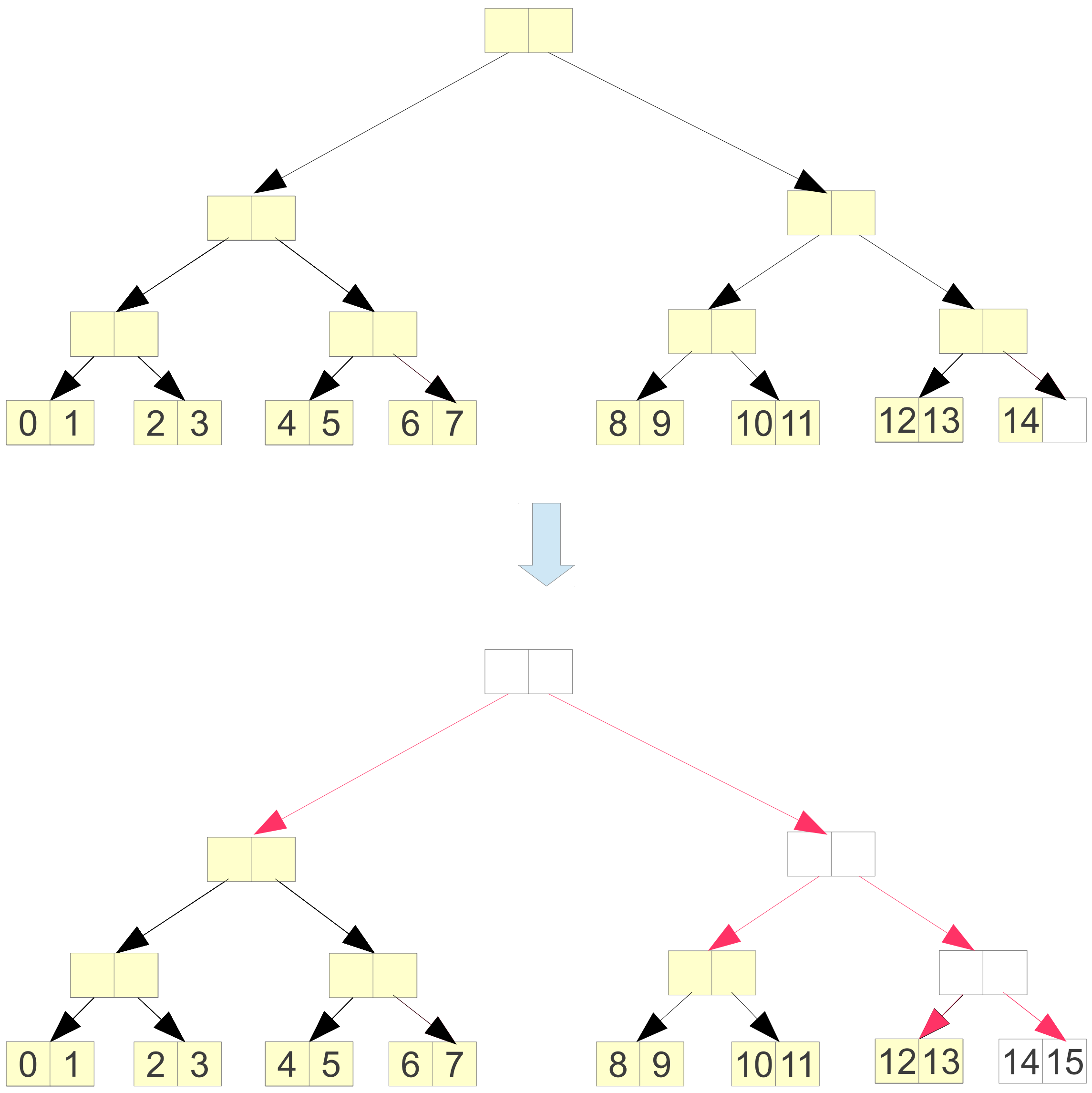

Figure 11: Adding element to the immutable binary trie

Example in the Figure 11 indicates how the structure can be shared between two different versions.

When an element 15 is added into free place at the end of the list (trie above) then the path beginning from the root node must be recreated.

Trie below shows how much it is possible to reuse the old structure. White blocks with red arrows (references) indicate the new path created. The blocks with the not changed color show how much is possible to reuse the trie from the previous version (above).

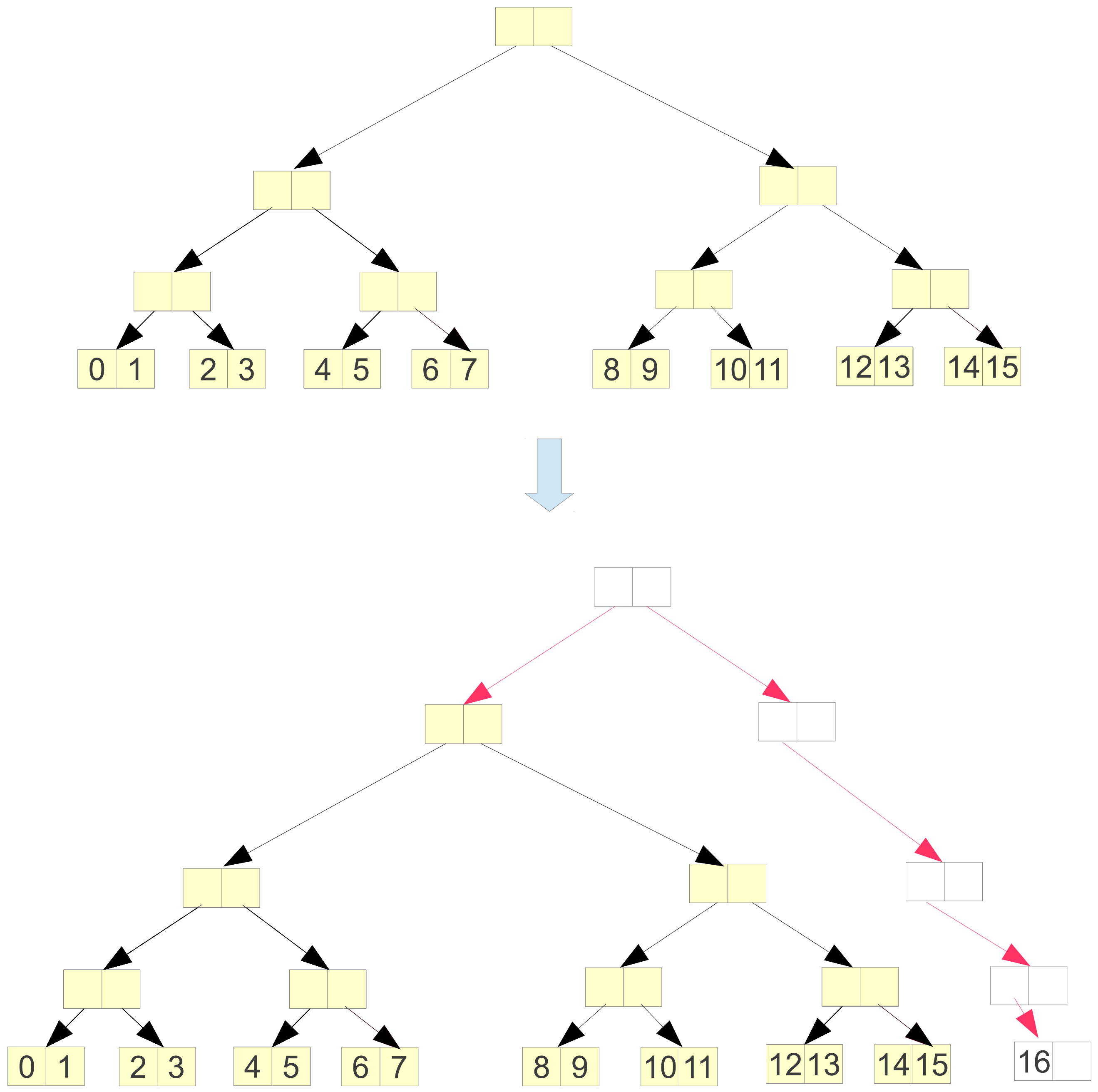

Figure 12: Adding element to the immutable binary trie

In Figure 12 the previous version of the list (represented by binary trie) is reused fully when new version that has element 16 added to the end is created.

Trie below shows how much it is possible to reuse the old structure. White blocks with red arrows (references) indicate the new path created. The blocks with the not changed color show how much is possible to reuse the trie from the previous version (above).

4.3.5.2 2^k tries

While the binary trie can be used as relatively efficient immutable indexed list implementation, it is only a special case of more generic and more interesting set of data structures – 2^k tries where 2^k represents branching factor of the tree ($2^1 = 2$ is the binary trie special case) (see Figure 13).

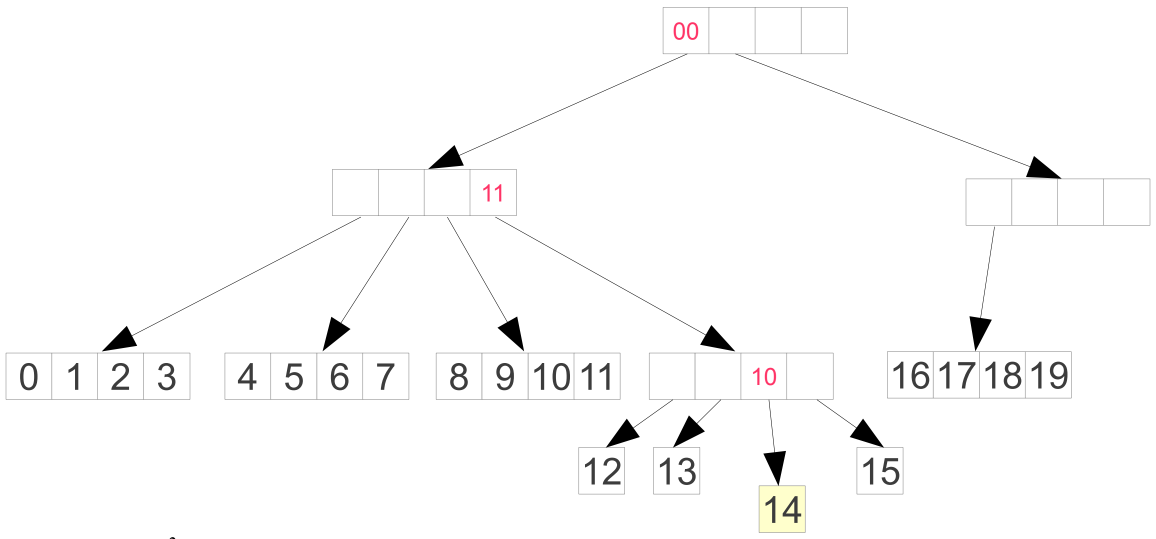

Figure 13: 2^2 trie with branching factor 4

For example for 2^2 trie with branching factor 4 with current depth 3, the trie can hold 64 elements and therefore all the elements in the trie can be represented by 6 bit number. The path for 14th element (counting from zero, see Figure 13) is decomposed then as follows (00), (11), (10).

