Benchmarking Contention Management Strategies in Clojure's Software Transactional Memory Implementation

Master's Thesis

Peder R. L. Nielsen Patrick T. Kristiansen

Aalborg University

Computer Science Department

Database and Programming Technologies

Spring 2010

Department of Computer Science
Aalborg University

Selma Lagerlöfs Vej 300
DK-9220 Aalborg Øst
http://www.cs.aau.dk

Title

Benchmarking Contention Management Strategies in Clojure's Software Transactional Memory Implementation

Research group

Database and Programming Technologies

Project term

SW10, Spring 2010

Project group

d608a

Supervisor

Bent Thomsen

Abstract

Lock-based concurrent programs are difficult to write and prone to a variety of well-known errors. Software transactional memory (STM) poses an alternative to locks, promising to make concurrent programming easier. One of the few languages that have included STM from an early stage is Clojure, a Lisp dialect for the Java Virtual Machine.

In our work, we benchmark the contention management strategies Aggressive, Priority and Karma in Clojure's STM. Our results show that the choice of strategy has significant impact on performance, but that there are only a few specific cases where it is an advantage to use any of those tested over Clojure's original. None of these cases warrant a replacement of the strategy that Clojure already uses.

Since we only investigate a limited selection of the strategies suggested in STM research, future work is needed to find whether there exist other strategies that surpass Clojure's.

Authors

Peder R. L. Nielsen and Patrick T. Kristiansen

Preface

The pitfalls of conventional concurrent programming are notorious and many. Sadly, the types of applications programmers need to write in the future are increasingly concurrent. In fact, as more and more applications move to the World Wide Web requiring that single applications handle many users, concurrent programming is becoming a necessity. In other cases, we are faced with the choice of either parallelizing our programs, making them concurrent, or accept that they will not execute any faster.

Concurrent programming is difficult primarily because of the insufficiency of the tools we use. Locks are the prevalent mechanism used to prevent race conditions in accessing shared memory, but only experts use them correctly. History has shown us many times the problems with locks.

Software transactional memory (STM) is the alternative to locks we have chosen to investigate in the work presented here. Specifically, we have chosen to work with Clojure's STM, because Clojure offers an interesting combination of features that seems to make software transactional memory viable in many situations. Moreover, this combination of features appears to be unique and has not been the subject of any academic literature that we know of.

Intended readers. Our thesis requires that the reader possesses basic knowledge about concurrent programming, functional programming and computer science. Even though we treat a relatively narrow subject, we have included chapters that give the reader the prerequisites to understand the subject area. We do, however, require that the reader has a basic ability to interpret the diagrams we present as the result of our work.

Furthermore, this thesis is for readers interested in getting an impression of Clojure's specific blend of functional programming, persistent data structures and software transactional memory. Besides evaluating different contention managers in Clojure's STM, we give an overview of Clojure the language and Clojure's existing STM implementation.

Acknowledgements. We are very grateful to our supervisor Bent Thomsen for his insights, suggestions and thorough reviews of this work. Furthermore, we thank the friendly people on Clojure's IRC channel that have answered many of our questions, especially Rich Hickey, the author of Clojure.

Contents

Preface i

1 Introduction 1
1.1 Problem and Method . . . 2
1.2 Contributions . . . 3

2 Software Transactional Memory 4
2.1 Origins . . . 4
2.2 Design Issues . . . 6
2.3 Multiversion Concurrency Control . . . 9

3 Clojure 11
3.1 Language Introduction . . . 11
3.2 STM Implementation . . . 15
3.3 Clojure in the STM Landscape . . . 21
3.4 Selected Contention Management Strategies . . . 22

4 Benchmarking 25
4.1 Micro or Macro . . . 27
4.2 Metrics . . . 28
4.3 Methodology . . . 29
4.4 Benchmark Suite . . . 31

5 Results 34
5.1 Discussion . . . 40

6 Conclusion 41

7 Future Work 43

Bibliography 44

A Benchmarks Source 49

B Benchmark Tools 57

Chapter 1

Introduction

Parallel computation is no longer restricted to supercomputers. Practically every new computer system bought by consumers have CPUs with two or more cores. There is good reason for that; the speed of individual CPU cores are no longer increasing at the pace they used to. To make computation faster, it must be parallelized. Parallelization demands concurrent programming.

Another tendency that demands concurrent programming is the development of an increasing amount of software for the World Wide Web. New web browsers allow development and execution of advanced applications that would previously have needed installation on the computer they were used. Such web applications, however, often need to handle large amounts of simultaneous users—users that expect the web applications to behave correctly. Whatever the motivation, concurrent programming is known to be notoriously difficult.

Locks are the prevalent mechanism for synchronizing access to shared memory and avoiding race conditions in concurrent systems. Unfortunately, history and literature is filled with evidence of the problems that can arise when locks are used. More specifically, locks are often the cause of deadlock, livelock, priority inversion and convoying. Some of the problems merely impact the potential performance gains of parallelization, while others lead directly to incorrect programs—the worst kind of programs.

An abundance of possible solutions exist to make concurrent programming less difficult. Some of these suggest replacing our conventional programming languages with ones that employ different paradigms, such as Erlang's [4] implementation of the actor model [27]. Others have suggested algorithms that employ non-blocking synchronization, i.e., synchronization that do not use locks, but still guarantee that race conditions do not occur. The simplest form of non-blocking synchronization is the primitive compare-and-swap operation (or similar) found in many machine architectures, which is essential in the implementation of more complex non-blocking algorithms.

The focus of our thesis is *software transactional memory* (STM), an alternative to conventional locks. Several implementations employ non-blocking synchronization [44, 25, 24, 22], although there is evidence that lock-using implementations are more efficient [15, 13]. It allows programmers to access shared

memory transactionally, i.e., with the well-known transaction semantics seen in many database systems. Shavit and Touitou published the first article [44] on the subject, and a flurry of research has been published since, resulting in the identification of a wide range of design issues. We give an overview of the area in Chapter 2.

The canonical example of using software transactional memory is a bank account scenario. Say we have two bank accounts A and B. Transferring money from A to B involves two operations: first withdrawing m amount of money from A and then depositing m amount of money on B. These two operations together must be atomic; either they both happen or none of them happen. To reflect this requirement, many suggestions for language support for software transactional memory adds an *atomic* keyword. Here is a Java-like example of the scenario just described:

```
void Transfer(Account A, Account B, double m) {
  atomic {
    Withdraw(A, m);
    Deposit(B, m);
  }
}
```

This thesis is focused on a specific STM implementation, namely the one found in *Clojure* [28]—a functional programming language with Lisp syntax that runs on the Java Virtual Machine.[1] The language is one of the few that has been designed to include software transactional memory from an early stage.[2] Clojure's combination of functional programming, persistent data structures (see Section 3.1.1), and software transactional memory with MVCC (see Section 2.3) appears unique. Furthermore, while Clojure has been the subject of a case study of multi-core parallelization [32], it seems that no or very little academic literature has been published on Clojure's STM. This has been a major motivation behind the work we present here.

1.1 Problem and Method

One important aspect of STM designs is *contention management*. Briefly stated, contention management deals with strategies that decide what happens when two transactions conflict. Contention management has significant influence on the performance of STM implementations, evidenced by the benchmarks of different strategies that have been published. In this work, we implement a selection of these in Clojure's STM to investigate the consequences.

> **Problem statement:** What are the implications of implementing a selection of alternative contention management strategies in Clojure's STM?

We present different strategies found in the STM literature in Section 2.2. Our choice of strategies to implement in Clojure's STM is accounted for in Section 3.4.

[1] A Clojure version of the bank account example is found in Section 3.1.
[2] Other notable examples include Fortress [1], Chapel [11] and X10 [49].

The method for assessing the consequences is based on benchmarking, which is the predominant method used in STM literature on contention management. We have instrumented the code for Clojure's STM implementation to collect data during the execution of the benchmarks. More details are presented in Chapter 4.

1.2 Contributions

The major contribution of this thesis is an analysis of the implications of implementing a selection of contention management strategies in Clojure's STM (see Chapter 5). In our work, we also have the following secondary contributions:

- A classification of Clojure's STM in terms of key design issues found in in the published STM literature. See Section 3.3.
- Discovery of a bug in the widely used DSTM2 [24] that casts some doubt on results obtained in the past few years. See Section 3.4.
- Discussion of the STM literature on benchmarking and data analysis. See Chapter 4 and Chapter 5.

Chapter 2

Software Transactional Memory

The problems and pitfalls of concurrent programming are well-known and a range of tools and solutions have been proposed during the brief history of computer science. Like any other tool, they each have benefits and drawbacks. But, as a purely subjective observation, where some tools and models (e.g., Erlang [4] and its implementation of the actor model [27]) require a radically different mindset as a programmer, transactional memory seems straightforward to use and adapt to. This argument is supported by the fact that programmers have adopted transactions as the means to manage concurrency in database systems.

In this chapter, we will give an overview of the *software transactional memory* landscape. We begin by briefly presenting the history and motivation behind transactional memory and then turn our focus to STM and explain key design issues faced in this research area together with the related terminology. We also include a section on *multi-version concurrency control*, seen in many database systems, because Clojure's STM implements this mechanism.

2.1 Origins

Mutual exclusion through locks is the basic tool for synchronization in multithreaded programming. Unfortunately, locks are known to give rise to several problems: the possibility of deadlocks, livelocks, priority inversion and convoying. Database systems have long offered transactions as an alternative to locks. In such systems, transactions typically give the following correctness guarantees [45]:

- **Atomicity:** *All* operations in a transaction occur or *none* do.
- **Consistency:** Every transaction takes the database from a consistent state to another consistent sate.
- **Isolation:** All tentative changes in a transaction is visible only to that transaction until it commits.

- **Durability:** When a transaction commits, its changes will persist.

Herlihy and Moss [26] first suggested a hardware-based implementation of *transactional memory*, which gave the guarantees of atomicity, consistency and isolation.[1] Their idea was to extend the instruction set of the CPU architecture with the following instructions for accessing memory:

- **Load-transactional**: Read the value of a shared memory location into a private register.
- **Load-transactional-exclusive**: Read the value of a shared memory location into a private register and "hint" that the location is likely to be updated.
- **Store-transactional**: tentatively writes a value from a private register to a shared memory location. The value is not available to other processors before the transaction commits.

The following instructions were proposed to manage transaction state:

- **Commit**: Attempt to make any tentative changes by the above instructions permanent. This instruction only succeeds if there are no conflicts with other processes' transactions, and if no such conflicts are detected, the changes become visible to other processors.
- **Abort**: Discard all tentative changes.
- **Validate**: Returns an indication of the current transaction's state; *true* if the current transaction has not aborted, otherwise *false*.

One problem with hardware transactional memory (HTM) is that it puts extra requirements on CPUs, i.e., to support the instructions mentioned above. Although certain companies[2] have produced systems supporting HTM, it is not widespread. Research in hardware transactional memory is still continuing nonetheless.

The problem mentioned above and the fact that the HTM proposed by Herlihy and Moss is *blocking* inspired Shavit and Touitou's *software transactional memory* (STM) [44]. Their solution only required the instructions conventionally offered by CPU architectures and it was designed to be *lock-free* and thus non-blocking.

Shavit and Touitou's original STM implementation is *static*, which means that the set of memory locations to access transactionally must be specified up-front. Although, as stated in their paper, this allows the implementation of transactional algorithms for most of the known and proposed synchronization operations in the literature, it has disadvantages. Herlihy, et al. [25] developed a *dynamic* STM called DSTM that does not have this restriction, which made it suitable for the implementation of dynamic-sized data structures. DSTM's progress guarantee in terms of *non-blocking synchronization* was weaker then the original STM's, but allowed for simpler implementation. We expand on this in the next section.

[1] Durability is not a concern with regard to main memory, since it is volatile.

[2] Azul Systems and Sun (now Oracle) to mention a few.

Hybrid transactional memory [12] (HyTM) has also been proposed as a means to overcome the lack of hardware support for transactional memory. This approach uses hardware transactional memory *if it is available*; otherwise, software transactional memory is used.

Many STMS implementations has been developed, and research is ongoing. Some believe and argue that transactional memory is still a research toy and that the future for STM looks challenging [8]. Others ask whether skepticism towards STM is analogous to the skepticism that once was towards *garbage collection* [18]. The jury is still out, but judging by the number of STM implementations available at this point and the number of published research articles, STM is far from dead.

2.2 Design Issues

There are many design decisions to make when developing an STM and an ideal design is yet to be discovered. In this section, we present the prominent design points we have found in the literature on software transactional memory.

Static or dynamic: STMs can either be *static* or *dynamic*. The first STM implementation by Shavit and Dice [44] was static, meaning that the set of accessed memory locations had to be specified in advance, i.e., before a transaction started. Dynamic STMs do not impose this requirement, i.e., memory locations are determined during the transaction. Herlihy, et al. [25] were the first to present a dynamic STM.

Granularity of sharing: Some STMs offer transactional access to memory at the word-level (e.g., Shavit and Touitou's original STM [44] and WSTM [22]) and others at the object-level (e.g., Clojure's STM and DSTM [25]). An advantage of word-based STMs is that they apply to a larger range of platforms, including platforms that are not object-based. However, STMs that are word-based will have per-word overhead for transactional access, which can lead to decreased performance. It appears that most STMs are object-based.