It appears that all the valuable qualities of the binary trie also do apply to the 2^k tries. Most importantly the index lookup works similarly but now the path direction for each subsequent node is expressed by k bits instead of 1. All the operations also become correspondingly faster because now the complexity of the mentioned operations is represented by $O(\log_{2^k} M^k)$, where M^k is list size rounded up to nearest power of 2^k, instead of $O(\log_2 M)$. Complexity for branching factor 32 (k=5) is already around 5 times better than for binary trie (see figures Figure 14 and Figure 15).

k	Branching factor	Nearest power of branching factor	Complexity
1	2	131072	17
2	4	262144	9
3	8	262144	6
4	16	1048576	5
5	32	1048576	4
6	64	16777216	4
7	128	2097152	3

Figure 14: Complexity of index lookup in 100000 element list

k	Branching factor	Nearest power of branching factor	Complexity
1	2	1048576	20
2	4	1048576	10
3	8	2097152	7
4	16	1048576	5
5	32	1048576	4
6	64	16777216	4
7	128	2097152	3

Figure 15: Complexity of index lookup in 1000000 element list

A 2^k trie with this branching factor (32, k=5) is the basis for the the Clojures persistent vector and this allows a very good $O(\log_{32}M^k)$ time complexity for random index lookups and updates. ([30], [31])

4.3.5.3 Performance improvements

More interesting is that the relatively big branching factor allows additional performance gains due ability to defer tree modifications. This can be done by an additional buffer of the size of the branching factor. This buffer can be used to collect added items before pushing them into trie as a whole effectively avoiding branching factor times tree modifications. While the whole buffer must be copied again and again for every new item to maintain the immutability, because buffers are realised as Java arrays, it can take advantage of the Java low level array copy operations.

The amortized O(1) complexity of the add to the end of the list, removal from the end of the list and peeking the last element of the list operations is amortized O(1) and because $O(\log_{32}M^5)$ is very low for pratical applications then most operations (one listed here and index lookups and updates by index) allows to make a claim that these are essentially constant time operations. [32] This makes it a considerably more universal list implementation compared to the linked list and shows that immutable data structures can reach behind simple and obvious linked lists.

4.3.6 Persistent vector performance measurement

One of the biggest interests of this work was to investigate the actual performance of the Clojures indexed list implementation because Java collection framework provides a very primitive but very efficient indexed list that is essentially a wrapper class around a Java array – *ArrayList*. Of course *ArrayList* is mutable. [33]

Querying and updating elements in the *ArrayList* by index does have an actual O(1) complexity because it is just one array lookup. Adding elements to the end of the list has complexity of amortized O(1) because while underlaying array must be enlarged when it becomes full, this is required only once after many inserts.

While Clojure vector looks good on paper, it would be interesting to see how this much more complex data structure compares against the baseline set by the *ArrayList*.

4.3.6.1 Performance tests

Four different tests were performend to compare the two data structures:

1. a sequential fill test where time to add new items to the end of the list is measured,
2. a sequential read test where the list is first filled outside of the test and then the time to iterate over all the list elements in the list is measured,
3. a random read test where the list is first filled outside of the test and then the time to access all the elements using the index is measured and
4. random update test where the list is first filled outside of the test and then the time to update all the elements by index is measured.

All test [34] were performed on the Amazon AWS 2 core High-CPU Medium Instance[12]. [35] Test were performed on Ubuntu Server 12.04.2 LTS 64-bit using Java OpenSDK 1.7.0_15[13].

Every test consists number of operations performed on list over set of predetermined operation counts. In every test the total number of operations equals with the (final) list size. Before the actual measurement is made the test is executed number of times configured to correlate with the test size (number of performed operations). Every measurement was executed independently in freshly started JVM instance. In total 10 different measurements were performed for each measurement point.

For constant time complexity operations the linear graph is expected.

4.3.6.2 Test results

Following figures illustrate the results of the performed tests. Performed tests show that the performance difference between the Clojure PersistentList and Java collections ArrayList is measurable for list filling resulting roughly in 5 times difference (in advantage of the ArrayList) (see Figure 16). The performance difference of the list iteration is roughly 2 times different in advantage of the ArrayList (see Figure 17). Additional tests are required to conclude the result of the random update test (see figures Figure 18, Figure 19 and Figure 20). Not very surprisingly the performance difference of the random updates is roughly 2 orders of magnitude (!) differenct in advantage of the ArrayList (see figures Figure 21 and Figure 22).

For sequential fill test in Figure 16 the list is filled for every measurement with n elements and the time to perform all the additions is measurement for every n. This test shows how much time it will take to fill the list with n elements.

Both graphs show linear growth tendency. For ArrayList the polynomial trend line did match slightly more accurately based on current measurements but it is also visible that current trend would more likely to continue till it reaches the linear trend line. The graph shows how need to increase the underlaying array size cumulates for different list sizes. There is anomaly of unknown origin for ArrayList around list sizes 100000 and 200000. Surprisingly this anomaly was consistent over multiple separate measurements.

12 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), 1.7 GiB of memory

13 OpenJDK Runtime Environment (IcedTea7 2.3.7) (7u15-2.3.7-0ubuntu1~12.04.1)
OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)

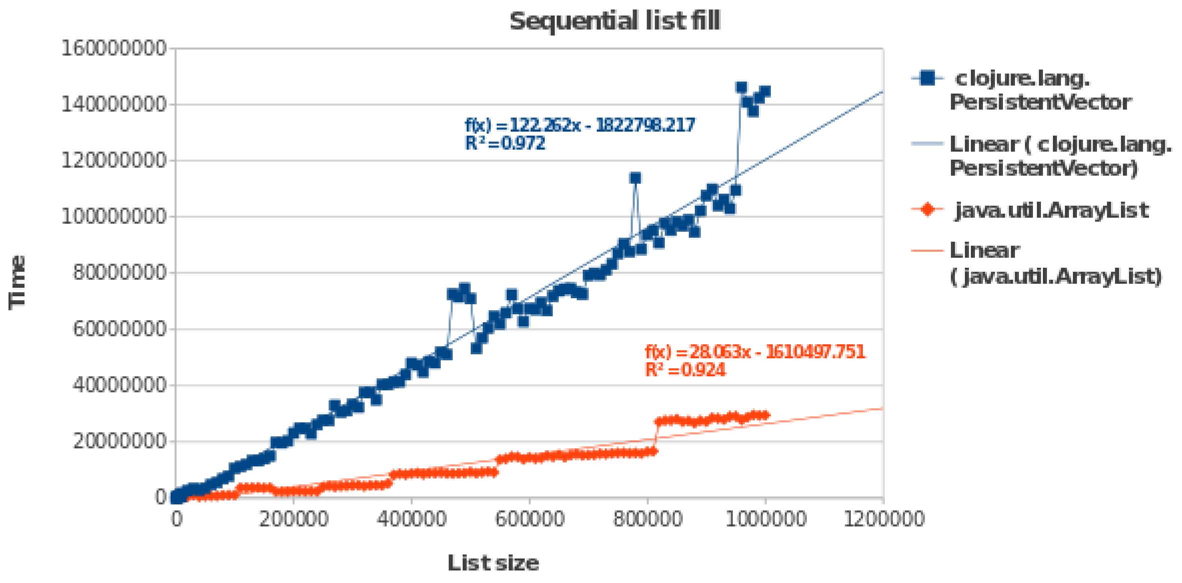

Figure 16: Test 1, sequential list fill test results. Clojure library class clojure.lang.PersistentVector is compared against Java library class java.util.ArrayList.

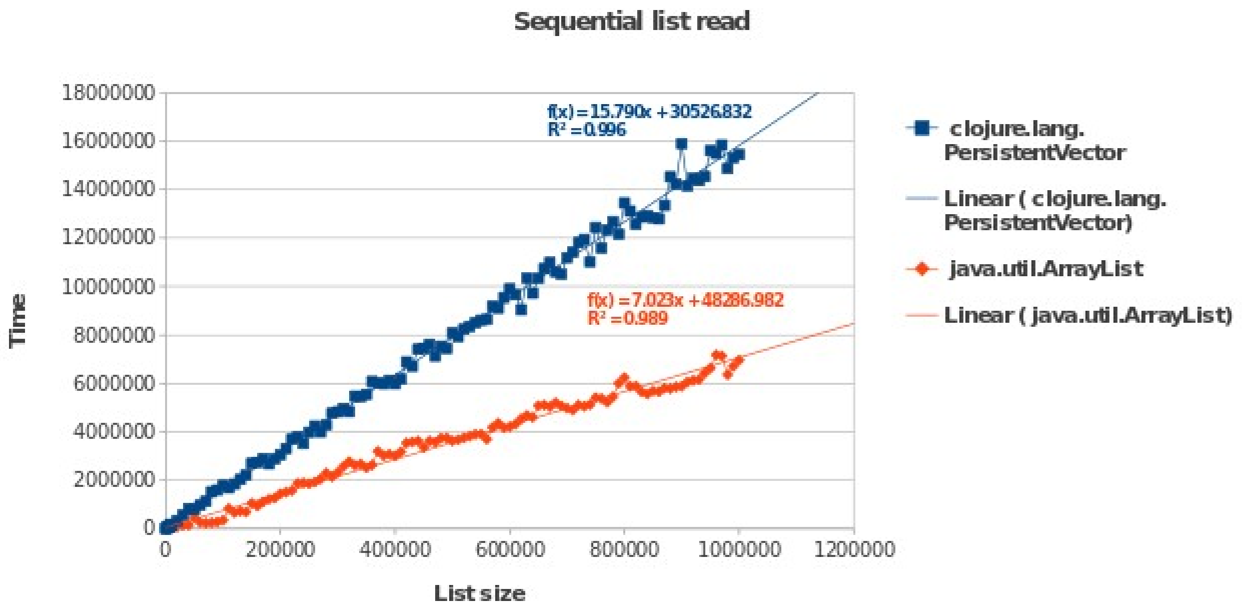

Figure 17: Test 2, sequential list read test results. Clojure library class clojure.lang.PersistentVector is compared against Java library class java.util.ArrayList.