Synchronization: Conventional synchronization in multithreaded programming uses locks and mutual exclusion, but locks have, as is widely recognized, several problems and disadvantages. As a response to these problems, researchers have developed a number of alternative synchronization techniques that do not employ locks and mutual exclusion. These techniques can give certain progress guarantees [23]:

- **Wait-freedom**: If a technique ensures that every thread will continue to make progress despite delay or failure of other threads, then it is *wait-free*.
- **Lock-freedom**: If a technique only ensures that *some* thread always makes progress, it is *lock-free*.
- **Obstruction-freedom**: If a technique guarantees that when a thread runs in isolation, with all other threads suspended, it will make

progress, it is *obstruction-free*. In practice, the guarantee applies when a thread runs for a while without encountering any synchronization conflicts.

There is a hierarchy between these guarantees: any technique that is wait-free is also lock-free, and any lock-free technique is also obstruction-free. Wait-freedom is thus the strongest guarantee. These guarantees describe properties of *non-blocking synchronization* and both lock-free and obstruction-free software transactional memory have been implemented.

Shavit and Touitou's original STM [44] was lock-free, but Herlihy, et al. [23] argued that lock-free algorithms are too complex and awkward to implement by using the primitive *compare-and-swap* operations conventionally available. Instead, they suggested to weaken the guarantee by introducing obstruction-freedom and showed examples of algorithms providing this guarantee. In [25], Herlihy, et al. presented *Dynamic STM*, or DSTM—the first obstruction-free STM implementation. In terms of transactional memory, Guearroui, et al. [19] defines obstruction-freedom as a guarantee that a transaction commits if it runs long enough without conflicting with another transaction.

Even though the progress guarantee is weaker than lock-freedom, implementations that are obstruction-free prevent deadlocks and priority inversion. An external mechanism, known as a *contention manager*, is needed to avoid livelock situations.

Ennals [15], however, noted that obstruction-freedom is an inhibiting property in relation to performance. He rebutted the arguments for obstruction-freedom and demonstrated how a faster STM could be implemented, if obstruction-freedom is not a requirement. His findings were backed up by Shavit and Dice [13].

Object metadata: Object-based STMs associate metadata with the objects that are accessed transactionally. The metadata contains information used by the STM, for example timestamps specifying reads and writes and a handle to the transaction that last *acquired* (see below) the object. Such information is necessary to resolve conflicts during transactions.

For some STMs, such as DSTM [25], there are several indirections between the metadata and the actual object, which means that the metadata and the object is unlikely to be stored adjacent to each other in memory. Such indirections were one of the points of criticism that Ennals [15] posed against obstruction-free STMs, since indirections may be the cause of cache-misses that can significantly decrease performance.

Acquisition: Objects to be updated in a transaction must at some point be *acquired*. Acquisition allows for *conflict detection*, i.e., detecting whether two concurrently running transactions overlap in terms of which objects they update. When a transaction T_1 successfully acquires an object O, it makes visible to other transactions, by modifying the metadata of O, that it intends to use (read and/or write) that object. If T_1 detects that O has been acquired by another transaction T_2, it must use a conflict resolution

strategy to determine which transaction should retry and which should continue. Such strategies are part of *contention management*.

Some STMs, such as DSTM [25], WSTM [22] and Clojure's STM, use *eager acquire*, which means acquisition on an object happens as soon as the object is used in the transaction. In other STMs, such as Shavit and Touitou's original [44], acquisition is lazy, which means objects are first acquired at the commit phase of a transaction. Eager acquire allows conflicts to be detected early, and doomed transactions can retry as soon as possible. In STMs with lazy acquire, doomed transactions will always run until the commit phase before they detect whether they should abort, which can be a waste of CPU time. On the other hand, with eager acquire, a transaction may abort another transaction only to be aborted itself at some point.

Reader visibility: Depending on the STM implementation, reading transactions (readers) can be invisible or visible to other transactions. Writing transactions (writers) are always visible to both readers and writers, since they need to make their intention to update an object visible to other transactions.

If readers are visible to writers, it allows writers to abort readers that have previously read an object that the writer updates. In the first DSTM version [25], readers were not visible. Instead, transactions maintained a *read list* containing the values read during the transaction. An object O read for the first time was stored in the read list. Every subsequent read of O would be checked against the read list to ensure that the value was the same as before. If not, the transaction would be aborted. Since readers were invisible it meant that writers could not potentially abort readers, but readers could potentially abort writers.

Contention management: When conflicts occur between transactions, some form of strategy must be used to resolve it. A range of different named strategies have been proposed (examples taken from from [43] and [25]):

- **Passive:** A transaction that discovers a conflict with another aborts itself and retries.
- **Aggressive:** A transaction that discovers a conflict aborts the *other* transaction.
- **Polite:** When a transaction discovers a conflict, it uses bounded exponential back-off. Each time the transaction runs into the same conflict, it doubles the time it waits before trying again until some limit is exceeded. When the limit is reached, the conflict-discovering transaction will abort the competing transaction and continue itself.
- **Karma:** For every object accessed during a transaction, the transaction accumulates *karma*. This is a simplistic estimate of how much work a transaction does—transactions that access many objects will accumulate karma faster than transactions that access less objects. Upon encountering a conflict, the transaction with least karma is forced to abort. A transaction's karma is not reset when it is forced

to abort and retry, meaning that its karma increases for every try. Its karma is reset when it commits.

- **Eruption:** This is a variation of karma. When a transaction detects a conflict, it adds its own karma to its opponents karma thereby (ideally) increasing the speed at which the opponent can complete. If the opponent can complete faster, it is likely that the conflict-detecting transaction can acquire the conflicting object subsequently and then finish faster itself.
- **Priority:** When a transaction starts, the system time is recorded. Older transactions are allowed to run in favor of younger transactions, i.e., transactions started latest are forced to abort.

Benchmarks show that different contention management strategies gives significantly different performance. In some cases, the results published on contention management strategies are contradictory. Polka was proposed by Scherer and Scott [43], because it performed well according to their results. However, results by Guerraoui [19] contradict this claim, saying that none of the benchmarked strategies won in all benchmarks.

We should note here that most contention management strategies, including all of the ones presented, do not provably prevent livelock situations. In practice, though, they do seem to provide progress.

2.3 Multiversion Concurrency Control

Clojure's STM implements a concurrency control mechanism found in several databases[3] called *multiversion concurrency control* or MVCC for short. MVCC ensures that a database transaction never has to wait for access to an object because several versions of that object are maintained. Because of this versioning scheme, transactions appear to be running on a single, isolated snapshot of the database—this is called *snapshot isolation*. MVCC can be a considerable advantage over explicit locking in terms of throughput, especially if there are many transactions that only read data. Transactions that read will not block transactions that write and vice versa.

MVCC is based on *timestamping* to ensure serializability of transactions [45]. Any transaction T_i is assigned a unique timestamp $TS(T_i)$ at its beginning, such that $TS(T_a) < TS(T_b)$ if T_a started earlier than T_b. Each database object P can be associated with different versions $\langle P_1, P_2, \ldots, P_m \rangle$ over time. Besides storing a value, any version P_k also has a read and a write timestamp, denoted Read-$TS(P_k)$ and Write-$TS(P_k)$. The values of the timestamps are as follows [45]:

- Write-$TS(P_k)$ is the timestamp of the transaction that created this version of the object.
- Read-$TS(P_k)$ is the largest timestamp of any transaction that successfully read P_k.

[3] For example, PostgreSQL [42], MySQL [39] and Oracle [40]

If a transaction T_i issues a read or write operation on database object P, then P_k is the latest version of P that satisfies $\mathsf{Write}\text{-}TS(P_k) \leq TS(T_i)$. The following rules apply [45]:

1. If T_i issues a read operation on P, the value T_i will get is the value of P_k.
2. If T_i issues a write operation on P, and if $TS(T_i) < \mathsf{Read}\text{-}TS(P_k)$, T_i is rolled back.
3. If T_i issues a write operation on P and $TS(T_i) = \mathsf{Write}\text{-}TS(P_k)$, then the value of P_k is overwritten; otherwise a new version of P is created.

The first and the third rule are obvious. The second rule ensures that a transaction cannot write a new version of P, if P_k have been read by another more recent transaction.

Snapshot isolation suffers from an anomaly called *write skew*, which we illustrate by example. Say a person has two bank accounts A and B. The bank has the restriction that the sum of the balances must be non-negative, i.e., $\mathsf{Balance}(A) + \mathsf{Balance}(B) \geq 0$. This means that either account can have a negative balance, as long as the other account compensates for the deficit. Let T_1 and T_2 denote two concurrently running transactions. Each transaction sees, for example, that $\mathsf{Balance}(A) = 100$ and $\mathsf{Balance}(B) = 100$. Now T_1 thinks it can withdraw 200 from A and T_2 thinks it can withdraw 200 from B, because each transaction will not see the other transaction's change, which is the property of snapshot isolation. Thus, after each transaction has run, we have that $\mathsf{Balance}(A) = -100$ and $\mathsf{Balance}(b) = -100$, which violates the constraint. In database systems, such as Oracle's, the programmer can specify that an object to be written depends on another object that is read to avoid write skew.

Chapter 3

Clojure

Clojure is a new Lisp. It shares many traits with classic Lisp dialects such as Scheme [48] and Common Lisp [31] but adds unique features as we shall see. In this chapter, we first give an introduction to Clojure the language, its persistent data structures and how to use its software transactional memory implementation. This is followed by a section giving details about the STM implementation. Finally we place Clojure in the landscape of STMs in accordance with Chapter 2.

3.1 Language Introduction

Clojure is a new Lisp dialect with the characteristic syntax based on S-express–ions, i.e., lists enclosed in parentheses. The language is authored by Rich Hickey, who released version 1.0 in 2007 and version 1.1.0 at the end of 2009. The information in this section is based on version 1.1.0, and what we present here is essentially a selection of what can be found on Clojure's official website [28]. The reader should consult this website to learn more about the language and its libraries. Alternatively, a book [21] is available. We only give a high-level overview in this section, but hopefully enough that the reader can grasp the benchmarks listed in Appendix A with relative ease.

Lists form the basic data structure in Clojure that programs are composed of. That is, Clojure is said to be *homoiconic*, because programs are written using the very same data structures that can be used *in* Clojure programs. As stated, lists are enclosed in parentheses and its elements separated by whitespace. In Clojure, commas are considered to be whitespace, which allows elements to be separated by commas if the programmer wants to, although it is not idiomatic Clojure to do so. Here is an example of a list of five elements:

```
(elem1 elem2 elem3 elem4 elem5)
```

The first element of a list is the name of a function or operator (which is essentially the same thing in Clojure). Elements that follow the first are the arguments. For example, the + operator to add numbers is used like so:

```
(+ 1 2 3 4)
```

Lists support nesting. Say we want to sum two lists of numbers and then divide the sums by each other:

```
(/ (+ 5 6 7 8) (+ 1 2 3 4))
```

All language constructs are used by employing lists. For example, an *if* conditional does not have special syntax contrary to many other programming languages. It is used as functions or operators are used (a semicolon denotes a comment):

```
(if (= 2 2)
  ; then clause
  (print "Obviously!")
  ; else clause
  (println "Fatal␣error:␣axiom␣exception."))
```

That is, *if* can be considered a function that takes three arguments: a test, a then clause and an else clause. But *if* is not *really* a function; it is called a *special form*. Special forms are constructs that cannot be written in Clojure itself for various reasons. Besides functions/operators and special forms, Clojure also includes *macros*. They too can be considered a type of function, but a function that produces code. The application of a macro is substituted by the code it produces. Moreover, macros and functions are different because macros can decide if and when its arguments are evaluated. Clojure's core library contains many macros which are commonly used in Clojure programs.

3.1.1 Data Structures

Besides lists, Clojure supports certain primitive data structures including numbers (integers, ratios and floating point), strings, characters and booleans. Furthermore, Clojure has *keywords* and *symbols*. Keywords start with a colon and are just names that evaluate to themselves (just like numbers evaluate to themselves). These are often used as keys in associative data structures such as hash maps. Symbols have the same semantics as they do in other Lisp dialects: they represent the value they are bound to. But symbols can be *quoted* (by placing a apostrophe in front of an identifier), which essentially gives them the same semantics as keywords, i.e., a quoted symbol evaluates to itself.[1]

Contrary to classic Lisp dialects such as Scheme and Common Lisp, Clojure has a set of *persistent data structures*, which comprises lists, vectors, sets and maps. Persistent data structures give certain guarantees [47]. Given an instance a of a persistent data structure, making a modification of a will produce a new instance a' based on a but including the modification. That is, a remains untouched by the modification.[2] Full persistence is supported when both the old and the new version can be accessed and modified subsequently. Partial persistence is supported, when only the new version can be modified but not the old. Clojure supports full persistence, and its implementation is efficient in retaining old versions and producing new ones. Clojure's persistent data

[1] In principle, quoted symbols could be used instead of keywords, but there are minor differences between them. For example, a keyword cannot have metadata, but a quoted symbol can.

[2] One could say it is a misnomer to use the word *modify* here. Persistent data structures are effectively *immutable*.

structures use *structural sharing* meaning that as much data as possible is shared between a and a' to minimize the amount of data that needs to be copied.[3]

We have already seen the syntax for lists, but vectors, sets and maps also have syntax for their literals:

```
; vectors are enclosed in square brackets
[1 2 3 4 5]
; sets
#{1 2 3 4 5}
; hash map using keywords for keys and strings for values
{:key1 "value1" :key2 "value2" :key3 "value3"}
```

Each type of data structure have corresponding functions to create them, i.e., functions named *list*, *vector*, *set*, *hash-map* and *sorted-map*.

Clojure has a plethora of functions to manipulate data structures. It is outside the scope of this thesis to go through all of them. Instead, we refer to Clojure's website [28].

3.1.2 Function Definition

Functions are first-class in Clojure, as is the case in many other functional programming languages. That is, functions can be defined and passed as arguments, returned as results of other functions and be stored in data structures. To define a function, Clojure provides the *fn* special form. A function can be bound to a name with *def*. Here is an example, were we define a function that sums two arguments and bind it to the name *sum*:

```
(def sum (fn [a b]
           (+ a b)))
```

For convenience, Clojure provides the *defn* macro. Using this, we instead get:

```
(defn sum [a b]
  (+ a b))
```

Clojure supports the definition of variadic functions. In the following example, we extend *sum* such that it can take a variable number of arguments and sum them.[4]

```
(defn sum [& args]
  (apply + args))
```

In the above example, *args* is a sequence of all the arguments passed to *sum*. The *apply* function is used here to apply a function to all the arguments in *args* together.

3.1.3 Agents

Clojure has a special type of variable called an *agent*. Agents have some resemblance with actors [27] and processes in Erlang [4], but is not designed to

[3] The official Clojure website gives more information about the performance characteristics of Clojure's persistent data structures: `http://clojure.org/data_structures`. Also, an unofficial overview is available at `http://bit.ly/blhsPN`.

[4] Of course, the + function could be used directly in all of these examples—but bear with us.

support two-way message passing that uses blocking message loops. Agents hold state that can be replaced asynchronously, and the basic operation for doing so is *send*, which sends *actions* to an agent. Actions are functions of one or more arguments that take the agents current state and returns the agents new state.

In the following example, we first bind the name *a* to a new agent, which initially holds an empty vector. Then we define a function that constitutes an action that we will send to *a*. The function takes two parameters; one is the agent's current state, the other is an element to add to the vector that the agent currently holds. The *conj* function is used to add an element to the end of a vector. Finally we use *send* to add a few elements asynchronously.

```
(def a (agent []))

(defn action [old-state element-to-add]
  (conj old-state element-to-add))

; Send the action a few times
(send a action 1)
(send a action 2)
(send a action 3)
```

After running the above program, the state of agent *a* will be a vector of three elements. Using *send* will apply the action function to the agent's state in another thread at some point. The threads used by *send* come from a fixed-size pool with $C + 2$ threads, where C is the number of CPU cores on the system.

Other threads could hypothetically have changed the state of the agent in between the above calls to *send*, in which case the state of the agent is determined by the interleavings of threads and the combination of *send* operations.

3.1.4 Refs and Transactions

A ref is a type of variable that can store an arbitrary value (an integer, a map, a vector etc.), and a ref's value can be retrieved using Clojure's *deref* function. Changing the value of a ref must happen inside a transaction, which is created by using the *dosync* macro. Here we present the canonical example of transferring money from one bank account A to another account B:

```
; Account A initially has a balance of 1000
(def A (ref 1000))
; Account B initially has a balance of 500
(def B (ref 500))

; Function to transfer amount
(defn transfer [from to amount]
  (dosync
    (alter A - amount)
    (alter B + amount)))

; Transfer 250 from A to B
(transfer A B 250)
```

First, the example shows how to bind the names *A* and *B* to a new ref each. Then it shows how to define a function, using the *defn* macro to bind the name *transfer* to a function that takes three arguments: *from*, *to* and *amount*. The function uses *dosync* to make the two operations of transferring money from *A* to *B* atomic. The last line shows how to use the *transfer* function to transfer 250 from *A* to *B*. Each account will end up with a balance of 750.

The function *alter* is used to replace the value of a ref with the result of applying a function (+ and – are functions) to its old value. In this case, we give *alter* three arguments: the ref to alter, the function to apply to it and the amount to withdraw or deposit. The applied function will receive the old value *and* the amount as arguments in said order.

It is also possible to call another function called *ref-set* to directly specify a ref's value. This function takes the ref as the first argument and its new value as the second argument. If we were to use *ref-set* in the above example, *transfer* would instead look similar to this:

```
(def transfer [from to amount]
  (dosync
    (ref-set A (- (deref A) amount))
    (ref-set B (+ (deref B) amount))))
```

It is more idiomatic and convenient to use the *alter* function.

To avoid write skew (see Section 2.3) the *ensure* function is provided, which must be called inside a transaction. It is used to indicate that the value of one or more refs depend on the value read from another. The function returns the value of a specified ref and protects the ref from modification by other transactions until the current transaction commits.

3.2 STM Implementation

Here we will give an overview of the implementation of Clojure's software transactional memory system, such that the reader is equipped to better understand the consequences of the changes we make outlined in Section 3.4. In this section we provide many details; a higher-level classification of the STM is presented in Section 3.3. For even more details about the implementation, the reader should consult [51]. We refer several times to central functions and macros related to Clojure's STM, which are explained in the previous section.

3.2.1 Creating Transactions

Transactions are started with the *dosync* macro. When a transaction is started within a transaction, it joins the outer transaction. This means that any inner transactions will commit when the outer transaction commits.

A transaction can *retry* for various reasons, which we will describe in the subsequent sections. On every try, the transaction is assigned a new *read point*. Read points specify the total order of transaction tries across all transactions started, and is used to implement snapshot isolation. Transactions will retry a maximum of 10,000 times before failing with an exception.

From the perspective of the STM implementation, only operations performed on refs in a transaction's body are important. The refs that have been assigned new values (see Section 3.2.3) will have their new values written when the transaction commits.

3.2.2 Dereferencing a Ref

Much of the implementation of the STM lies within the operations that can be used on refs. In what follows, we will use these operations as the basis for explaining how Clojure implements software transactional memory.

A ref can be *dereferenced*—or read—outside transactions. If several refs are to be read to get a consistent snapshot of their values at a single point in time, a transaction is needed. In any case, reading data from a ref will not block other readers or writers.

Dereferencing is done with the aptly named *deref* operation. If this operation is used outside a transaction, the last commited value at the point of dereferencing is returned. If it is used inside a transaction, the process is more involved. Say we have a ref R and a transaction T. Upon dereferencing R during T, a check is first performed to see if T has set a value for R earlier—this value is called the *in-transaction value* of R and is not visible to other transactions. If R has an in-transaction value in T, that value is returned. If that is not the case, then dereferencing R returns the last *committed* value older than T. When values are committed, they are assigned a commit point. The transaction's read point is compared to commit points to find the newest committed value older than the transaction.

Every ref has a bounded chain of committed values. When dereferencing a ref inside a transaction, it may happen that the chain contains no values committed before the current transaction. This situation can occur when many transactions are running, and the consequence is that the transaction must retry. When this is the case, a *faults* counter is incremented. This will instruct Clojure's STM to increase the size of the chain of committed values to decrease the likeliness of this situation to occur again. The chain cannot grow beyond the ref's specified bound, which has a default value of 10. The bounds can be changed programmatically.

The process of dereferencing a ref inside a transaction is illustrated in Figure 3.1.

3.2.3 Substituting the Value of a Ref

The operations to substitute the value of a ref must all be used inside a transaction. Using them outside a transaction will cause an exception to be raised.

Notice that we deliberately use the word *substitute*, not *change*. The value inside a ref is not changed, because this is the nature of values: they do not change.[5] Instead, the value of a ref is replaced atomically by another value. A consequence of Clojure's data structures being *persistent* (see section Section 3.1.1) is that they have value semantics, and Clojure's STM is only helpful if

[5]Rich Hickey has written an essay about the importance of this in Clojure, which is available at `http://clojure.org/state`.

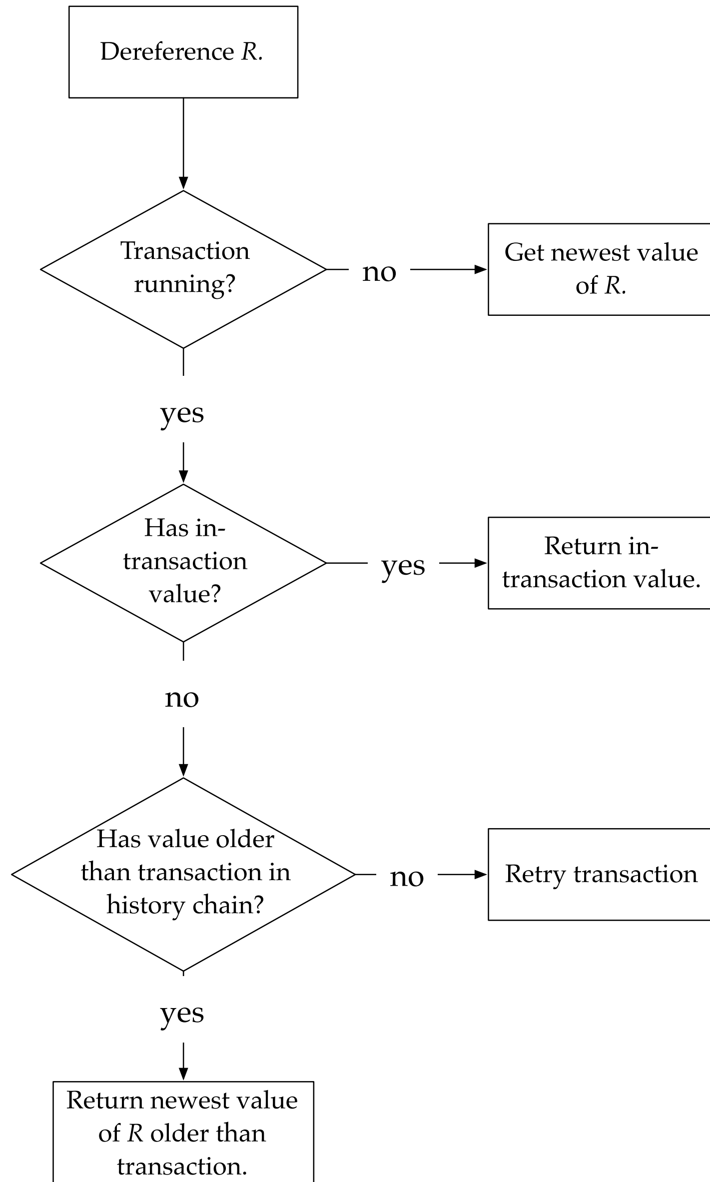

Figure 3.1: A flow chart illustrating the process of dereferencing a ref.

you use Clojure's data structures. If you were to store a reference to a regular Java object in a ref, for example, you are suddenly able to change what the ref stores without using a transaction, since Java objects are not persistent (and hence immutable).

The following operations can be used to replace the value of a ref with another:

- **ref-set**: To directly specify a ref's value.
- **alter**: To specify a ref's next value by applying a function to the ref's current value.
- **commute**: To replace the value of a ref with the result of applying a commutative function to its current value. This allows for more concurrency as we will explain in Section 3.2.4.

We will explain the two operations *ref-set* and *alter* together, since *alter* can be considered a convenience operation implemented in terms of *ref-set*. In case *alter* is used, the new value of the ref is calculated by first applying a function to the ref's old value and using the result as the ref's new value. In both cases, the ref's value is set using the same call to the underlying STM implementation, and it is this call we are interested in explaining. We have illustrated the process of using *ref-set* in Figure 3.2. In the following, we refer to using either *ref-set* or *alter* as *setting* the value of a ref.

When, during a transaction, a new value for ref R is attempted set, the STM first checks whether there is a write conflict. A write conflict occurs when a transaction T_1 detects that another running transaction T_2 has already acquired R (see *Acquisition* in Section 2.2). First, T_1 will attempt to *barge* the other transaction, which means T_1 will try to make T_2 retry, such that T_1 itself can continue and acquire R. If T_1 does not succeed in barging, it will retry itself.

There are three conditions that must be met before T_1 can barge T_2. First, T_1 must have run for at least 10 milliseconds. Second, T_1 must be older than T_2. The third condition requires that we know the different statuses a transaction can have:

- **Running**: The transaction is running.
- **Committing**: The transaction is in the process of committing.
- **Retry**: The transaction will retry, but has not begun yet.
- **Killed**: The transaction has been barged.
- **Committed**: The transaction has finished by successfully committing.

Transaction T_1 can only successfully barge T_2 when T_2's status is *running*. If T_2 is committing, has committed, has been killed or is about to retry, then T_1 will retry.

If T_1 cannot successfully barge T_2 upon detecting a write conflict, then T_1 will set its status to *retry*. Before actually retrying, T_1 will block and wait for a maximum of 100 milliseconds or until T_2 stops by committing or retrying.