The measured performance of the clojure.lang.PersistentVector roughly 5 times less than performance of java.util.ArrayList.

Figure 17 shows results from the sequential read test. For every measurement the list is filled with n elements and the time to perform sequential passage of the list (using the iterator) is measurement for every n. This test shows how much time it will take to iterate over all the elements in the list of the size of n. Both graphs show linear growth tendency. The measured performance of the clojure.lang.PersistentVector roughly 2 times less than performance of java.util.ArrayList.

Figure 18 shows test results for the random read test. For every measurement the list is filled with n elements and the time to perform random passage of the list (using the index) is measurement for every n. This test shows how much time it takes to randomly read over all the elements in the list of the size of n.

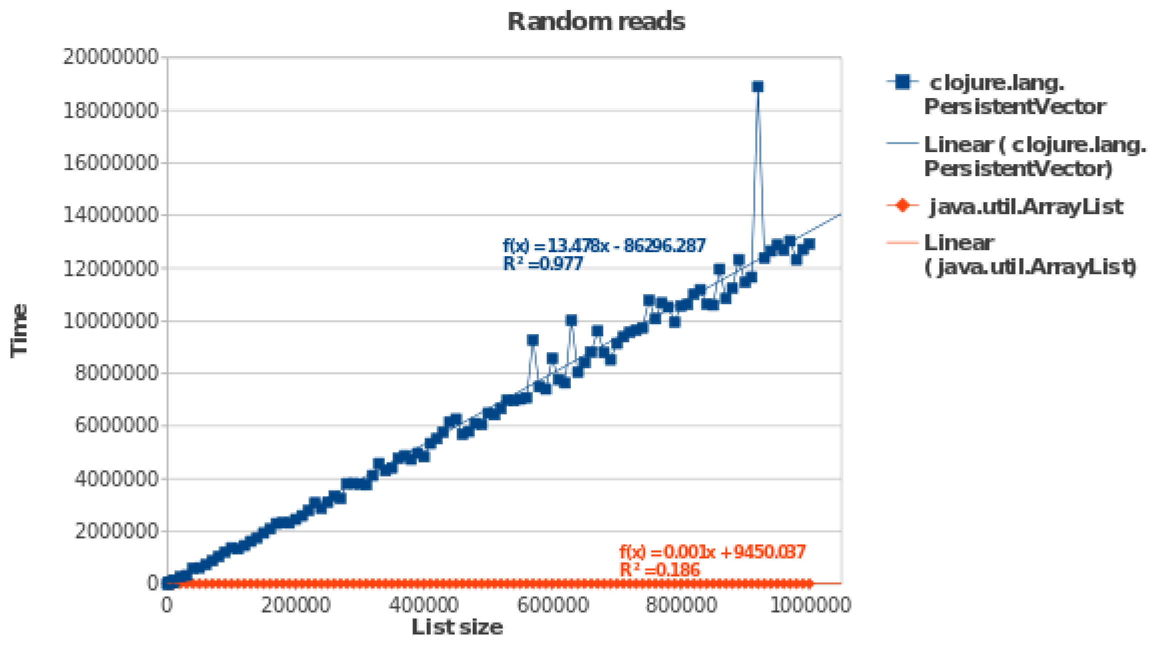

Figure 18: Test 3, random list read test results.

The graph for PersistentVector show linear growth tendency but for ArrayList while the graph looks linear in the diagram it really is not.

The ArrayList performance measurement graph in the Figure 19 does not follow the logarithmic trend either. It was hypothesized that because no actual operations are performed with the returned list elements and after inlining the method calls the JIT may realize this and optimize it away and the execution time would remain constant after certain test size.

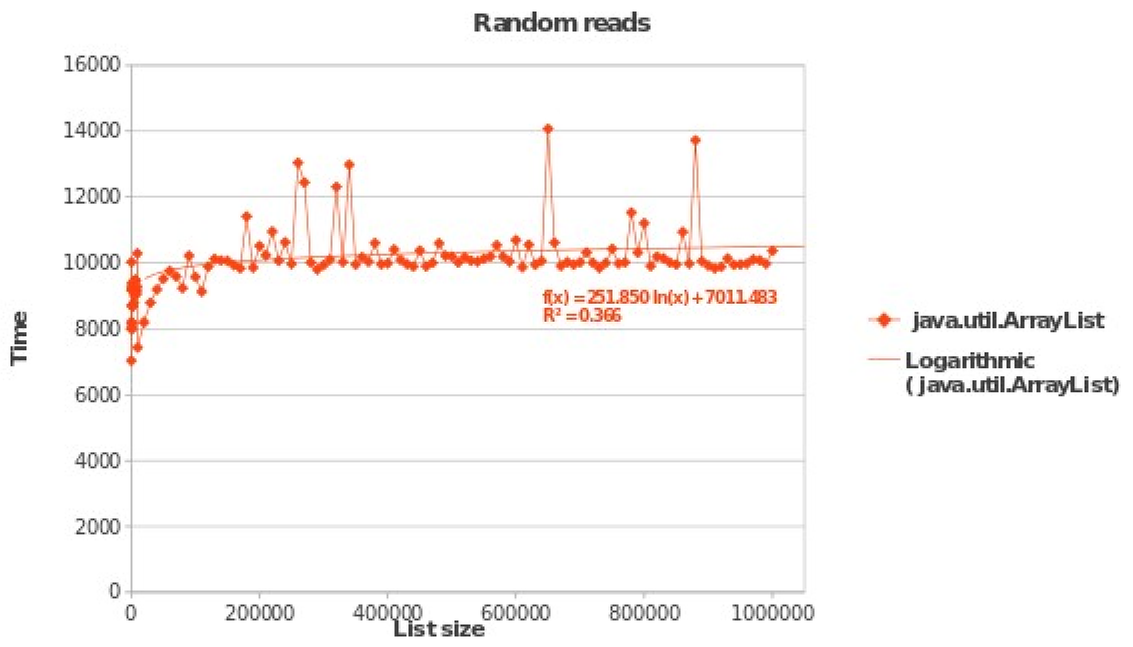

Figure 19: Test 3, random list read test results, showing only ArrayList.

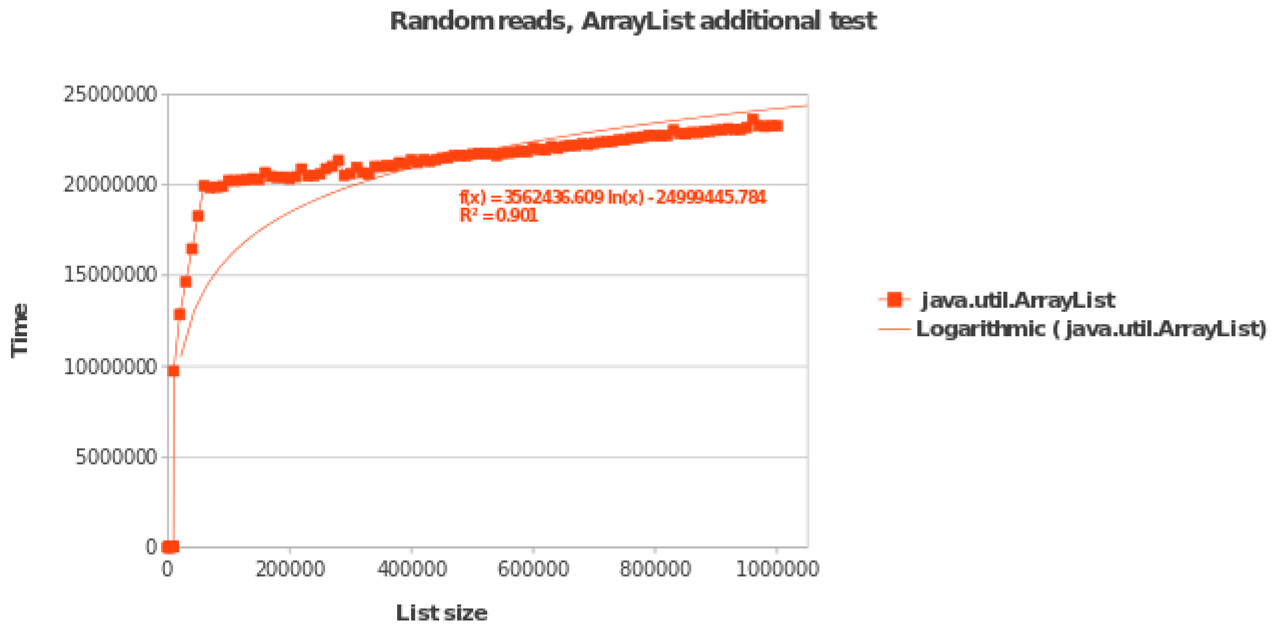

Figure 20: Test 3, random list read test results, ArrayList additional test.

The test was modified slightly to perform the calculation using the returned elements (total of returned elements was calculated). The results (shown in the Figure 20) indicate that the hypothesis may be true but additional measurements of both sets with different test methodology are required before final conclusions about performance difference can be made.

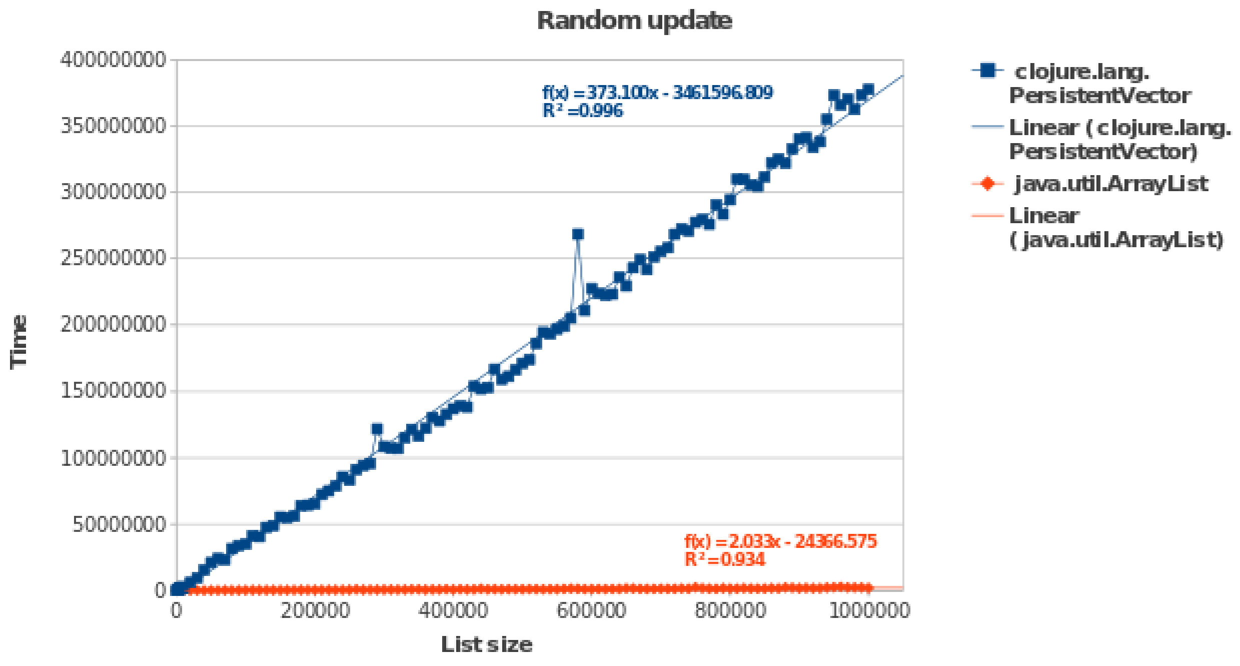

Figure 21: Test 4, random list update test results. Clojure library class clojure.lang.PersistentVector is compared against Java library class java.util.ArrayList.

Graphs in Figure 21 show test result from the random update test. For every measurement the list is filled with n elements and the time to perform random updates of the list (using the index) is measurement for every n. This test shows how much time it to randomly change all the elements in the list of the size of n.

In this diagram both graphs show linear growth tendency (the values of the ArrayList is showed separately on the Figure 22). The measured performance of the clojure.lang.PersistentVector roughly 2 orders of magnitude (!) less than performance of java.util.ArrayList. This big difference can be explained by PersistentVector need to update the underlaying trie after every update while ArrayList update is only change in underlaying array.

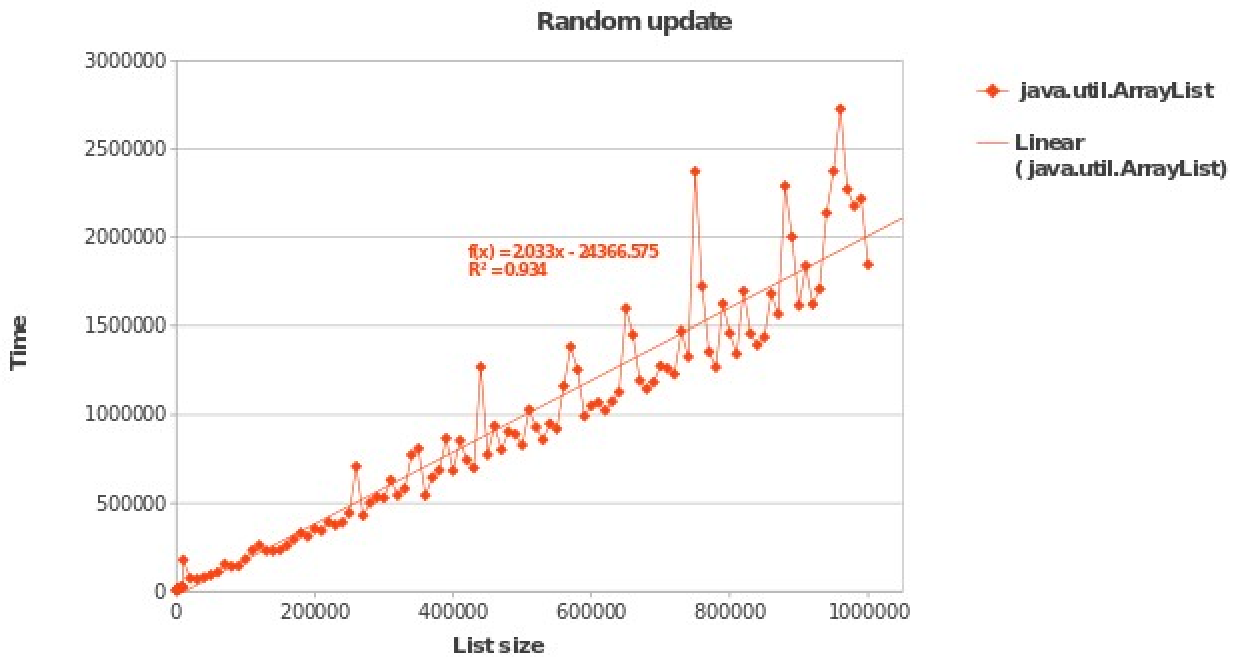

Figure 22: Test 4, random list update test results, showing only ArrayList.

4.3.7 Suggestions for persistent vector additional performance improvements

4.3.7.1 Broaden immutability definition

It is interesting to note that element added to the end collecting buffer (or tail buffer) would become out of the bounds of the previous version of the vector. An old version can not see and access the new elements even when the buffer for added elements is shared. Vice versa the new version can indeed white into shared buffer because it does not modify previous array elements, it just adds new item outside the bounds of visibility of the previous version.

When simple array is used for the buffer then this kind of behavior is allowed by the Java memory model because updates to separate array elements are independent. Of course when the new element is added by new thread then the old thread still holding the reference to the old version may not see this change but it is not relevant because it will be out of the bounds for it and can not be accessed. [3]

Conflict occurs when the initial version is used to add another element into it. When this operation would now share the buffer that is already shared by two previous versions, it would overwrite the change made by the first addition.

This shows that when such conflicts could be detected and avoided it may create opportunity for better performance. Because immutable data structures are intended to be safely accessed by different threads without additional synchronization then the thread safety must be handled within the vector itself.

4.3.7.2 Change control using AtomicBoolean

One opportunity to implement such feature is to add an additional boolean type flag that will be initially false and would be marked true when the new element is added by first time. This first addition can then share the buffer securely but a new buffer must be created when the flag examination determines that it is needed.

The flag reading and updating is a classical situation for the check and act bug. It must be ensured that no two threads could see the false value and try to update it to true at the same time.

One option to realize this is to use the AtomicBoolean class available in Java since version 5 (see Figure 23). This class allows thread safe updating a boolean variable with the new value and returning a previous value at the same time. Because this class uses special processor operators for this instead of locks then it should be theoretically perform better than doing the same operation with the synchronization. Despite of this it would add an additional overhead that was not presented before. But it would be still interesting to compare this against potential gain from removing the need to copy the array over and over again.

```
public class BitmappedTrie <E> {

      private final E[] tailBuffer;
      private final Object[] trie;
      private final AtomicBoolean stale; // thread-safe mutable boolean

      public  BitmappedTrie(E newElement, E[] tailBuffer, Object[] trie) {
                  stale = new AtomicBoolean(false);
                  tailBuffer = pushTail(trie, tailBuffer);
                  tailBuffer += element;
      }

      public BitmappedTrie <E> addElement(E e) {

                  if (stale.getAndSet(true) {
                        return new  BitmappedTrie(e, trie, tailBuffer.copyOf());
                  }
                  else {
                        return new  BitmappedTrie(e, trie, tailBuffer);
                  }
      }
}
```

Figure 23: Immutable vector optimization 1 pseudo code

4.3.7.3 Identify different threads

There exists another potentially better performing method to guarantee that the flag is handled safely. Namely the Java runtime provides an identifier of the the current running thread through *Tread.currentThread* interface. If this identifier is recorded during the creation of new vector version then it would be possible to verify if the request to add the new element is coming from the same thread or from different one.

If the thread is different from the creating thread then creation of the new copy of the buffer should be forced, otherwise the flag can safely examined to determine if the current buffer can be reused because access to the flag is limited to the one thread only (see Figure 24).

```
public class BitmappedTrie <E> {

      private final E[] tailBuffer;
      private final Object[] trie;

      // non thread-safe mutable boolean primitive value accessed
      // in thread-safe manner
      private boolan  stale;

      private final creatorTread;

      public  BitmappedTrie(E newElement, E[] tailBuffer, Object[] trie) {
                  stale = false;
                  creatorThread = Thread.currentThread();
                  tailBuffer = pushTail(trie, tailBuffer);
                  tailBuffer += element;
      }

      public BitmappedTrie <E> addElement(E e) {

                  if ( creatorTread != Thread.currentThread()) {
                        return new  BitmappedTrie(e, trie, tailBuffer.copyOf());
                  }
                  if (stale) {
                        return new  BitmappedTrie(e, trie, tailBuffer.copyOf());
                  }
                  else {
                         stale = true;
                        return new  BitmappedTrie(e, trie, tailBuffer);
                  }
      }
}
```

Figure 24: Immutable vector optimization 2 pseudo code

This would provide potential performance gains when the vector is first created fully by one thread and then shared or when only one or few threads are updating the vector and others are only reading it.

The same optimizations can be applied to the buffer push operation where the buffer is pushed into trie after it is full. This would allow creation of the whole vector without additional intermediate structures.

Five different versions of data structure comparable to Clojure vector were created to examine this hypothesis. The base version (BitmappedTrie1) implements the add element (add to the end of the list) and the get element by index functionality similarly Clojure vector implementation. One enchanted version (BitmappedTrie2) implements the version check using the AtomicBoolean and a second version (BitmappedTrie4) uses the thread identifier verification. There are two additional versions (BitmappedTrie3 and BitmappedTrie5 accordingly) based upon these two to how much speedup additional tree update optimization provides (table comparing different test can be seen on the Figure 25).

	Uses tail buffer to defer change in trie	Shares tail buffer with the previous version	Uses AtomicBoolean to control stale state	Checks for stale state in one thread,gives other threads new buffer	Shares parts of the trie
BitmappedTrie1	X				
BitmappedTrie2	X	X	X		
BitmappedTrie3	X	X	X		X
BitmappedTrie4	X	X		X	
BitmappedTrie5	X	X		X	X

Figure 25: Test versions

4.3.7.4 Test results

Firstly the effects of sharing the tail buffer with the previous versions was tested. Figure 27 shows that the Clojures persistent vector implementation and test class without optimizations show similar performance. The buffer sharing optimized version that uses *AtomicBoolean* instance to track the changes (BitmappedTrie2, red) shows slight improvement despite added additional overhead. The other optimization with thread confinement (BitmappedTrie4, green) performs well in this single threaded test providing roughly 2 times improvement compared to the difference between the PersistentVector and *ArrayList* (violet).

AtomicBoolean optimization (BitmappedTrie2, red) test show that this optimization does not bring considerable performance gain and adds risk of the thread contention in case of multi threaded access.

do not show measurable performance difference

Results for tests to measure possible performance improvements from additional trie sharing (see Figure 26) show that the expected performance gain from additional trie sharing did not realize.