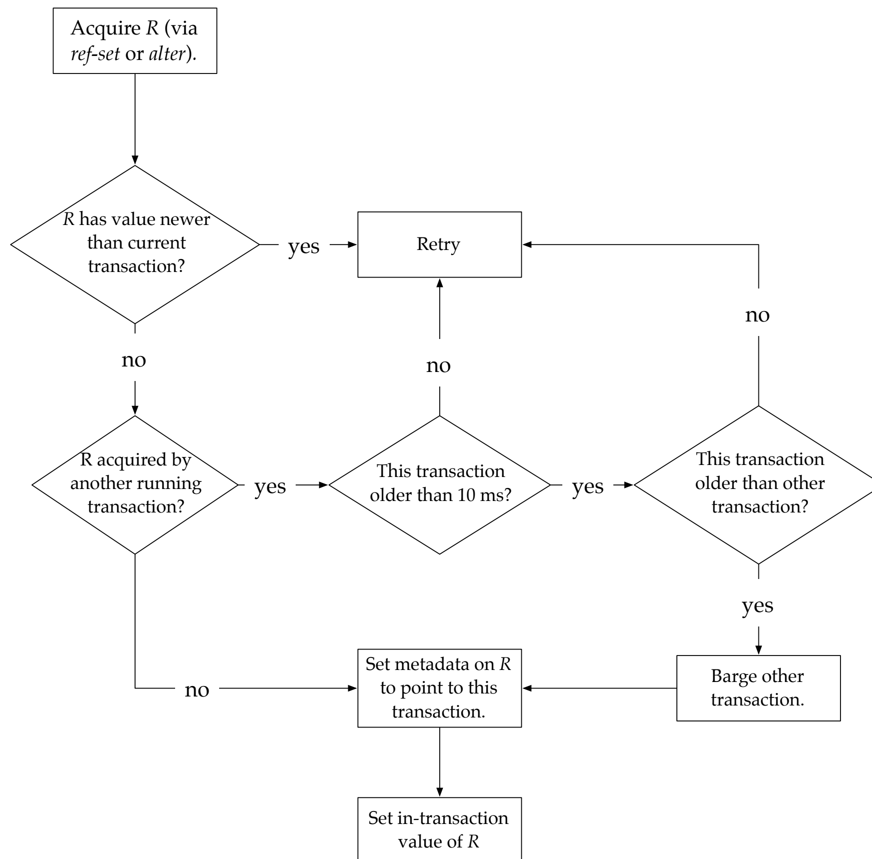

Figure 3.2: A flow chart illustrating the process of acquiring a ref via *ref-set* or *alter*.

This is to reduce the overall likeliness that the two transactions will experience a write conflict again.

At commit time, the STM will check that all refs attempted written by the transaction have not been changed by others and that no other transactions are causing a write conflict. If a write conflict is detected, the conflict resolution process is the same as explained earlier.

The *commute* operation replaces the value of a ref with the result of applying a commutative function[6] to the ref's current value. But whereas *alter* is used when you rely on the resulting value during your transaction, *commute* is for situations where you do not. Upon invoking *commute*, Clojure's STM does not try to detect any write conflicts—the reverse is true for *alter* (and *ref-set*). Write conflicts for commutes will not be detected until commit time.

Refs can have *validators* attached, which are simply functions that take a single argument. When a transaction reaches the commit phase, all the attached validators are applied to the ref's potentially new value. If the validator returns false or raises an exception, the transaction aborts (without retrying) by raising an exception and the ref remains unchanged.

3.2.4 Locks

Locks are used in several places in Clojure's STM implementation, but care has been taken to ensure that they are held for the shortest amount of time possible. Clojure makes extensive use of a *reentrant read/write lock*, which allows multiple readers or a single writer at any given time.[7]

To dereference a ref inside a transaction, a read lock is taken on the ref, which is held while running through the chain of previously committed values until a value older than the transaction is found. If dereferencing happens outside a transaction, a read lock is taken to get the most recent value, which does not involve running through the aforementioned chain.

When *ref-set* or *alter* is successfully used on a ref, i.e., without causing a retry, a write lock is taken to associate data about the transaction attempting to write to the ref. This data is used by other transactions during write conflict resolution to determine what to do (retry or try to barge the other transaction). The write lock is released as soon as this data has been set.

Clojure provides the *ensure* operation to avoid write skew in situations where a transaction may read and depend on the value of a ref, but does not change the ref. When *ensure* is successfully used on a ref, a read lock is taken on that ref and held for the rest of the transaction. When the transaction commits or retries, the read lock is released. Setting the value of a ref (via *ref-set* or *alter*) that has already been ensured will release the read lock previously acquired.

Using *commute* will only take a short-lived read lock during the transaction, which is the reason it allows for more concurrency than *ref-set* or *alter*, which both take a write lock that excludes all other readers and writers.

At the commit phase of a transaction, all refs to be assigned a new value are locked with a write lock. This also applies to refs that have been commuted.

[6] It is up to the programmer to make sure that the function *is* commutative.

[7] The corresponding class is called *ReentrantReadWriteLock* from Java's concurrency package.

As soon as the new values have been validated and written, the write locks are released in the order they were taken.

3.3 Clojure in the STM Landscape

In this section, we place Clojure's STM in the landscape of software transactional memory according to the design decisions outlined in Section 2.2.

Static or dynamic: Clojure's STM is dynamic. The range of refs accessed during a transaction need not be specified in advance. Every ref that needs to be given a new value in the commit phase of a transaction is added to a list during the transaction. Commutes are added to a special list.

Granularity of sharing: The only objects that are considered by Clojure's STM are refs. Data in refs are not of any fixed size, i.e., there is no restriction that the size of data in refs must align on machine word boundaries. The granularity of sharing can thus be said to be at the ref-level.

Synchronization: Clojure uses blocking synchronization in its STM (see Section 3.2.4). Rich Hickey, the author of Clojure, has confirmed in personal correspondence with us that this decision was inspired by the paper by Ennals [15].

During a transaction, when a transaction tries to replace the value of a ref (via *ref-set* or *alter*) it will attempt to take a write lock (to *acquire* the ref) as explained in Section 3.2.4. This attempt has a time-out (100 milliseconds) that ensures a transaction will not wait indefinitely for a lock, which could lead directly to deadlocks.

When a transaction enters the commit phase, other transactions can no longer abort it and acquire refs that it is about to update. The other transactions will abort and retry themselves. Thus, when in the commit phase, the transaction can take write locks on the refs in any order.

Using *commute* will not acquire a ref during the transaction—the acquisition does not happen until the commit phase. Here, the order of the locks taken matters. Say two transactions T_1 and T_2 each use *commute* on refs R_1 and R_2 but in different orders. If the locks were taken in the order of the *commute* operations, a dead lock could occur in the commit phases of the two concurrently committing transactions. For example, T_1 would first lock R_1 then R_2; T_2 first R_2 then R_1:

T_1	T_2
lock R_1	lock R_2
lock R_2	lock R_1

This gives a deadlock, since T_1 is never allowed to get the lock on R_2 and T_2 never gets the lock on R_1.

Object metadata: The metadata used by Clojure's STM is stored together with the refs. Refs have a reference to an object that encapsulates data about

the last transaction that successfully acquired the ref, including the transaction's timestamp and status. This is used by transactions to detect which transaction last acquired the ref and to determine if a transaction can be barged.

Acquisition: Clojure implements *eager acquire* in its STM. Acquisition means setting the *tinfo* object mentioned under *object metadata* above. Since acquisition is eager, conflict detection occurs on every attempt to replace the value of a ref. Commutes do not acquire the ref during a transaction but first in the commit phase.

Reader visibility: Read-only transactions (readers) are not visible to other readers or writers, and thus writers (or other readers) will not be able to barge readers. In terms of the implementation, only writing transactions set metadata on the refs (which makes the transaction visible to others).

Contention management: When conflicts occur in Clojure, the STM uses a strategy similar but not identical to *Priority* (see Section 2.2). As with Priority, a transaction T_1 is allowed to abort another transaction T_2 if T_1 is oldest. But Clojure further imposes the requirement that T_1 has run for at least 10 milliseconds. The time is measured since the transaction was *created*, not since the beginning of the last retry. As part of our results, we show the difference between Priority and Clojure's original implementation in Chapter 5.

As mentioned in Section 2.3, Clojure employs multiversion concurrency control in its STM implementation. Clojure's persistent data structures fits particularly well with this concurrency control mechanism:

- Creating a new version of an instance of a persistent data structure is an efficient operation.
- Creating the new version will not in any way alter the old version.
- Since Clojure's persistent data structures have structural sharing, the memory needed to store several versions of a particular data structure instance is low.

3.4 Selected Contention Management Strategies

We have selected the contention management strategies *Aggressive*, *Priority* and *Karma* (see Section 2.2). The Aggressive strategy is in a sense simple and naïve since any detected conflict just forces an opposing transaction to retry right away. Intuitively, this should make the strategy a cause of frequent livelocks, and we are interested in observing whether this holds true in practice.

Priority is not very different from the contention management strategy already found in Clojure. In Clojure's original implementation, a transaction T_1 can only *barge* (see Section 3.2) another transaction T_2 if T_1 has run for at least 10 milliseconds, known as the *barge time*. This check is not performed in the

```java
public void resolveConflict(Transaction me, Transaction other)
{
  ContentionManager otherManager = other.getContentionManager();
  for (int attempts = 0; ; attempts++) {
    if(other.isActive()) {
      long delta = otherManager.getPriority() - priority;
      if (attempts > delta) {
          other.abort();
      }
    } else break;
  }
}
```

Listing 3.1: The implementation of contention management with Karma in DSTM2 [24].

Priority strategy, which handles conflicts by letting only the oldest transaction continue. By removing the check whether the barge time has elapsed, we effectively have an implementation of Priority in Clojure's STM. This lets us gauge the effect of the barge time to see if it has any real significance in our benchmarks. It is worth noting here that our implementation of Priority was inspired by the implementation found in DSTM2 [24].

Karma was by far the most difficult strategy to implement in Clojure's STM. To illustrate, we list the code to resolve conflicts with Karma from DSTM2 [24] in Listing 3.1 and the implementation we have written for Clojure's STM in Listing 3.2. The code we list from DSTM2 is from the obstruction-free version, i.e., it does not use locks. Clojure's STM, on the other hand, uses locks, which greatly complicates the implementation. We can safely say that our experience with implementing a correct version of Karma in Clojure have proven to us that programming with locks is a pain![8]

The implementation of Karma from DSTM2 contains a bug, which is visible in Listing 3.1: there is no sleep time between each check whether the opposing transaction has more Karma. This means that a transaction that discovers a conflict will gain Karma very quickly and will be able to abort the opposing transaction accordingly. With this bug, Karma will resemble Aggressive to a certain degree. It is worth mentioning that any published research based directly on the available DSTM2 implementation may have been affected by this bug.

Most published benchmarks using these strategies have used obstruction-free (see Section 2.2) DSTM [25] or DSTM2 [24]. In general, it is interesting to see the result of applying the strategies on an STM such as Clojure's with its rather unique combination of features.

[8]Not to say that our implementation is perfect. More experienced programmers, such as Bryan Cantrill [7], would likely have been able to simplify and improve the implementation.

```java
private void karmaResolve(Ref ref) {
  int attempts = 0;
  boolean barged;
  boolean locked;

  while(true) {
    if(!info.running())
      throw retryex;

    locked = false;
    attempts++;
    try {
      if(ref.lock.writeLock().tryLock()) {
        locked = true;

        if(ref.tvals != null && ref.tvals.point > readPoint) {
          throw retryex;
        }

        if(ref.tinfo == null || ref.tinfo == info) {
          ref.tinfo = info;
          return;
        }

        if(ref.tinfo.running()) {
          LockingTransaction otherTx = ref.tinfo.transaction;
          int otherPriority = otherTx.karmaPriority.get();
          int thisPriority = karmaPriority.get() + attempts;

          if(otherPriority < thisPriority) {
            barged = ref.tinfo.status.compareAndSet(RUNNING, KILLED);
            if (barged) {
              ref.tinfo.latch.countDown();
              ref.tinfo = this.info;
              return;
            }
          }

          ref.lock.writeLock().unlock();
          locked = false;
          spinSleep(1000);
        }
        else {
          ref.tinfo = info;
          return;
        }
      }
      else {
        spinSleep(1000);
      }
    }
    finally {
      if(locked)
          ref.lock.writeLock().unlock();
    }
  }
}
```

Listing 3.2: Our implementation of Karma. Adapted for inclusion.

Chapter 4

Benchmarking

The many published articles in the area of software transactional memory research reveals benchmarking as the preferred method of gauging characteristics of implementations and techniques. We borrow a definition from [46], which defines a benchmark *"as a test or set of tests used to compare the performance of alternative tools or techniques."* A keyword in that definition is performance, which is a source of much trouble because it is hard to define with respect transactional memory.

Despite the prevalence of the method, there is no explicit consensus on how to benchmark transactional memory. Several benchmark programs and suites have been proposed but none have gained widespread adoption and are ususally only available in just one or two languages, e.g.: STMBench7 [20] is available in Java and C++, STAMP [37] in C, and Wormbench [55] in C#. There is neither any consensus on the precepts for performing a benchmark as evidenced by the diverse approaches to benchmarking found throughout the literature. This situation is to be expected:

> "During the early days, when a research area is becoming established, it is necessary and appropriate to go through a stage where diverse approaches and solutions proliferate. At this time, the bounds of the area are being established and different methods are being applied. This proliferation is desirable, so there will be a variety of tools and techniques to be compared by the benchmark." [46]

The literature attests to the proliferation of tools and techniques with different approaches to transactional memory, such as hardware, software and hybrid transactional memory, the introduction of adaptive mechanisms, eager and lazy strategies, and different levels of guarantee, e.g. wait- and obstruction-freedom, to name a few. How far the area is in establishing it's bounds is unclear, but there are signs of it beginning to take place:

> "Evidence that a community has reached the required level of maturity and is ready to move to a more rigorous scientific basis comes in many forms. Typical symptoms include an increasing concern

> with validation of research results and with comparison between solutions developed at different laboratories; attempted replication of results; use of proto-benchmarks (or at least attempts to apply solutions to a common set of sample problems), [...]" [46]

There is a growing concern with results gathered from current benchmarking efforts. Some researchers have proposed frameworks in which different STM algorithms can be implemented and benchmarked against a common workload [24]. From using mostly simple benchmarks, such as red-black trees and lists, generally called micro-benchmarks, researchers have built, and are continuing to build, ever more complex benchmarks because of a growing concern for the validity of results [37, 20, 52, 55]:

> "One common limitation for all the TM evaluations referred to so far is that they use micro-benchmarks or benchmarks where the parallel structuring has already been done, and therefore do not properly examine the practicality of TM programming." [52]

Some researchers have turned their attention to investigating the benchmarks in use to assess whether or not they are representative of real workloads. One effort looked at the characteristics the benchmark suites SPLASH-2 [53], PARSEC [5] and STAMP [37], and found a high degree of clustering with respect to several metrics, meaning a lack of diversity in the ways in which they stress a TM system, and concluded that "more emphasis should be placed on the design and implementation of transactional memory programs if this field is going to continue to grow" [29]. That opinion is also found in [24] which says: "the community needs more experience with applications written in a transactional style before we can reach consensus on the best ways to support this API."