AtomicBoolean optimization (BitmappedTrie2, bues) does not differ from its optimized version (BitmappedTrie3, red). As well optimization with thread confinement (BitmappedTrie4, yellow) does not differ from its optimized version (BitmappedTrie5, green).

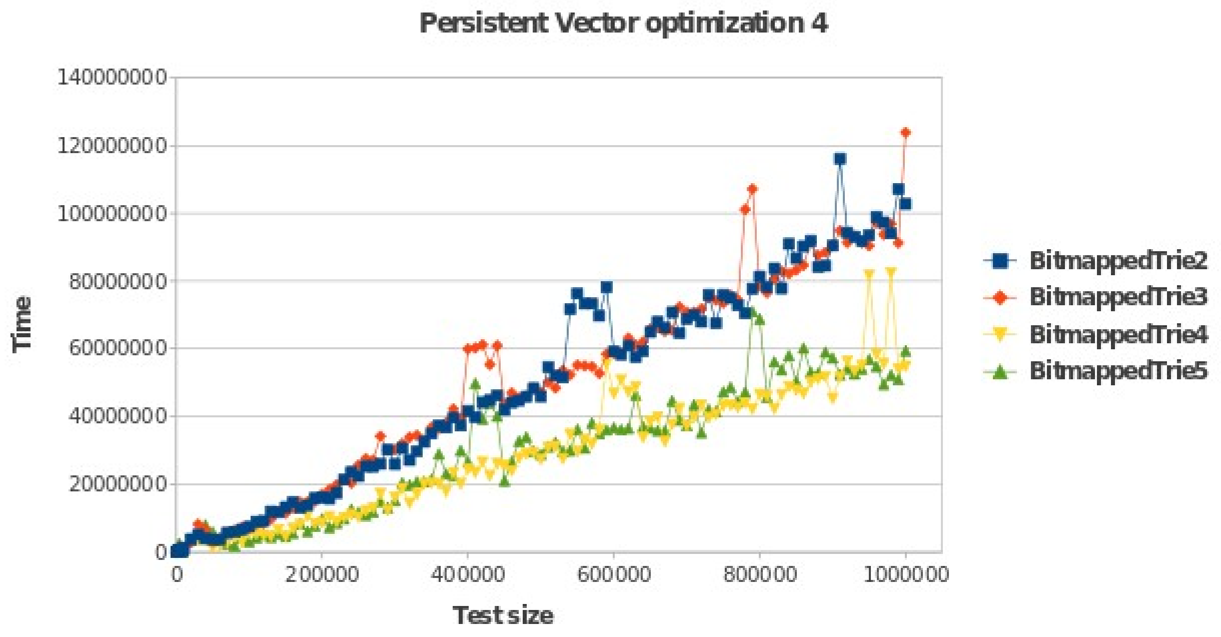

Figure 26: Persistent vector optimization, effects of additional trie sharing.

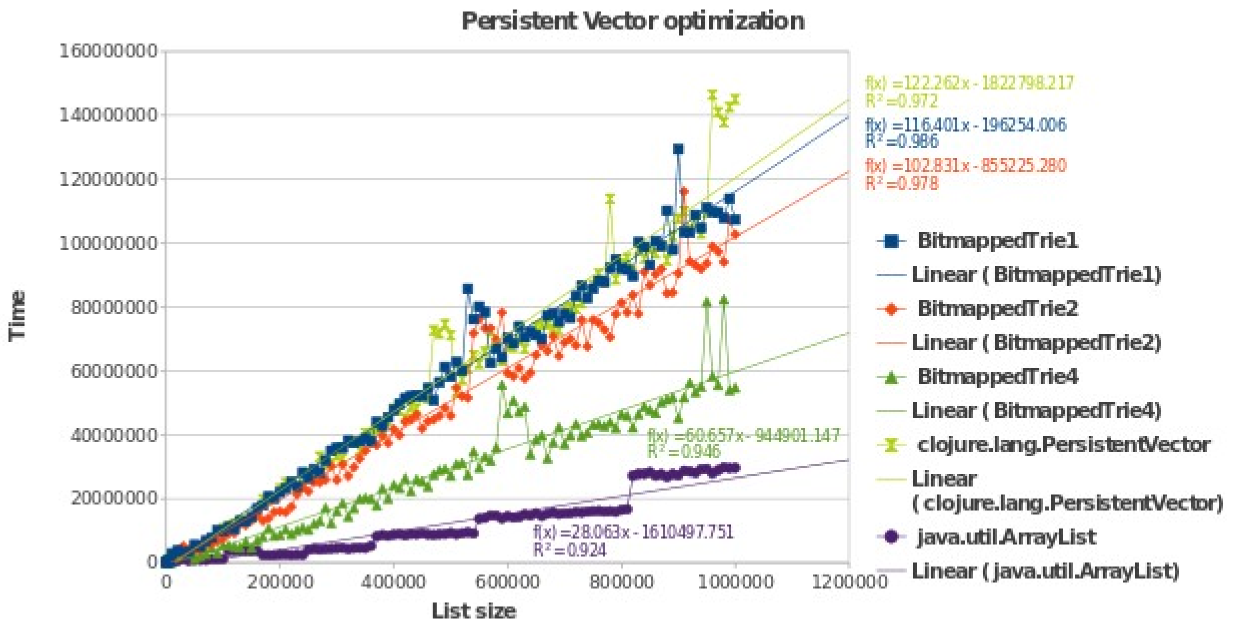

Figure 27: Persistent vector optimization, tail buffer sharing.

4.3.7.5 Immutable list building with mutable companion

There is another possibility to improve the performance of the vector creation suggested by the immutable classes design guidelines. Immutable class can be companioned with the mutable counterpart that can be turned into immutable one after set of performance critical operations are completed.

Clojure follows these recommendations by providing a so called transient vector implementation (that implements *ItransientVector* interface). This is a version of vector that has similar internal structure as the persistent immutable version but the changes are mutated internally instead of creation of the immutable intermediate versions.

Following the same initial principles a different approach was examined. Java collection class *ArrayList* organizes the list elements into one uniform array that is enlarged when it reaches its limits. It is possible to add another layer where elements are not directly inserted into array but are first collected into buckets and then inserted into array with the bucket when it is full. [36] These principles were used to build the *FastArrayList* test class.

One benefit of this is that when the bucket size is the same as the vector branching factor then it is possible to directly reorganize the buckets into vector representation.

Only care should be then taken to not overwrite the elements in buckets after the list is turned into separate tree form. When the initial list is limited only to one thread then this can be simply archived by an additional array where the status of each buckets is recorded. After conversion all the buckets must be marked as stale and the check should be performed before each element update. When element in the stale bucket is updated, a full copy of the bucket must be first created before the update.

BitmappedTrie1 test class was extended to support building itself from FastArrayList using method described here.

An additional sequential fill test was performed where the time to add new elements into mutable list first and then convert it into immutable version was measured.

4.3.7.6 Test results

Surprisingly the performance of the list building immutable list with the transient helper class can exceed the performance of the vanilly ArrayList (see figures Figure 28 and Figure 29).

Figure 28 shows the performance comparison between two different transient build strategies (clojure.lang.TransientVector (yellow), BitmappedTrie1 (darker blue, based on FastArrayList)) and regular immutable list building (clojure.lang.PersistentVector (brown)) and ArrayList baseline (green). Thread confined optimization (BitmappedTrie4) and FastArrayList (red) are included for comparison and scale.

Figure 29 shows the performance comparison between two different transient build strategies (clojure.lang.TransientVector (yellow), BitmappedTrie1 (darker blue, based on FastArrayList)) and ArrayList baseline (green). FastArrayList (red) is included for comparison and scale.

Result shows that transient creation of immutable vectors can provide similar performance as ArrayList and even outperform it at bigger list sizes.

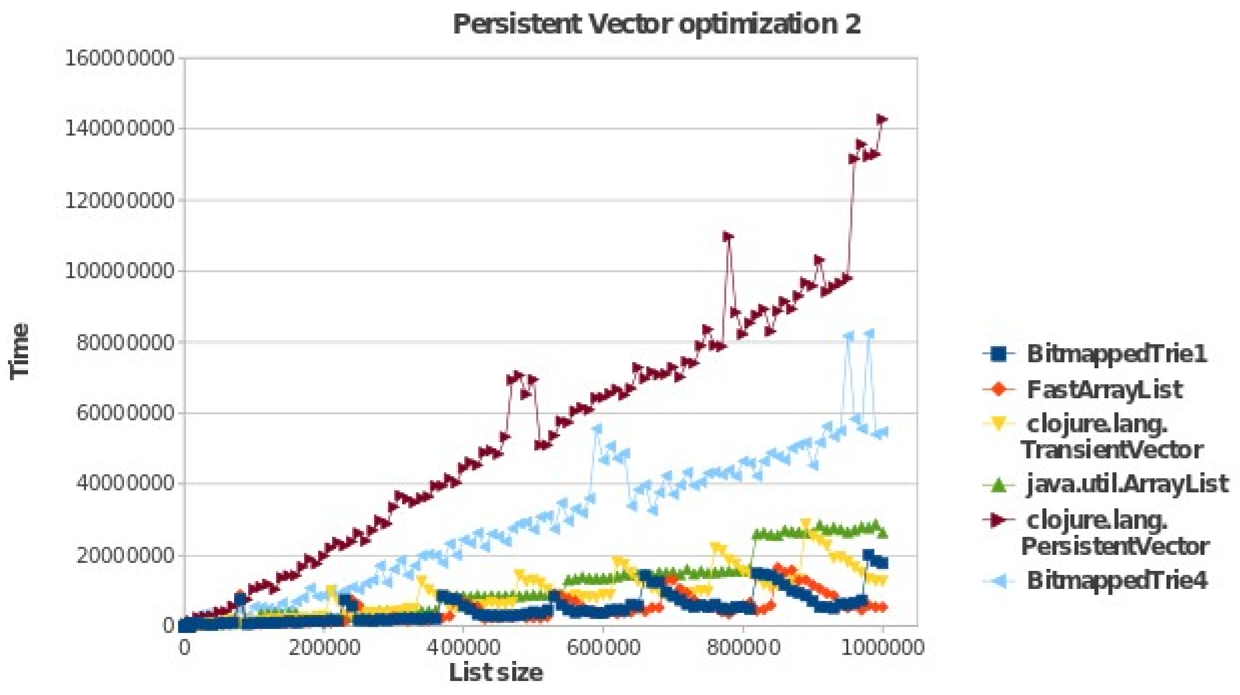

Figure 28: Persistent vector optimization, creating immutable instances from mutable data.