In other cases, researchers have sought to provide a better understanding of programs that exhibit transactional behavior by analyzing 35 lock-based programs [9]. Three programs from the STAMP [37] suite, vacation, genome and k-means along with Lee-TM [3] were investigated in [2]. Some efforts with the explicit goal of providing realistic benchmarks include STMBench7 [20] and Lee-TM [3].

The benchmarks mentioned so far, and in fact nearly all benchmark programs and suites available, are implemented in imperative languages. The authors of [41] provide the first and only benchmark for the well-known functional programming language, Haskell. The dominance of languages, such as C, C++ and Java, with respect to benchmarking of STM systems is noted in [10] where the authors translated the Delaunay Mesh Refinement benchmark from the Lonestar [33] benchmark suite, from C++ to Common Lisp. They intend to translate the STAMP suite to Common Lisp.

The dearth of STM benchmarks for functional programming languages in general, and the complete lack of ones written in Clojure or Common Lisp, requires that we develop some ourselves. This brings up several questions, but one in particular, which is the scale, or size, of the benchmark or benchmarks. This we deal with in the next section.

4.1 Micro or Macro

Generally, benchmarks are distinguished by their scale and labelled as either micro- or macro-benchmarks. To our knowledge these categories have not been clearly defined, but most articles apply the term micro-benchmark to workloads that consists of operations performed on a single data structure, e.g., red-black tree, hash-map or linked-list, and macro-benchmark to anything more complex than that. Examples of macro-benchmarks include: Atomic Quake [56], a transactified version of the server application for the once popular multiplayer game; STMBench7, which consists of *"a set of graphs and indexes intended to be suggestive of many complex applications, e.g., CAD/CAM"* [20] and Stanford Transactional Applications for Multi-Processing [37], or STAMP, which as of version 0.9.10 includes eight applications that do various tasks, such as gene sequencing, maze routing and simulation of a travel reservation system.

A single micro-benchmark by itself, provides only a narrow view of the performance of a TM design and can produce misleading results, for the reason that it is unlikely to be representative of the tasks that the STM eventually will be subjected to. Using several different micro-benchmarks can provide more insight, but [29] notes that:

> "If too few benchmarks are chosen, the applications may not provide the stressors needed to evaluate a design. If too many benchmarks are chosen, their behavior may overlap and increase design time without providing additional useful information."

The microbenchmarks are losing in favor of larger and more complex benchmarks, such as STMBench7, based on the rationale that:

> "Measuring performance of an STM in an overly simplified setting can be at best uninformative and at worst misleading as it may steer researchers to try to optimize irrelevant aspects of their implementations." [20]

The authors of [20] found that of the four STMs they benchmarked with STMBench7, all *"crashed at some point or another"* [14]. They explain in detail that the problems were mostly related to memory management. An important finding of [14] is that the use of micro-benchmarks had not revealed the fact that none of the four STMs were unbounded and dynamic—a point which hardware transactional memory is often critized for. Additionally, the authors found that:

> "Performance results we gathered also differ from previously published results. We found, for instance, that conflict detection and contention management have the biggest performance impact, way more than other aspects, like the choice of lock-based or obstruction-free implementation, as typically highlighted." [14]

In their conclusion they note: *"[. . .] observations made on small-scale tests do not translate well to large-scale, thus testing of larger-scale systems is necessary."* [14]

In light of the critique of microbenchmarks, or small-scale tests in general, it seems large-scale benchmarks are the way to go. However, constructing a benchmark of considerable scale, such as STMBench7, is a major undertaking in itself that requires much effort—far more than the scope of this project allows. The difficulty of developing good, representative, large-scale benchmarks is discussed in [46, 50, 30].

A great deal of research has been conducted with the use of micro-bench–marks and produced valuable results. Precisely because they are small, they can be tailored to stress a particular aspect of an STM, which is valuable to the researcher who needs data on how a particular part of an STM behaves when stressed excessively. The major pitfall is being unaware of the limitations of small-scale, or micro-benchmarks, and stretching ones conclusions too far.

Rather than settling on a single, large benchmark, we opt for a collection of smaller programs of varying size. Instead of writing up all the programs from scratch, we have found inspiration in various existing benchmarks. Our benchmark programs are presented in Section 4.4. We believe this to be a reasonable collection as our purpose is not to compare Clojure's STM to other implementations, but to gauge the impact of the changes we make.

4.2 Metrics

We have found no set of generally accepted metrics for benchmarking STM algorithms in the literature. On the contrary, there is a variety of metrics used, some more esoteric than others. For example: [29] counts instructions and memory addresses, uses nine metrics, six of which are ratios and an additional five metrics, four of which are counted in processor cycles; [34] measures only speedup ratio, execution time and transaction abort ratio; [36] uses transactions per second exclusively; [41] uses 15 metrics, counts transactions and size of read- and write-sets. Agreement upon the usage of metrics or at least a form of guideline in this respect would benefit future research by making results more amenable to comparison.

We use a selection of metrics from [2]. They are well-defined, provide a good deal of information and in some cases complement each other. Descriptions are adapted from the same source.

- **Speedup**: A measure of how performance scales as the number of threads increase. The closer the speedup is to being linear, the better.
- **Wasted work**: The percetage of time spent executing transactions that was wasted due to of retries/rollback. High values can indicate poor contention management
- **Aborts per commit (retries)**: The average number of aborts per commit. In conjunction with Wasted work, it gives an indication of whether the wasted work was caused by many or only by a few, but long transactions.
- **Contention management time**: The percentage of the total execution time spent managing contention.

4.3 Methodology

As noted in the beginning of this chapter, there is no discernable agreement about how to approach benchmarking in the field of transactional memory research. In particular methodology and data analysis is approached in many different ways and with varying levels of rigor [16]. This section gives a detailed description of the central elements of our method.

Blackburn et al. [6] states that there are *at least* five *degrees of freedom* in experimental evaluation that can introduce bias into the obtained results. Two of them, the host platform (hardware and operating system) and the language runtime, are common to managed as well as unmanaged languages. The remaining three, heap size, nondeterminism and warm-up, are only relevant to managed languages.

Researchers are exhorted to experiment with different hardware and operating systems to avoid results that are biased towards a particular configuration [6]. We have only access to a single machine with eight cores and because it is used by other researchers, we have to settle for the operating system that is already installed.

The size of the heap greatly impacts how active the garbage collector is as well as overall performance [54]. Generally, a smaller heap means more frequent collections which is desirable in some circumstance, but not in this one. The transactions in our benchmarks are in many cases so short that even a fast collect can extend the measured duration significantly. With faster but more frequent collections, a greater number of transactions are affected which exacerbates the distortion because the duration of a fast collect, in some cases, is greater than the durations of the transaction it affects. Furthermore, very frequent activity from the garbage collector can slow down the program [54] In contrast, by configuring the JVM to use most of the available memory, many of the benchmarks can be executed without invoking the garbage collector at all during measurements. In the cases where the garbage collector is invoked, only a few transactions are affected, albeit to a larger extent.

The JIT-compiler in Sun's HotSpot JVM nondeterministically optimizes parts of a program as it runs, based on information collected at runtime. Intuitively, compilation activity is greatest during the first iteration after which it tapers off. This is the issue of warm-up. Furthermore, the compiler may optimize the same program more aggressively in some cases and less so in others. It is possible to start the JVM in interpreter-mode, but performance is greatly reduced.

For the obtained results to be credible, it is necessary to control nondeterminism sufficiently. This is by some considered the most challenging aspect of experimental evaluation, from a methodological standpoint [6]. This is understandable, considering that seemingly insignificant changes can have a significant impact, such as the alignment of the C stack in memory [38]. Two strategies have been proposed but neither have seen any significant adoption within the area of software transactional memory research. One is replay compilation [17] in which a compilation plan is created from a series of training runs and subsequently used to deterministically apply optimizations. This however requires support from the virtual machine.

An alternative which does not require any support is *multi-iteration determinism* [6]. This strategy counters nondeterminism by means of repetition. Specifically, the run-time is started multiple times and in each instance, the benchmark is executed multiple times. This is the basic idea in our slightly modified version of the *"practical statistically rigorous methodology"* [16].

Invoking the run-time several times makes it possible to obtain statistically independent results [16]. Multiple iterations of the benchmark allows for the JIT-compiler to warm up the code. As compiler activity decreases, we expect most programs to reach *steady-state*. This general notion is made quantifiable by the methodology proposed in [16].

Specifically, steady-state is determined by what is called the *coefficient of variation* or CoV, which is defined as the standard deviation divided by the mean. For a friendly introduction to these concepts we refer to [35]. By measuring the time it takes for each iteration of a benchmark, an average duration can be calculated for series of iterations and the coefficient of variation can be determined. If this value falls below a preset threshold, for example 0.01 or 0.02 as suggested by [16], the iterations are considered to be steady-state. For data analysis, an estimate of the average duration and it's associated confidence intervals are calculated based on the mean of the steady-state iterations in each instance. The reason is that the durations measured within the same instance are not statistically independent, but the means from different instances are [16].

As described in [16] a fixed number of iterations is performed for each instance of the run-time. At a later stage, steady-state is determined for each instance by considering a window of iterations, starting from the last and extending backwards to include prior iterations, as long as the coefficient of variation does not exceed a desired threshold. The approach we take differs in that we use a sliding window of fixed size. As a benchmark is executed repeatedly, the new duration is included, the oldest excluded and the coefficient is determined anew. If it is below the specified threshold, the instance is shut down and the next is instance started. If the benchmark is executed a specificed maximum number of times, the instance is shut down.

The sliding window serves two purposes, one of which is to save time. Because we have several benchmarks, contention managers and CPU core configurations, doing a full run can easily take several days if configured too optimistically. This is especially true if some benchmarks are unstable, as determined by the CoV-threshold, and thus exhaust the maximum number of iterations allowed in each instance.

For anyone doing experiements, especially with managed languages, [6, 16, 38, 54] are recommended reading.

4.3.1 Benchmark Rig

To conduct the experiements in a methodical manner, we constructed a rig which entirely automates the process of executing the experiments and collecting data. The rig is comprised of the following components: instrumentation, scaffolding and a shell script.

There are two parts to the instrumentation: the code injected at certain

places in Clojure's *LockingTransaction* class to collect information about each transaction and a number of arrays constituting a global transaction log. In addition there are utility functions for initializing and resetting the log, writing the log to a file, collecting metadata about the JVM instance and calculating the coefficient of variation.

The scaffolding is written in Clojure and uses the utility functions exposed by the instrumentation to perform repeated runs of a benchmark until steady state is reached or a maximum number of runs has been performed. Before each run the garbage collector is invoked to do a full collect, the log is reset and the benchmark is initialized. After each run the log is written to a file. The source code for the scaffolding is included in Appendix B. Note in particular the functions *dobenchmark*, *run-until-steady* and *timed-run* in the scaffolding API.

The shell script is responsible for continually invoking the JVM. It consists primarily of four nested for-loops. The three outermost loops produces the Cartesian product of contention managers, benchmarks and number of cores to use. The innermost loop repeats each combination a given number of times, in our case ten. The shell script is included in Appendix B.

We designed the rig to have the smallest possible overhead. The shell script has effectively no impact and the scaffolding code is only executed inbetween the repeated runs of a benchmark when no measurements are being made. The instrumentation is essentially the sole contributor of overhead because it imposes extra work on each transaction. However, the added work is neglible in comparison with the overhead of the transactions themselves.

As data is collected during a transaction, it is stored temporarily in instance variables in the `LockingTransaction` object. When the transaction has finished committing and effectively ended, the data is written to the shared log. By postponing the writes until the transaction has ended, only a minimum number of accesses are made to the shared log. Any delays at that point affect the measured duration of the benchmark iteration, which, by virtue of being several orders of magnitude longer, is less sensitive. Finally, the `LockingTransaction` object and the instance variables that hold the data are likely in the cache because each thread reuses the object for subsequent transactions. While there is no guarantee that a certain thread will not be executed on a different core and thus with an outdated cache, the potential overhead is insignificant.

4.4 Benchmark Suite

We have developed four benchmarks to test Clojure's STM with the different contention managers we have implemented. We describe them below, and the source code for them is included in Appendix A. Note that the benchmarks have not been written for best performance in their respective scenarios. Instead, they have been adapted to stress Clojure's STM in different ways.

The number of threads started by each benchmark is either 1, 2, 4 or 8, depending on which configuration is currently running.

Shared Integer: This is the simplest of the benchmarks. Multiple threads are started and each of them has the simple task of incrementing a shared integer. Each thread starts a transaction and increments the integer by

one 100,000 times. The benchmark will resultingly run a large number of short transactions. We found the inspiration for this benchmark in [41].

Vacation: A simulation of a system for vacation planning. The system holds four Clojure refs as shared memory, each representing a table containing cars, flights, rooms and customers. Each car, flight and room has an associated ID, total count and available count. Each customer has a list of reservations associated, which specifies what has been reserved, the ID of the reserved item and the price.