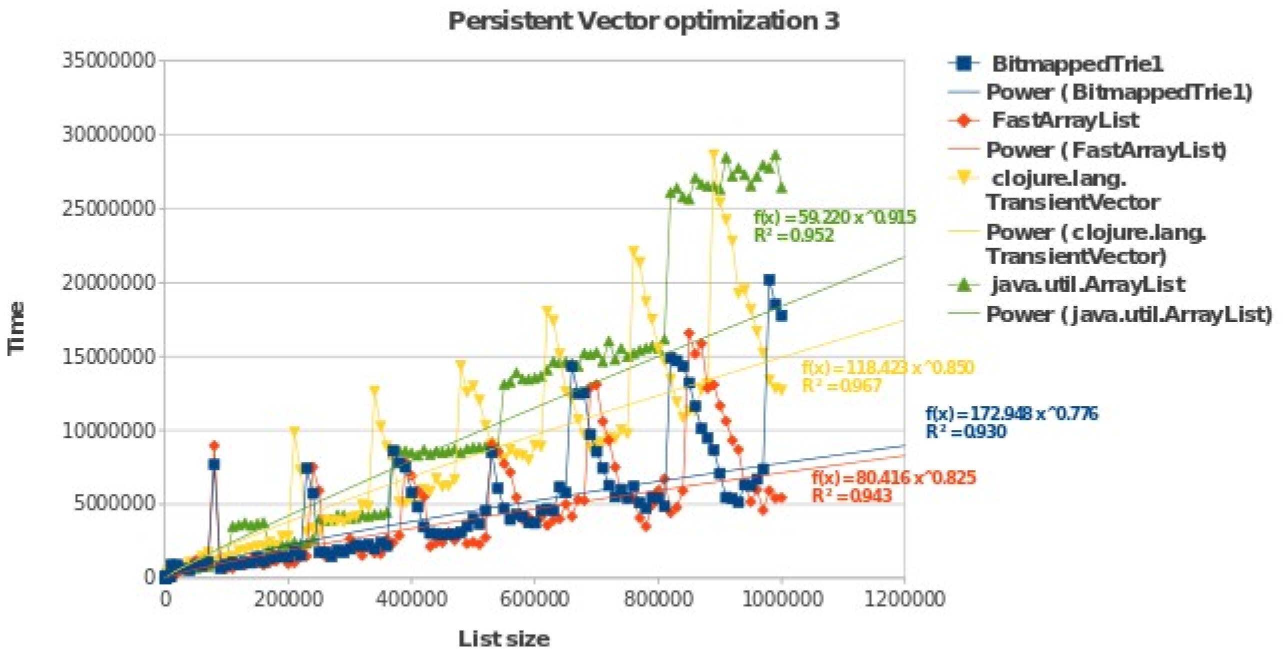

Figure 29: Persistent vector optimization, creating immutable instances from mutable data, closer view.

4.4 Sharing data between threads

4.4.1 Actor model

Actor model allows sending asynchronous messages to the subroutines called actors. Actors act on messages by returning replies to the messages or by sending additional messages to another actors (including themselves). [37]

Some actors implementations allow messages sent to actors block until the reply is prepared and returned from the actor. This may be appear similar to the call of the synchronized method but there are following important differences.

1. The synchronized method will be executed inside the calling thread but the actors runs inside the separate thread different from every caller[14].
2. While access to the synchronized method is queued, once passed to queue the synchronized method can directly access and change objects internal state. Only actor thread can access actors state directly. Callers can only access the actors message queue.
3. Call to the synchronized method will block every other thread during full call duration. Calls to the actors are only synchronized at the message queue allowing other callers to proceed even when there is only caller waiting for the reply.
4. It is relatively easy to combine two synchronized method calls into one atomic operation by preliminary synchronization. Synchronization asynchronous actors is more complicated.

Actor model can be explaining with the producer consumer model where there is one consumer and many producers separated by the message queue. Actor is the consumer and the callers are the producers.

As there are always at least two threads involved in the actor model then it becomes apparent that the internal state of the data shared between threads should not be modified after it is sent to the actor or returned from it.

When using mutable data structures inside the message then there are few possible ways to accomplish this.

1. One ways is to use thread safe classes that can be mutated securely. While this would be safe when properly implemented it would defeat the purpose of the actor model due added complexity and requirement to synchronize the state outside the hand over process.
2. Another way is an agreement inside the development team that the handed over data is not changed by sending party. While possibly achievable within small project and limited problem range it would become difficult to maintain when for example the purpose of the actor is to take in a collection, apply changes onto it and hand it over to another actor.
3. Finally it would be possible to create a copy of data before handling it over but this solution would not scale well when handed over data becomes large.

This shows that for actor model to work as expected and scale well, an immutable data structures are

14 Except from actor itself when actors send message to itself recursively.

needed if not inevitable.

Actors are ofter realized with one method that processes messages sent to the actor within one switch statement. This approach may feel foreign to programmers coming from the object oriented programming backgrounds. Java replection API option to create object proxis allows to provide a mechanism that automates message dispaching. Actors can be designed as regular classes with regular methods.

When new actor is created from the class by the actor framework then the framework can provide the actor creator with the proxy class that would turn the calls on the methods into messages. The framework then create a message dispacher on the actor side that would read the messages from the queue and turn them into message calls on the actor class. This affords a more familiar object oriented interface to the actors. [37]

4.4.2 Agents

Clojure offers a more functional concept of the actors called agents. [37] When in the actor model the code to handle the incoming message is coupled with the actor, Clojure separates the message handling from the agent and lets the agent only store the data, bind the date to the incoming message handlers and execute messages inside its own thread similarly to the actors.

Agents are good for asynchronous changes where all the relevant data is held inside one agent but need an additional coordination to change state atomically across multiple actors.

4.4.3 Software transactional memory

Software transactional memory (STM) provides a method to access multiple memory locations atomically. Differently from locks that employ a pessimistic approach where every possible memory location must be guarded beforehand, in STM method the changes are calculated locally without taking locks in hopes that the state does not change during the calculation. When the calculations are ready only then the locks are taken for the brief time to verify that the initial state is not changed and to write changes into memory when it has remained the same. When the state has changed then the process must be repeated with the updated initial state.

There are few possible methods how to implement STM. [38] STM support can be implemented as a library that can be used independently from the direct language support, it can be integrated directly into language with related keywords supported by the compiler or it can be introduced on the virtual machine level what examines special annotations inside the code.

Clojure support for STM can be classified rather as library implementation. While Clojure macro system allows to integrate library code naturally into language, it does not provide compile time code verification to control if transactions are applied properly[15]. This control is performed during the run time inside the library code. Because Clojure STM is implemented as a Java class library then it is directly available to use inside the Java code. [37]

Purpose and function of the software transactional memory is often explained through the analogy with the transactional databases. While this analogy is true to certain extent then the very principle of

15 Every change to the variable managed by STM must be enclosed inside transaction. When STM managed variable is accessed outside of the transaction it will result in runtime exception.

Clojure software transactional memory can be explained by much more primitive and basic analogy.

4.4.3.1 Compare and swap

Modern processors incorporate an instruction that allows to update one word size (32 bit or 64 bit depending on processor architecture) value within one processor instruction without compromising its integrity and at the same time compare its expected (before update) value and reset the update process when actual value before the update is not the same as the expected value. [7]

This operation is usually referred as compare and swap (CAS) and it would allow concurrent change of one word size value without taking locks. This includes values that would fit inside one word including references to the complex objects.

CAS provides an alternative to the lock based synchronization and allows to build optimistic lock free algorithms. In addition it can offer better performance than locks, especially under low contention. [7]

Java provides support for this operation within *java.util.concurrent.atomic* package. When underlaying platform does not support this operation explicitly then it is emulated by using locks to retain the Javas portability. [7]

While it might sound very limited, quite complex concurrent behavior can be implemented on top of this method. ([7], [39])

Common pattern to modify the value with CAS is

1. read current value and store it,
2. calculate new value,
3. compare current value with the stored value and try to swap with the new value when it has not been changed,
4. go to step 1 when unsuccessful.

It is apparent that it might be necessary to repeat the step 2 several times (see Figure 30[16]). It concludes from this that step 2 should not contain functionality with side effects that require recovery when the step 3 is not successful. It is also visible that when step 2 can be contained it would be possible to create an reusable pattern that automates this kind of updates.

It would be also usable when this kind of behavior could be extended to multiple values and multiple occurrences could be nested into one uniform update with single success and failure point.

In its essence this is exactly what Clojure STM does (see Figure 31[17]).

4.4.3.2 STM comparison with CAS

Beside atomic primitives like *AtomicBoolean*, *AtomicInteger*, Java provides a class *AtomicReference* (see Figure 30). This class implements a holder class for any Java object by keeping the reference to the object. In many senses it can be viewed as advanced *volatile*[18] variable [7] because the reference is

16 It is also worth to examine the source code of the Java atomic classes.

17 While it is very expressive in Java, analogous code in Clojure is much more terse.

18 Keyword *volatile* guaranties only the memory visibility but does not guarantee atomicity of multiple operations.

marked with *volatile* keyword but in addition to common *get* and *set* functionality it provides methods to update this reference atomically. The most interesting is the *compareAndSet* method that accepts two parameters *expect* and *update* and it atomically updates the reference with the new reference sent in *update* parameter when the old reference equals[19] with *expect* parameter. [18]

To implement a very simple STM we could write a *Ref* class that extends the functionality of the *AtomicReference* class by caching the initial value and changes locally for every thread and by providing a method to try to make the final result visible globally using *AtomicReference compareAndSet* method.

```
public class BookManager {

      private final AtomicReference<Book> bookRef;

      public  BookManager(String authors, String title, int edition) {
                  Book book = new Book(authors, title, edition);
                  bookRef = new AtomicReference<Book>(book);
      }

      public Book changeAuthors(String authors) {

                  for (;;) {
                        Book previousBook = bookRef.get();

                        Book newBook = new Book(
                              authors,
                              previousBook.getTitle(),
                              previousBook.getEdition());

                        if (bookRef.compareAndSwap(previousBook, newBook)) {
                              return newBook;
                        }
                  }
      }
}
```

Figure 30: Pseudo code example of the CAS usage

In addition to that we could write a *TransactionManager* class that executes and controls the CAS cycle by providing a method that accepts a simple Java *Callable* interface instance[20]. An additional method can provide a method to register as a transaction participant. The *Ref* class can then register itself within the *TransactionManager* class when the get or set methods are called.