Multiple threads will run an equal fraction of 25,000 rounds, and in each round they will perform four actions. These four actions are selected at random in each round:

- make a random reservation of a car, room or flight,
- delete a random customer and that customer's reservations, or
- update the total count of an item by adding or subtracting a random number while ensuring that no customer will loose reservations, and that there will be no state of inconsistency.

Each action involves reading and writing multiple tables. The transactions are longer than the transactions in the *Shared Integer* benchmark, and their duration will vary slightly. The duration of a transaction depends on which of the above actions are seleced for that transaction.

The benchmark was inspired by the vacation benchmark in the STAMP suite [37].

Monte Carlo Pi: Monte Carlo methods refer to algorithms that use repeated random sampling to compute results. In our case, we use the Monte Carlo method to calculate an approximation of Pi.

Calculating π is done by first randomly scattering a number of points into a coordinate system represented by a square grid. We now imagine that we inscribe a circle into the grid. Let c denote the number of points inside the circle and t the total number of points. The approximation P of π is then defined as $P = 4\frac{c}{t}$.

In our implementation, each cell in the grid is represented by a ref. Points are placed into these refs through transactions. Occasionally, two or more threads will conflict, which ensures contention. The transactions in this benchmark are relatively long-lived compared to Shared Integer and Vacation.

K-means: This benchmark is a parallelized version of the K-means algorithm. The algorithm's goal is to find k clusters of data by using continual refinement. Data is represented as feature vectors and as such can be understood as points in some n-dimensional space.

The actual clustering happens when each data point (feature vector) is assigned to the cluster that has the closest centroid. A centroid is a single n-dimensional vector that represents the mean of all data points in a cluster. When data points have been added to clusters, the centroid of

each cluster is recalculated and potential reassignment of data points is done. The algorithm stops when centroids do no longer change or when a specified maximum number of iterations has been reached.

Parallelization of this algorithm can be done by dividing the complete set of data points into c partitions, where c is the number of CPU cores available, and then have c threads take care of the actual cluster assignment of each partition. The clusters are represented as Clojure refs.

We were inspired to implement k-means by both the STAMP [37] suite and a Clojure parallelization case study [32]. In the latter case, the case study was done to perform comparisons with other implementations of parallel k-means. This is not our aim, which is the reason we have made no effort to optimize our implementation to make it particularly fast.

Compared to the other benchmarks, the K-means benchmark uses transactions that are longer-lived, depending on the size of the data set.

Chapter 5

Results

We here give an overview of the results obtained through our benchmarks and highlight the most interesting findings. The confidence intervals shown in the following diagrams are at the 95 % level unless otherwise noted. For all diagrams showing a confidence interval, three numbers are displayed per data point. The number at the top is the upper bound of the confidence interval, the middle number is the average, and the bottom number is the lower bound of the confidence interval. We show diagrams according to the metrics presented in Section 4.2. When we refer to Clojure in the following, we mean the contention management strategy found in the Clojure's STM.

Figure 5.1 shows the speedup ratio of contention managers across benchmarks and CPU cores. Clojure, Karma and Priority has a speedup in K-means as the number of cores increases. Aggressive, on the other hand, slows down dramatically when going from two to four cores. In the three other benchmarks, this pattern does not occur. In Monte Carlo Pi (abbreviated *mcp* in the diagrams), Clojure and Priority speeds up for two and four cores, but slows down for eight cores. All contention managers experience a slowdown in the Shared Integer and Vacation benchmarks. Karma and Aggressive exhibits a similar pattern in both Monte Carlo Pi, Shared Integer and Vacation, i.e., they slow down as the number of cores increases.

Figure 5.2 shows benchmark durations for the different contention managers as the number of CPU cores increases. It is clear from the large confidence intervals that Aggressive is very unstable in the K-means and Monte Carlo Pi benchmarks, especially for four and eight cores. Clojure and Priority are practically identical in K-means independent of the number of cores, but in Shared Integer, Vacation and Monte Carlo Pi they differ significantly with non-overlapping confidence intervals. Karma performs comparably to Clojure and Priority in K-means and Vacation. For Shared Integer, Karma is better than Priority but worse than Clojure. Finally, in Monte Carlo Pi, Karma is dramatically worse than Clojure and Priority for eight cores. The pattern is the same for less cores, albeit less pronounced.

Figure 5.3 shows the average percentage of wasted work per transaction. Aggressive's confidence intervals are large for K-means in general and for Monte Carlo Pi with two cores, indicating a significant amount of fluctuation

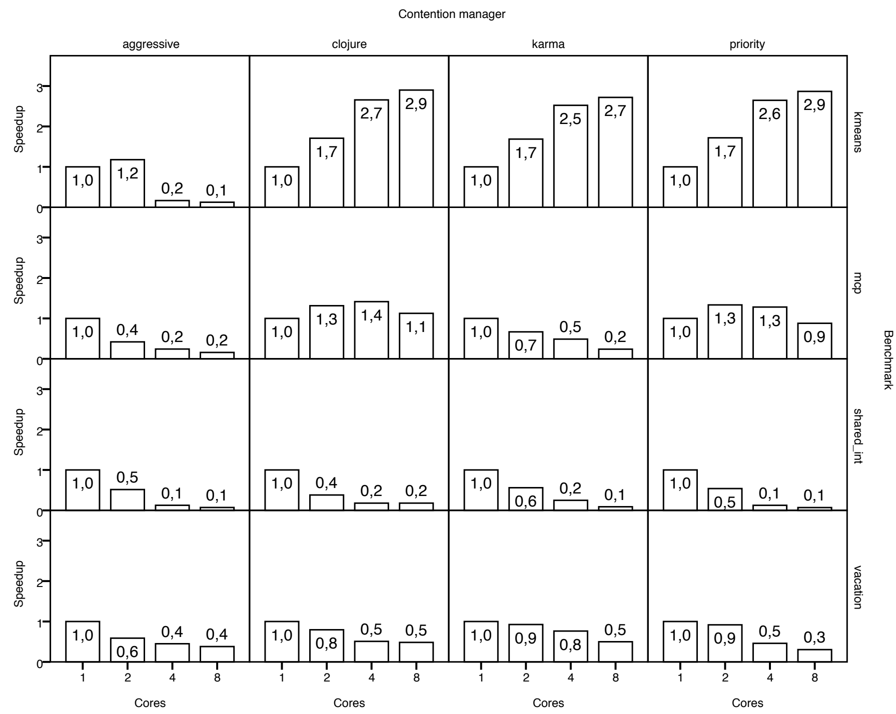

Figure 5.1: Speedup

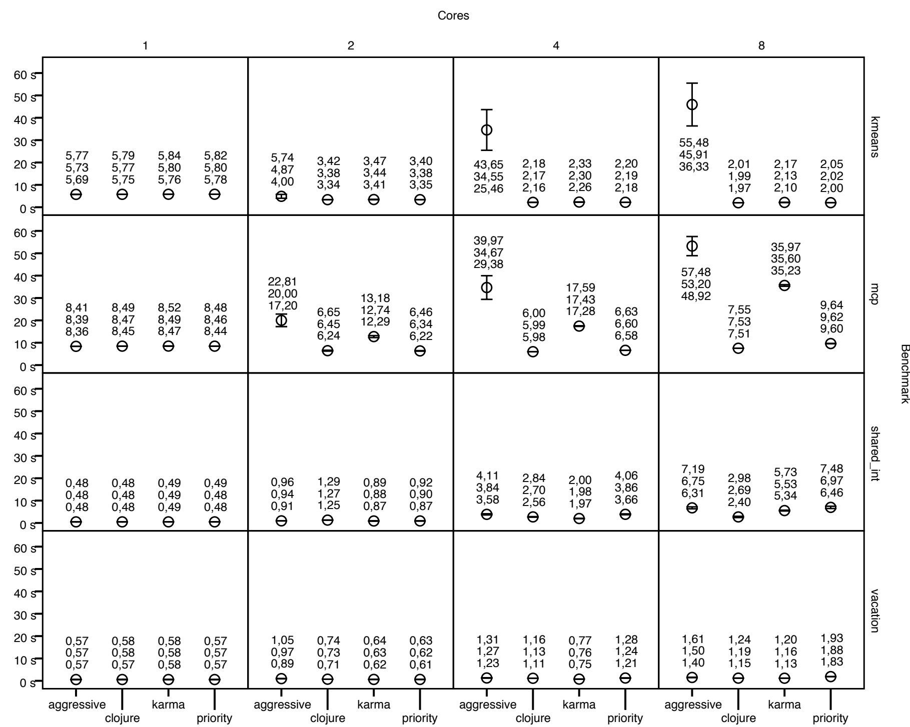

Figure 5.2: Benchmark duration.

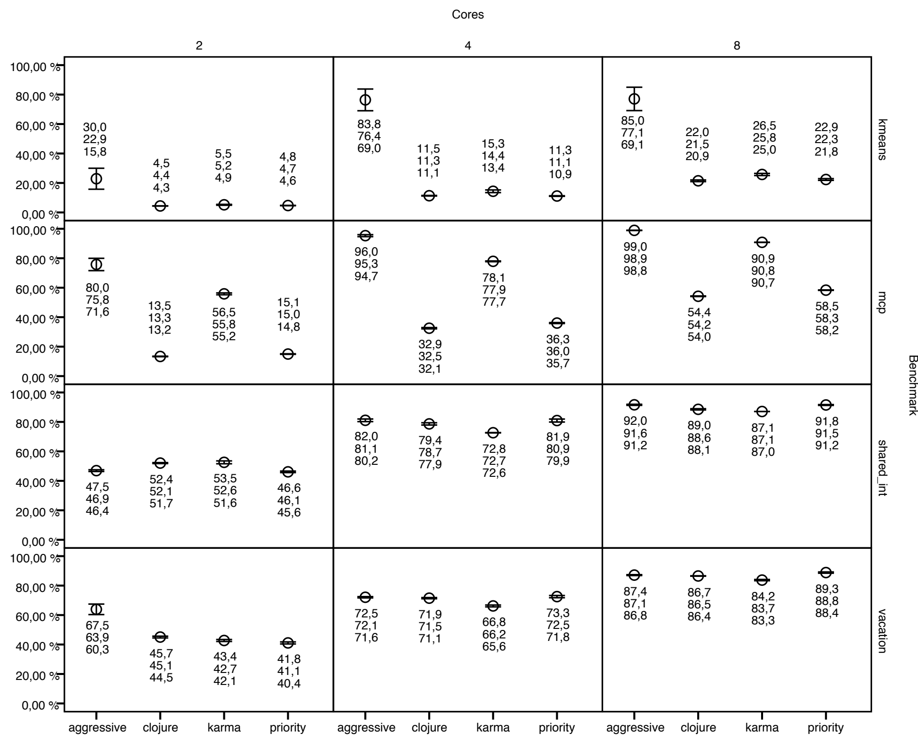

Figure 5.3: Wasted work.

in these benchmarks. For two cores in Vacation, Aggressive has significantly more wasted work compared to the other contention managers, but as the number of cores increases, they approach Aggressive, and for eight cores, Priority is slightly worse than Aggressive. Clojure and Priority are almost the same in the K-means benchmarks. Shared Integer and Vacation, shows a pattern where Priority is better than Clojure on two cores, but the reverse is true for four and eight cores, although the difference is small. Karma shows a significantly higher percentage of wasted work in Monte Carlo Pi compared to Clojure and Priority, but is still better than Aggressive. Karma is the best contention manager in terms of wasted work for four and eight cores in Shared Integer and Vacation.

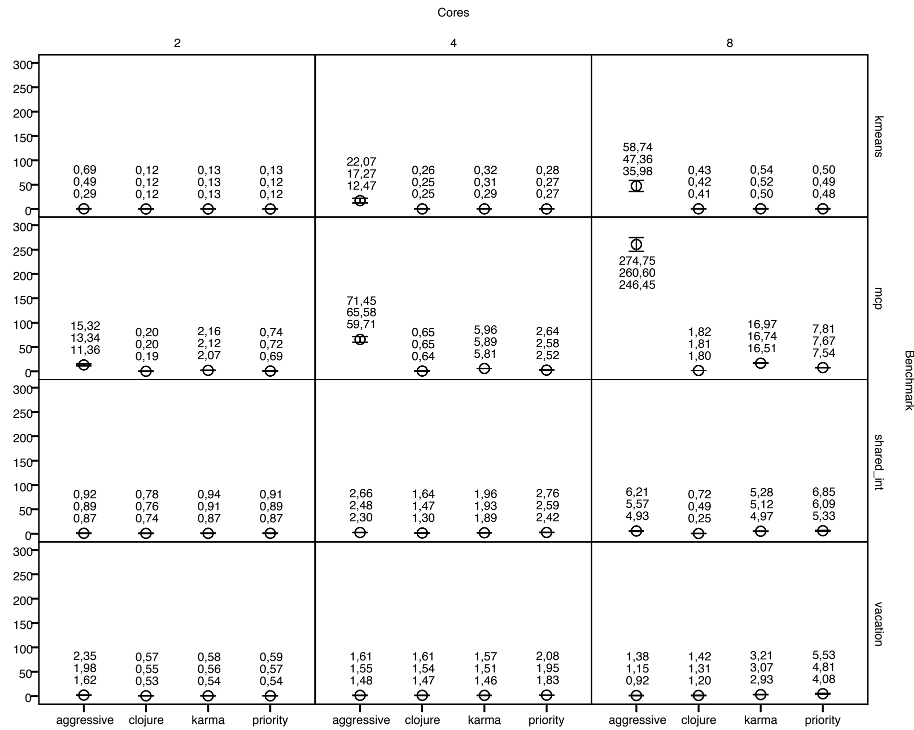

Figure 5.4: Retries per commit.

Figure 5.4 shows the average number of retries for every succesful commit. Aggressive causes a high number of retries per commit in K-means and Monte Carlo Pi as the number of cores increases. For four and eight cores in Vacation, Aggressive is comparable to Clojure, but on two cores Aggressive is significantly worse. The number of aborts per commit decreases for Aggressive on Vacation, but increases for the other contention managers. Clojure, Karma and Priority are similar on the K-means benchmark. In Monte Carlo Pi, Clojure is always best, followed by Priority and then Karma with more than twice the number of retries as Priority.

Figure 5.5 shows the average amount of time spent on contention managent

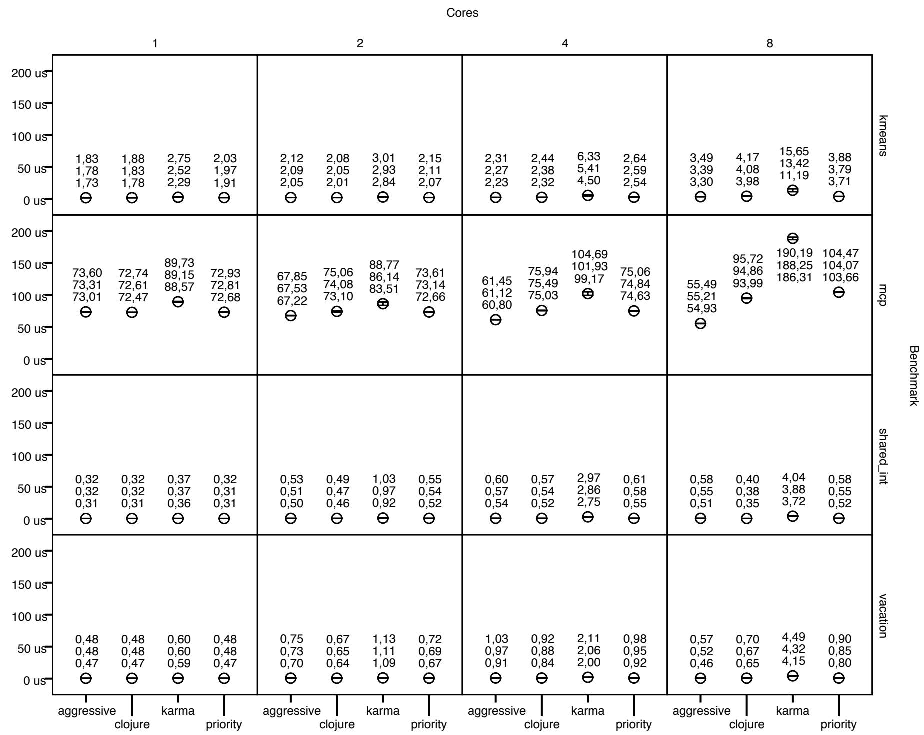

Figure 5.5: Contention management time.

per transaction. For K-means, Shared Integer and Vacation, Aggressive, Clojure and Priority spend roughly the same amount of time on contention management. Karma, on the other hand, consistently uses more time for contention management than all the others.

5.1 Discussion

It is obvious that Aggressive is unstable, especially when the number of cores increases. The reason is that Aggressive is more sensitive to the timing of thread interleavings because: (1) there is no waiting between retries, and (2) there is no priority ordering of transactions, i.e., any transaction can force any transaction to retry. This makes Aggressive more sensitive to non-determinism introduced through the underlying platform's thread scheduler. In some cases, the threads running transactions may execute in an order that causes many conflicts and thus retries (a livelock situation). In other cases, threads may just happen to execute in an order that causes less conflicts. Furthermore, the reason that we see more fluctuation in Monte Carlo Pi and K-means than the other benchmarks is that transactions in these are longer. Longer transactions have a wider window of time in which they can potentially be aborted, increasing the likeliness of them having to retry. Short transactions, such as those in Shared Integer and Vacation, are more likely to get their work done before being aborted.

Contrast this to Priority and Clojure. Here, the priority ordering ensures that the set of transactions that can cause a transaction T to restart becomes smaller over time. At some point, T will take precedence over any other transaction, simply because it is the oldest.

One of our results show that the benchmark duration, retries per commit ratio and percentage of wasted work is high for the Monte Carlo Pi benchmark on Karma. We attribute this to the fact that Karma waits for a constant amount of time for every attempt to acquire a ref. This happens in a loop until either the accumulated Karma is so high that the ref can be acquired, or until the transaction is aborted by another with higher priority. Since transactions are relatively long in Monte Carlo Pi and because many refs are writte, the potential for retries increases. Every retry incurs more wait time.

Compared to the contention management strategies we tested, it seems that Clojure's current contention management strategy provides the best overall performance. There are some cases where Priority, Karma and even Aggressive has slight advantages, but not enough to warrant substituting Clojure's current strategy with one of them. There may be other strategies that can provide better overall performance than the one currently offered in Clojure. Our results have clearly shown that the choice of contention management strategy is very significant for performance of STMs.

Chapter 6

Conclusion

Concurrent programming with locks is difficult and brings a well-known range of problems with it. In the introduction to this thesis, we set out to explore an alternative to lock-based concurrency known as *software transactional memory* (STM). We chose to investigate the STM implementation found in Clojure—a functional programming language with the characteristic Lisp syntax. Clojure has a unique set of features including software transactional memory that employs *multiversion concurrency control*, and persistent data structures. To our knowledge, this combination has not previously been investigated in academic literature.

One of the STM design issues that can have significant impact on performance is *contention mnagement*, which deals with strategies for handling conflicts between transactions. Our specific goal was to investigate the consequences of implementing a selection of different contention management strategies in Clojure's STM.

We began our investigation by presenting an overview of software transactional memory in general, including a history of its origins and a list of the most important design issues that we have come across in STM literature. Besides contention managent, *synchronization* is an important factor in the design of STMs. Much previous work has been invested in *obstruction-free* STMs, which do not employ locks in their implementations, but newer research indicates that faster STMs can be developed if locks are used. We also included a section describing multiversion concurrency control, which Clojure—as mentioned—uses in its STM.

In the subsequent chapter, we gave an introduction to Clojure the language. The reader was presented with a basic overview of Clojure's syntax, data structures and how to use the STM. We explained how the STM is implemented in Clojure and followingly placed Clojure in the landscape of STMs in accordance with the design issues mentioned previously. Finally, we discussed the contention management strategies that we chose to implement in Clojure's STM, namely Aggressive, Priority and Karma.

Our method for gauging the consequences of using the aforementioned contention managers is based on *benchmarking*. We presented an overview of the literature on benchmarking, including the problems and pitfalls associated

with benchmarking in a relatively new research area—a category that software transactional memory falls under. A choice in benchmarking is *scale*, i.e., whether to use micro or macro benchmarks. Much research on STM uses micro benchmarks, but some argue that macro benchmarks are more representative. In our work, we chose to use a combination of smaller programs to benchmark the different contention managers, and these programs were presented. In the final section about benchmarking, we gave a detailed overview of the statistically rigorous method we used in obtaining and analyzing results.

The results we have obtained from running our benchmarks showed that the choice of contention management strategy has a significant influence on the performance of Clojure's STM. Nonetheless, Clojure's original contention management strategy is best of the ones tested, except for a few specific cases. The Aggressive strategy appeared to cause many livelock situations resulting in a large amount of wasted time and longer benchmark durations. Karma and Priority both performed better in most cases, but did not top the contention management strategy already found in Clojure.

Chapter 7

Future Work

As our results show, the choice of contention management strategy has a significant influence on the performance of Clojure's STM. While the ones we tested mostly performed worse than Clojure's own, there is still a wide range of strategies in published research that have not been tested in Clojure's STM.

Our results also showed that in certain cases, some of the other contention management strategies were a little better than Clojure's. Some research has suggested that STMs should be adaptive so as to dynamically choose the strategy that suits the current situation best. Again, this is something that has not been done in Clojure's STM.

It is our impression that more work needs to be done to standardize benchmarking for software transactional memory. Because STM implementations are relatively complex and because the requirements posed by applications differ, it is impossible to design the one true benchmark that tests everything. Many questions are still left unanswered and much work is ahead to assess the viability of software transactional memory as a tool for writing concurrent programs in the future.

Bibliography

[1] E. Allen, D. Chase, C. Flood, V. Luchangco, J.W. Maessen, S. Ryu, and G.L. Steele Jr. Project fortress: A multicore language for multicore processors. *Linux Magazine*, pages 38–43, 2007.

[2] M. Ansari, K. Jarvis, C. Kotselidis, M. Lujan, C. Kirkham, and I. Watson. Profiling transactional memory applications. pages 11–20, feb. 2009.

[3] M. Ansari, C. Kotselidis, I. Watson, C. Kirkham, M. Luján, and K. Jarvis. Lee-tm: A non-trivial benchmark suite for transactional memory. *Algorithms and Architectures for Parallel Processing*, pages 196–207.

[4] Joe Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 1st edition, 7 2007.

[5] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[6] Stephen M. Blackburn, Kathryn S. McKinley, Robin Garner, Chris Hoffmann, Asjad M. Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony Hosking, Maria Jump, Han Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovik, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. Wake Up and Smell the Coffee: Evaluation Methodology for the 21st Century. *Communications of the ACM*, 51(8):83–89, 2008.

[7] Bryan Cantrill and Jeff Bonwick. Real-world concurrency. *Queue*, 6(5):16–25, 2008.

[8] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. Software transactional memory: why is it only a research toy? *Commun. ACM*, 51(11):40–46, 2008.

[9] JW Chung, H. Chafi, CC Minh, A. McDonald, B. Carlstrom, C. Kozyrakis, and K. Olukotun. The common case transactional behavior of multithreaded programs. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 266–277, 2006.

[10] P. Costanza, C. Herzeel, and T. D'Hondt. Context-oriented software transactional memory in common lisp. In *Proceedings of the 5th symposium on Dynamic languages*, pages 59–68. ACM, 2009.

[11] Cray Inc. *The Chapel Parallel Programming Language*. Online at `http://chapel.cray.com`.

[12] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

[13] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006. Electronic, no. page numbers.

[14] A. Dragojevic, R. Guerraoui, and M. Kapalka. Dividing transactional memories by zero. In *Proc. of the 3rd ACM SIGPLAN Workshop on Transactional Computing, Salt Lake City, UT*. ACM, 2008.

[15] Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, Intel Resarch, 2006.

[16] Andy Georges, Dries Buytaert, and Lieven Eeckhout. Statistically Rigorous Java Performance Evaluation. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-Oriented Programming Systems and Applications*, pages 57–76. ACM, 2007.

[17] Andy Georges, Lieven Eeckhout, and Dries Buytaert. Java Performance Evaluation Through Rigorous Replay Compilation. *SIGPLAN Not.*, 43(10):367–384, 2008.

[18] D. Grossman. The transactional memory/garbage collection analogy. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 695–706. ACM, 2007.

[19] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic contention management. *Distributed Computing*, 3724/2005:303–323, 2005.

[20] R. Guerraoui, M. Kapalka, and J. Vitek. STMBench7: a benchmark for software transactional memory. *ACM SIGOPS Operating Systems Review*, 41(3):315–324, 2007.

[21] Stuart Halloway. *Programming Clojure*. Pragmatic Bookshelf, 1st edition, 2009.

[22] Timothy L. Harris and Keir Fraser. Language support for lightweight transactions. In *OOPSLA*, pages 388–402, 2003.

[23] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 522–529, 2003.

[24] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, page 262. ACM, 2006.

[25] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[26] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, 1993.

[27] Carl Hewitt. Viewing control structures as patterns of passing messages. *Artificial Intelligence*, 8(3):323 – 364, 1977.

[28] Rich Hickey. Clojure website. Online at `http://clojure.org`.

[29] C. Hughes, J. Poe, A. Qouneh, and Tao Li. On the (dis)similarity of transactional memory workloads. pages 108–117, oct. 2009.

[30] K. Huppler. The Art of Building a Good Benchmark. *Performance Evaluation and Benchmarking*, pages 18–30, 2009.

[31] Guy L. Steele Jr. *Common Lisp the Language*. Digital Press, 2nd edition.

[32] J.M. Kraus and H.A. Kestler. Multi-core parallelization in Clojure: a case study. In *Proceedings of the 6th European Lisp Workshop*, pages 8–17. ACM, 2009.

[33] M. Kulkarni, M. Burtscher, K. Pingali, and C. Cascaval. Lonestar: A suite of parallel irregular programs. In *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2009.

[34] M. Kulkarni, P. Carribault, K. Pingali, G. Ramanarayanan, B. Walter, K. Bala, and L.P. Chew. Scheduling strategies for optimistic parallel execution of irregular programs. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 217–228. ACM, 2008.

[35] David J. Lilja. *Measuring Computer Performance: A Practitioner's Guide*. Cambridge University Press, 2000.

[36] V.J. Marathe, W.N. Scherer III, and M.L. Scott. Adaptive software transactional memory. *Distributed Computing*, pages 354–368.

[37] C.C. Minh, J.W. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization, 2008. IISWC 2008*, pages 35–46, 2008.

[38] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter F. Sweeney. Producing Wrong Data Without Doing Anything Obviously Wrong! In *ASPLOS '09: Proceeding of the 14th international conference on Architectural Support for Programming Languages and Operating Systems*, pages 265–276. ACM, 2009.