Of course because current processors do not support directly this kind of multiple variable CAS, it should be implemented using locks. Still, the Clojure STM provides many benefits over using manually managed locks.

19 References comparison (== operator) is used and not the method *equals*.
20 This implementation could be provided as inline class directly where the method is called.

```
import static clojure.lang.LockingTransaction.runInTransaction;

public class BookManager {

      // Clojure does not support generics but it can be
      // added with a wrapper class.
      private final Ref<Book> bookRef;

      public  BookManager(String authors, String title, int edition) {
            Book book = new Book(authors, title, edition);
            bookRef = new Ref<Book>(book);
      }

      public Book changeAuthors(String authors) {
            return runInTransaction( new Callable<Book>() {
                  public Book call() {
                        Book previousBook = bookRef.deref();
                        Book newBook = new Book(
                                    authors,
                                    previousBook.getTitle(),
                                    previousBook.getEdition());
                         bookRef.update(newBook);
                        return newBook;
                  }
            });
      }
}
```

Figure 31: Pseudo code example of the Clojure STM library usage inside Java

4.4.3.3 STM comparison with locks

One of the greatest advancement of the STM is probably the easy method to combine multiple independent operations into one uniform transaction. [37] For example with the *BookManager* class (in Figure 31) we could implement a *swapAutors* function simply by calling *changeAuthors* methods of two *BookManager* instances inside the transaction. When STM implementation supports nested transactions then we could just wrap an additional transaction around method calls. When available STM does not support nested transactions then we should change the *changeAuthors* method and remove the transaction from there (see Figure 32).

```
import static clojure.lang.LockingTransaction.runInTransaction;

[…]

public static void swapAutors(final Ref<Book> a, final Ref <Book> b) {

      runInTransaction( new Callable<Void>() {
            public Void call() {
                  Book pA = a.deref();
                  Book pB = a.deref();

                  a.update(new Book(pB.getAuthors(),
                                    pA.getTitle(), pA.getEdition());
                  b.update(new Book(pA.getAuthors(),
                                    pB.getTitle(), pB.getEdition());

                  return null;
            }
      });
}
```

Figure 32: Pseudo code for the swapAutors function.

Because locks in STM are taken in the same order over multiple transactions then the deadlocks are not possible (code in the Figure 32 is not able to deadlock). [27] Because locks are taken only for short period of time then it should also theoretically result the lock contention.

In theory the STM makes it more reliable to manage complex state that requires applying multiple operations atomically. Only thing programmer should do is to wrap atomic operations inside a transaction. Of course this does not free the programmer from identifying the parts of the application that must be executed atomically and it is therefore still possible to produce simple check and act type bugs and other types of illegal interleaving.

Like it is not a good practice to start parallel threads and update the shared variable with the direct locks usage (with synchronized block, locks or atomic variables) when there are other options, it is also not a good practice with the STM. Updating the same memory address from different threads will cause contention because the threads must proceed sequentially during the update. [37]

While STM eliminates the possibility of the deadlock, it may produce another liveness hazard – a livelock. Livelock occurs when thread can not make progress due continuously repeating an operation that fails. [7] When many threads try to update a shared value then some change requiring a longer calculation may never succeed because there is always a faster transaction that changes the value before the longer running transaction is able to update it.

Clojure STM tries to so solve such situations by using barging. [27] In barging the slower running older transaction is allowed to continue causing the newer and faster transactions to retry instead. When barging still does not allow the older transaction to complete then the transaction causing the problem is terminated with a runtime error. This would allow the programmer(s) to investigate what could cause the problem.

Another problem is that because the code inside the transaction could be executed several times repeatedly it should not have side effects outside changing the transactional references. Clojure STM resolves this by giving a option for an additional callback function what is executed when the transaction completes. The similar functionality allows to call agents from inside the transaction. Calls to the agents from the transaction are collected and executed only after transaction completes.

It is important to note that Clojure STM does not manage content of the references but only references themselves. Therefore letting Clojure STM manage the references to the mutable data structures would render Clojure STM at least dangerous to use if not unusable.

4.5 Summary of Clojure concurrent programming

Probably the foremost important quality of the Clojure is that it prevents errors caused by subtle nuances of the Java memory model. It is not possible to unintentionally share a variable. Variables are either stack or thread confined and special effort must be taken to share them among multiple threads.

Clojure relies heavily on immutable data types to gain its goals. Immutable data types make it possible to restrain the problems to the reference level. While complex immutable data types do not perform always the same as mutable analogues, they make the sharing of complex data structures considerably simpler.

5 Summary

Concurrent programming in Java inherits most of its problems from the direct incorporation of the shared memory model. Because it is possible in Java to access shared memory without properly applied mutual exclusion, it can produce hard to detect software bugs. Moreover Java method of mutual exclusion relies mostly on the locks that can introduce additional hard to detect problems. Most notoriously the program can contain a possible deadlock when lock acquisition is not correctly ordered. It is difficult to compose separate thread safe atomic operations into a new atomic operation using locks because an additional complex synchronization is required when combining multiple method calls. While concurrent programming has become much simpler since Java version 5, its new additions do not solve all of the conceptual problems.

A new Lisp inspired functional language Clojure that is implemented on top of the Java platform introduces a more limited ruleset for the data sharing between different threads. Most importantly it is not possible to unintentionally share the data between the threads in Clojure – all such operations must expressed explicitly. This approach can arguably reduces the set of possible programming errors.

Clojure offers two methods for the data sharing. It can be accomplished asynchronously with the agents or synchronously with either software transactional memory (STM) for operations that require updating multiple values in one atomic operations or with simpler atomic updates when only one values is shared.

Clojure software transactional memory provides syntactically simple method for combining multiple separate atomic operations into new atomic operation by simply wrapping given operations into a new transaction. Clojure STM can reduce programming errors further by using runtime verification to check that no updates are performed outside of the transaction. Still it does not free the programmer from correctly identifying set of the operations that should be executed atomically.

Clojure method of the concurrent programming relies heavily on the immutable data structures. Immutability lets it regard complex data structures as simple values whose state does not change outside of the control of the reference holder. Therefore it is important for Clojure to provide rich set of different data structures that follow these principles.

One of such data structures in Clojure is Persistent Vector from Clojure collections library. The internal working principles of this data type were explored. In summary it is a bit mapped trie with the high branching factor that allows possibility of the deferred additions into the end of the vector by collecting the new elements into a tail buffer before pushing them into the trie as a whole. Persistent Vector can share a bulk of its internal structure with the previous versions making it effective immutable data structure.

The actual performance of the Persistent Vector was evaluated. The findings show that compared to the Java collections *ArrayList* it can provide similarly performing addition and iteration operations. The update by index performs two orders of magnitude slower than analogue operation on *ArrayList*. The performance difference of lookup by index operation was not conclusively determined due probably JIT induced difficulty to measure *ArrayList* index lookups reliably. Performed measurements still allow to speculate that the performance difference of the index lookup operation between Persistent Vector and *ArrayList* is similar to the performance difference of the update by index operation.

Few additional performance enchantments were evaluated and it was concluded that it would be possible to improve the addition operation performance around two times when additional thread confined flag is used to allow further sharing of the tail buffer between different versions.

It can be argued that relatively good addition and iterating performance would allow to use Persistent Vector to solve a set of useful problems. For example the Persistent Vector can be used to load a list of the records from the database to be iterated over to build a web page based on that data.

Due hardness of proper performance testing of the parallel operations such tests were not included into this work. It can be suggested that testing the performance of sharing persistent vector between multiple threads is needed.

Clojure shows that it is possible to make concurrent programming relatively safer when a set of design principles are changed. It can be argued that difficulty of concurrent programming in Java does not improve unless its memory access principles are considerably reevaluated.

6 Muutumatud andmetüübid konkurentses programeerimises Cloure keele näite varal

Bakalaureusetöö

Kristjan Kelt

Resümee

Konkurentne programmeerimine keskendub probleemidele, kus erinevaid ressursse tuleb jagada mitmete lõimede vahel. Kõige lihtsamal juhul võib selleks olla protsessori arvutusressurss, kuid tänapäevased mitme tuumaga protsessorid lisavad probleemile lisamõõtme, kus valdavaks probleemiks saab mälu ühine konkurentne kasutamine.

Selle töö eesmärk on uurida konkurentses programmerimises esinevaid probleeme ja võimalikke lahendusi Java ja Clojure keelte näite varal pannes rõhku keeles Clojure kasutusele võetud uuendustele.

Töös leitakse, et konkrurentne programmeerimine Javas pärib enamiku oma probleemidest konkurentsete programmeerimise vahendite suhteliselt madalatasemiselisest lisamisest Java keelde. Enamik konkurentse programmeerimise probleeme Javas tuleneb ühismälu mudeli kasutuselevõtust. Kuna Javas on võimalik pöörduda ühismälu poole ilma korrektse vastastiku välistuseta, siis võib see põhustada raskesti leitavaid tarkvara vigu. Peale selle põhineb Java lahendus vastastikuks välistuseks enamasti lukkudel, mis võib luua raskesti leitavaid uusi probleeme.

Näiteks võib programm sisaldada tupikut, see on olukorda, kus programmi kaks lõime ootavad vastastiku võetud lukkude taga. Selle põhjuseks on ebakorrektne lukkude võtmise järjekord programmis.

Kasutades lukke on keeruline koostada mitmest eraldi seisvast atomaarsest operatsioonist uut operatsiooni, mis peab tagama mõlema eelneva operatsiooni ühise atomaarsuse. Vaatamata sellele, et Java versioon 5 muutis konkurentse programmeerimise märgatavalt lihtsamaks, ei lahendanud selle uued võimalused kõiki kontseptuaalseid probleeme.

Töö põhirõhk on keele Clojure uuendustel. Töös leitakse, et Lispist inspireeritud funktsionaalne Java platformil põhinev programmerimise keel Clojure pakub rohkem piiratud reeglistikku andmete jagamiseks mitme lõime vahel. Kõige olulisem on see, et ei ole võimalik jagada andmeid ettekavatsematult. Kõik andmete jagamised mitme lõime vahel peavad olema väljendatud tahtlikult. Selline lähenemine võib arvatavalt vähendada võimalik programmeerimise vigade hulka.

Clojure pakub kaks erinevad lahendust andmete jagamiseks. Andmeid võib jagada asünkroonselt kasutades agente või sünkroonselt kas kasutades tarkvaralisi mälutransaktsioone pöördumiste jaoks, mis nõuavad mitme väärtuse atomaarset muutmist või lihtsamaid atomaarseid uuendusi kui on vaja jagada ainult ühte väärtust.

Clojure tarkvaralised mälutransaktsioonid pakuvad lihtsa viisi kuidas kombineerida mitut eraldi seisvat atomaarset operatsiooni uueks tarvikuks. Selleks tuleb lihtsalt need operatsioonid ümbritseda uue transaktsiooniga. Clojure tarkvaralised mälutransaktsioonid võivad veelgi vähendada tarkvara tootmise vigu kuna need on võimelised programmi töö käigus kontrollima kas jagatud mälu poole pöördumine toimub transaktsiooni siseselt või mitte.

Töös jõutakse järeldusele, et eelnevale vaatamate ei vabaste see programeerijat korrektsest vajalike atomaarsete operatsioonide tuvastamisest programmi koodis.

Clojure lähenemine konkurentsele programmeerimisele põhineb enamjaolt muutumatute[21] muutujate kontseptsioonil. Kuigi muutumatut muutujat võib ekslikult pidada konstandiks, on sellel mitmeid huvitavaid omadusi. Muutumatud muutujad võimaldavad neid käsitleda keerukaid andmestruktuure lihtsate väärtustena mille olek ei muutu väljaspool viite haldaja kontrolli. Seega on oluline, et Cloure pakuks kasutuseks erinevaid andmeüüpe mis järgivaid neid põhimõtteid.

Üks sellistest andmestruktuuridest Clojures on Persistent Vector – Clojure suvapöördusega loend. Käesolevas töös uuriti selle andmestruktuuri töötamise printsiipi ning jõudlust. Kokkuvõtvalt võib öelda, et tegemist on *bitmapped trie* andmetüübiga, millel on kõrge hargnevustegur, mis võimaldab puhverdada lisamise operatsioone kogudes lisatavad elemendid esmalt nii öelda sabapuhvermällu ennem nende lisamist terviklikuna puusse. Persistent Vector andmetüübi ülesehitus võimaldab sel jagada oma sisemist struktuuri oma eelnevate versioonidega, mis teeb sellest tõhusa muutumatu andmetüübi.

Käesolevas töös uuriti Pesistent Vector andmetüübi tegelikku jõudlust. Leitud tulemused näitavad, et võrreldes Java ArrayList andmetüübiga võib see pakkuda sarnast jõudlust nii elementide lisamisel nimekirja lõppu kui ka nimekirja järjestikusel läbimisel. Elemendi positsiooni järgi uuendamise jõudlus on siiski kaks suurusjärku madalam kui see on andmetüübil ArrayList.

Elemendi positsiooni järgi pärimise jõudlusele ei õnnestunud anda selgepiirilist hinnangut tänu arvatavasti Java JIT kompilaatori poolt põhjustatud probleemidele ArrayList elemendi positsiooni järgi pärimise jõudluse hindamisel. Teostatud mõõtmised annavad siiski alust spekuleerida, et Persistent Vector'i ja ArrayList positsiooni järgi pärimise jõudlus on sarnane positsiooni järgi uuendamise operatsiooni jõudlusele.

Käesolevas töös pakutakse välja loetletud jõudluse paranduse ettepanekud, mille tulemusi analüüsiti. Järelduseks võib öelda, et Persistent Vector lisamise jõudlust on võimalik tõsta ligikaudu kaks korda kui osutub võimalikuks jagada selle lisamise sabapuhvermälu elementide lisamisel ühe lõime piires.

Võib arvata, et piisavalt hea lisamise ja läbimise operatsioonide jõudlus võimaldaks Persistent Vector andmetüüpi kasutada mitmete praktiliste ülesannete lahendamisel. Näiteks võiks seda kasutada andmebaasist laetud nimekirjast veebilehe koostamisel vahepuhvermäluna.

Korrektsete paralleeltestide koostamise keerukuse tõttu parallelljõudluse testid ei kajastu antud töös. Seega võib soovitada nende testide sooritamis edasiseks uurimisvaldkonnaks.

Kokkuvõtvalt jõuti töös järeldusele, et Clojure näitab, et on võimalik muuta konkurentne programmerimine suhteliselt turvaliseks kui loetletud disaini portsiibid on järgitud. Võib arutleda,et raskused Java konkurentses programeerimises ei vähene kuniks Java mälu kasutus ei ole kriitiliselt üle vaadatud.

21 Immutable

References

[1] P. McKenney. Memory Barriers: a Hardware View for Software Hackers. http://www.rdrop.com/users/paulmck/scalability/paper/whymb.2009.04.05a.pdf (Last viewed May 10. 2013).

[2] F. Long, D. Mohindra, R. Seacord, D. Svoboda. Java Concurrency Guidelines. http://www.cert.org/archive/pdf/10tr015.pdf (Last viewed May 10. 2013).

[3] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley. The Java Language Specification, Java SE 7 Edition. 2011.

[4] Wikipedia. Register allocation. http://en.wikipedia.org/wiki/Register_allocation (Last viewed May 10. 2013).

[5] Wikipedia. Mutual exclusion. http://en.wikipedia.org/wiki/Mutual_exclusion (Last viewed May 10. 2013).

[6] Wikipedia. Lock (computer science). http://en.wikipedia.org/wiki/Lock_%28computer_science%29 (Last viewed May 10. 2013).

[7] B. Göetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes. D. Lea. Java Concurrency In Practice. Addison-Wesley, USA, 2007.

[8] Intel Corporation. Managing Lock Contention: Large and Small Critical Sections. http://software.intel.com/en-us/articles/managing-lock-contention-large-and-small-critical-sections (Last viewed May 10. 2013).

[9] Wikipedia. Context switch. http://en.wikipedia.org/wiki/Context_switch (Last viewed May 10. 2013).

[10] Patrik Nordwall. Scalability of Fork Join Pool . http://letitcrash.com/post/17607272336/scalability-of-fork-join-pool (Last viewed May 10. 2013).

[11] Wikipedia. Readers–writer lock. http://en.wikipedia.org/wiki/Readers%E2%80%93writer_lock (Last viewed May 10. 2013).

[12] Oracle. The History of Java Technology. http://www.oracle.com/technetwork/java/javase/overview/javahistory-index-198355.html (Last viewed May 10. 2013).

[13] Kieron Murphy. So why did they decide to call it Java?. http://www.javaworld.com/jw-10-1996/jw-10-javaname.html (Last viewed May 10. 2013).

[14] Wikipedia. Java (programming language). http://en.wikipedia.org/wiki/Java_%28programming_language%29#History (Last viewed May 10. 2013).

[15] Wikipedia. Write once, run anywhere. http://en.wikipedia.org/wiki/Write_once,_run_anywhere (Last viewed May 10. 2013).

[16] T. Lindholm, F. Yellin, G. Bracha, A. Buckley. The Java®VirtualMachine Specification Java SE 7 Edition. http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf , 2011.

[17] Wikipedia. List of JVM languages. http://en.wikipedia.org/wiki/List_of_JVM_languages (Last viewed May 10. 2013).

[18] Oracle. Java™ Platform, Standard Edition 7 API Specification.

http://docs.oracle.com/javase/7/docs/api/ (Last viewed May 10. 2013).

[19] D. Ghosh. Scala Actors 101 - Threadless and Scalable. http://java.dzone.com/articles/scala-threadless-concurrent (Last viewed May 10. 2013).

[20] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: elements of reusable object-oriented software. Addison-Wesley, USA, 2012.

[21] J. Bloch. Effective Java Second Edition. Addison-Wesley. USA, 2008.

[22] M. Papathomas. Concurrency Issues in Object -Oriented Programming Languages. pp. 207-245, Tsichritzis D., Ed. Universite de Geneve. 1989.

[23] Yu Lin and Danny Dig. CHECK-THEN-ACT Misuse of Java Concurrent Collections. In Proc. 6th IEEE International Conference on Software Testing, Verification and Validation (ICST 2013), Luxembourg City, Luxembourg, 18-22 March 2013.

[24] Wikipedia. Java version history. http://en.wikipedia.org/wiki/Java_version_history (Last viewed May 10. 2013).

[25] J. Manson. What Volatile Means in Java. http://jeremymanson.blogspot.com/2008/11/what-volatile-means-in-java.html (Last viewed May 10. 2013).

[26] Rich Hickey. Clojure homepage. http://clojure.org/ (Last viewed May 10. 2013).

[27] C. Emeric, B. Carper, C. Grand. Clojure Programming. O'Reilly, USA, April 2012.

[28] P. Morin. Open Data Structures (in Java). http://opendatastructures.org/ods-java.pdf (Last viewed May 10. 2013).

[29] C. Okasaki, A. Gill. Fast Mergeable Integer Maps. pp. 77-86 of Workshop on ML, USA, 1998.

[30] K. Krukow. Understanding Clojure's PersistentVector implementation. http://blog.higher-order.net/2009/02/01/understanding-clojures-persistentvector-implementation/ (Last viewed May 10. 2013).

[31] Rich Hickey. PersistentVector source code. https://github.com/clojure/clojure/blob/master/src/jvm/clojure/lang/PersistentVector.java (Last viewed May 10. 2013).

[32] M. Fogus, C. Houser. The Joy of Clojure: Thinking the Clojure Way. Manning Publications, USA, 2011.

[33] M. Naftalin, P. Wadler. Java Generics and Collections. O'Reilly, USA, 2006.

[34] K. Kelt. Immutable data types for concurrent programming on basis of Clojure language tests source code. https://github.com/kristjankelt/Clojure_Immutable_Tests/ (last modification date May 13. 2013).

[35] Amazon Web Services, Inc.. Amazon EC2 Instance Types. http://aws.amazon.com/ec2/instance-types/ (Last viewed May 10. 2013).

[36] Javolution. Source of Javalution Fasttable class. http://grepcode.com/file/repo1.maven.org/maven2/javolution/javolution/5.5.1/javolution/util/FastTable.java (Last viewed May 10. 2013).

[37] V. Subramaniam. Programming Concurrency on the JVM. Pragmatic Bookshelf, USA, 2011.

[38] M. Mohamedin, B. Ravindran. ByteSTM: Virtual Machine-levelJava Software Transactional

Memory. http://www.ssrg.ece.vt.edu/papers/transact2013_submission_8.pdf (Last viewed May 10. 2013).

[39] Dr. Cliff Click. 2007 Java One Conference: A Lock-Free Hash Table. http://www.azulsystems.com/events/javaone_2007/2007_LockFreeHash.pdf (Last viewed May 10. 2013).