[39] Oracle Corporation. *Falcon Storage Engine Guide*. Online at `http://dev.mysql.com/doc/falcon/en/se-falcon-features.html`.

[40] Oracle Corporation. *Oracle Database Concepts—10g Release 2 (10.2)*. Online at `http://download-uk.oracle.com/docs/cd/B19306_01/server.102/b14220/consist.htm#i17881`.

[41] C. Perfumo, N. Sonmez, A. Cristal, O. Unsal, M. Valero, and T. Harris. Dissecting transactional executions in Haskell. In *The Second ACM SIGPLAN Workshop on Transactional Computing (TRANSACT)*. ACM, 2007.

[42] PostgreSQL Global Development Group. *PostgreSQL 8.4.2 Documentation*. Online at `http://www.postgresql.org/docs/8.4/interactive/mvcc-intro.html`.

[43] W.N. Scherer III and M.L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, page 248. ACM, 2005.

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

[45] A. Silberschatz, H. F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, 5th edition, 2006.

[46] S.E. Sim, S. Easterbrook, and R.C. Holt. Using benchmarking to advance research: A challenge to software engineering. page 83, 2003.

[47] D.D. Sleator and R.E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[48] Michael Sperber, R. Kent Dybvig, Matthew Flatt, and Anton van Straaten. Revised^6 Report on the Algorithmic Language Scheme. Online at `http://www.r6rs.org/`.

[49] The X10 Project. *X10*. Online at `http://x10-lang.org`.

[50] WF Tichy. Should computer scientists experiment more? *Computer*, 31(5):32–40, 1998.

[51] R. Mark Volkmann. Software Transactional Memory. Online at `http://java.ociweb.com/mark/stm/article.html`.

[52] I. Watson, C. Kirkham, and M. Lujan. A study of a transactional parallel routing algorithm. In *Parallel Architecture and Compilation Techniques, 2007. PACT 2007. 16th International Conference on*, pages 388–400, 2007.

[53] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *Proceedings of the 22nd annual international symposium on Computer architecture*, pages 24–36. ACM, 1995.

[54] Yi Zhao, Jin Shi, Kai Zheng, Haichuan Wang, Haibo Lin, and Ling Shao. Allocation Wall: A Limiting Factor of Java Applications on Emerging Multi-core Platforms. In *OOPSLA '09: Proceeding of the 24th ACM SIGPLAN conference on Object Oriented Programming Systems Languages and Applications*, pages 361–376. ACM, 2009.

[55] F. Zyulkyarov, A. Cristal, S. Cvijic, E. Ayguade, M. Valero, O. Unsal, and T. Harris. WormBench: A configurable workload for evaluating transactional memory systems. In *Proceedings of the 9th workshop on MEmory performance: DEaling with Applications, systems and architecture*, pages 61–68. ACM, 2008.

[56] F. Zyulkyarov, V. Gajinov, O.S. Unsal, A. Cristal, E. Ayguadé, T. Harris, and M. Valero. Atomic quake: using transactional memory in an interactive multiplayer game server. *ACM SIGPLAN Notices*, 44(4):25–34, 2009.

Appendix A

Benchmarks Source

This appendix gives the source code for our benchmarks. None of the programs are written for any sort of real-world usage but merely to stress Clojure's STM in certain ways. All of them are adapted to our needs in terms of running the benchmarks.

SharedInt

```
(ns
  benchmarks.shared-int
  "This namespace contains the simple SharedInt benchmark inspired by the one
  mentioned in the article 'Dissecting Transactional Executions in Haskell'. The
  benchmark is very simple; it spawns a number of threads matching the number of
  available cores. Each thread will continually increase a shared integer, until
  it reaches 100000."
  (:use benchmarks.core))

(def *shared-integer* (ref 0))

(def max-value 100000)

(defn agent-action
  "Represents the action performed by the agents to increase the value of the
  shared integer."
  [increments]
  (dotimes [_ increments]
    (dosync
      (alter *shared-integer* inc))))

(defn run-benchmark
  "Starts n agents, where n is the number of available cores on the machine.
  Each agent will run agent-action. This function will wait for each agent to
  finish, and then display the value of *shared-integer*."
  []
  (let [agents  (for [_ (range *core-count*)]
                  (agent (/ max-value *core-count*)))]
    (print "Starting" (count agents) "agents... ")
    (doseq [a agents]
      (send a agent-action))
    (println "Done.")
    (print "Waiting for agents to finish... ")
    (flush)
    (apply await agents)
    (println "Done.")
    (println "Value of shared int:" @*shared-integer*)))
```

Vacation

```
(ns
  benchmarks.vacation
  "This namespace contains a benchmark inspired by the one found in STAMP called
  vacation."
  (:use
     clojure.contrib.def
     benchmarks.core))

(def cars (ref {}))
(def rooms (ref {}))
(def flights (ref {}))
(def customers (ref {}))

(defvar *config* {:queries-per-action          4
                  :table-item-count            500000
                  :actions                     25000})

(defstruct item
           :price
           :total-available-count
           :reserved-count)

(defstruct reservation
           :type    ; :room, :car, :flight
           :id
           :price)

(defn make-table [max-relations]
  (loop [i      0
         table  (transient {})]
    (if (< i max-relations)
      (let [item  (struct item
                          (rand-int 1000)
                          (rand-int 500)
                          0)]
        (recur (inc i) (conj! table [i item])))
      (persistent! table))))

(defn initialize-tables!
  "Initializes the tables by adding a number of tuples to them. The number of
  tuples is specified in the :table-item-count config setting. Each tuple gets
  a key used to look it up in the table."
  []
  (let [relations       (*config* :table-item-count)
        cars-table      (make-table relations)
        rooms-table     (make-table relations)
        flights-table   (make-table relations)
        ; we represent the customer's reservation as a vector of reservation
        ; structs
        customers-table (zipmap (range relations) (repeat relations []))]
    (dosync
      (ref-set cars cars-table)
      (ref-set rooms rooms-table)
      (ref-set flights flights-table)
      (ref-set customers customers-table)))
  ; return nil for convenience - when using the REPL, we don't want to have the
  ; contents of any tables printed - they may be pretty large.
  nil)

(def item-type->item-table {:car      cars
                            :flight   flights
                            :room     rooms})

(defn reserve-item!
  "Reserves an item of a given type by increasing the item's :reserved-count and
  adding the reservation to the customer's list of reservations. Must run in a
  transaction and the tables must have been initialized. Returns the reservation
  (an instance of the reservation struct.) if succesful, nil if not."
  [item-type item-id customer-id]
```

```
  (let [item-table  (item-type->item-table item-type)
        customer    (@customers customer-id)
        item        (@item-table item-id)]
    (if (and item-table item customer)
      (let [free-items  (- (item :total-available-count)
                           (item :reserved-count))]
        (if (< 0 free-items)
          (let [price     (item :price)
                reserv    (struct reservation
                                  :car
                                  item-id
                                  price)]
            (alter item-table update-in [item-id :reserved-count] inc)
            (alter customers update-in [customer-id] conj reserv)
            ; return reservation to signify that items were available for the
            ; given id
            reserv)
          ; return nil to signify that there were no free items
          nil))
      ; return nil to indicate that neither the item-table, customer or item
      ; was found using the specified params
      nil)))

(defvar reserve-car!
  (partial reserve-item! :car)
  "Reserves a car. See reserve-item!.")
(defvar reserve-flight!
  (partial reserve-item! :flight)
  "Reserves a flight. See reserve-item!.")
(defvar reserve-room!
  (partial reserve-item! :room)
  "Reserves a room. See reserve-item!.")

(defn make-random-reservations!
  "Attempts to make a random number of reservations (a flight, room or car.) for
  a random customer."
  [n-max]
  (dotimes [_ (inc (rand-int n-max))]
    (let [item-type   ([:car :flight :room] (rand-int 3))
          item-table  (item-type->item-table item-type)
          item-id     (rand-int (count @item-table))
          customer-id (rand-int (count @customers))]
      (dosync
        (reserve-item! item-type item-id customer-id)))))

(defn delete-customer!
  "Deletes a specified customer and the customer's reservations. Returns the
  customer (i.e. a vector of the customer's reservations) that was deleted. Must
  run in a transaction."
  [customer-id]
  (let [customer (@customers customer-id)]
    (if customer
      (do
        ; remove customer
        (alter customers dissoc customer-id)
        ; decrease :reserved-count for the items that the customer had reserved
        (doseq [{item-type :type id :id} customer]
          (let [item-table    (item-type->item-table item-type)
                old-item      (@item-table id)
                updated-item  (update-in old-item
                                         [:reserved-count]
                                         dec)]
            (alter item-table assoc id updated-item)))
        customer-id)
      ; return nil if the customer wasn't found
      nil)))

(defn delete-random-customer!
  "Deletes a random customer and the customer's reservations. A random customer
  ID is chosen between 0 and (count @customers), which may mean that no customer
  is deleted in a random number of cases."
  []
```

```
  (let [customer (rand-int (count @customers))]
    (dosync
      (delete-customer! customer))))

(defn update-item-count!
  "Adds delta-count to the total available count of the specified item. If an
  item with the specified ID doesn't exist, if the count will become negative,
  or if the count will become less than :reserved-count, return nil. If it does
  exist, return the updated item."
  [item-type item-id delta-count]
  (let [item-table    (item-type->item-table item-type)
        item          (@item-table item-id)
        updated-item  (update-in item [:total-available-count] + delta-count)
        {new-total    :total-available-count
         reserved     :reserved-count}
                      updated-item
        consistent?   (and (<= 0 new-total)
                           (<= reserved new-total))]
    (when consistent?
      (alter item-table assoc item-id updated-item)
      updated-item)))

(defn update-random-item-count!
  "Attempts to update a random item's count in a random table a random number of
  times."
  []
  (let [item-type   ([:car :flight :room] (rand-int 3))
        item-table  (item-type->item-table item-type)
        rand-op     ([+ -] (rand-int 2))
        rand-count  (rand-op 0 (rand-int 100))
        rand-id     (rand-int (count @item-table))]
    (dosync
      (update-item-count! item-type rand-id rand-count))))

(defn client-action [actions queries-per-action]
  (dotimes [_ actions]
    (let [possible-actions  [#(make-random-reservations! queries-per-action)
                             delete-random-customer!
                             update-random-item-count!]
          action            (possible-actions
                              (rand-int (count possible-actions)))]
      (action))))

(defn initialize-benchmark
  "Run before benchmarking."
  []
  (initialize-tables!))

(defn run-benchmark
  "Run benchmarks. Remember to call initialize-benchmark beforehand."
  []
  (println "Number of cores:" *core-count*)
  (let [agents (for [_ (range *core-count*)]
                 (agent (/ (*config* :actions) *core-count*)))]
    (println "Number of agents:" (count agents))
    (flush)
    (doseq [a agents]
      (send a client-action (*config* :queries-per-action)))
    (apply await agents)))
```

K-means

```
(ns benchmarks.kmeans
  "A benchmark based on a parallelized version of the k-means (Lloyd's)
  algorithm."
  (:use
     clojure.set
     benchmarks.core))

(defstruct cluster :centroid :data)
```

```
(def clusters nil)

; Copied from Clojure 1.2
(defn partition-all
  "Returns a lazy sequence of lists like partition, but may include
  partitions with fewer than n items at the end."
  {:added "1.2"}
  ([n coll]
     (partition-all n n coll))
  ([n step coll]
     (lazy-seq
      (when-let [s (seq coll)]
        (cons (take n s) (partition-all n step (drop step s)))))))

(defn generate-random-data
  "Returns a seq of random 2-dimensional vectors. The seq has n elements, and
  the vectors lie within the bounds of x-max and y-max."
  [n x-max y-max]
  (for [_ (range n)]
    [(rand-int x-max) (rand-int y-max)]))

(defn distance
  "Calculates the distance between two n-dimensional coordinates (vectors)."
  [p q]
  (let [squared-coord-diffs (map #(Math/pow (- %1 %2) 2) p q)]
    (Math/sqrt (reduce + squared-coord-diffs))))

(defn closest-cluster
  "Gets the ref of the cluster in cluster-refs that is closest to v."
  [v cluster-refs]
  (apply min-key #(distance v (:centroid (deref %))) cluster-refs))

(defn unique-initial-centroids
  "Attempts to find k unique initial centroids by iterating over the data points
  in data. Stops as soon as the centroids are found or when all data has been
  iterated through.

  This function has O(n) complexity, where n is the number of data points.
  Hopefully the data is varied enough that n is never reached."
  [k data]
  (let [data-count (count data)]
    (loop [centroids  #{}
           iter       0]
      (if (and (< iter data-count)
               (< (count centroids) k))
        (recur (conj centroids (nth data iter))
               (inc iter))
        centroids))))

(defn initial-clusters
  "Creates a seq of k initial clusters, i.e., clusters where the first cluster
  contains all data points (in a set) and the others are empty. Each of the
  clusters have a centroid randomly selected from the data set. Clusters will
  not share a centroid."
  [k data]
  (let [centroids     (unique-initial-centroids k data)
        first-cluster (struct cluster (first centroids) (set data))
        rest-clusters (for [c (rest centroids)]
                        (struct cluster c #{}))]
    (cons first-cluster rest-clusters)))

(defn centroid
  "Calculates the centroid from the vectors in data. The centroid is calculated
  by finding the arithmetic mean of the vectors. This function assumes that all
  vectors have the same dimensions as the first one."
  [data]
  (let [d               (count (first data))
        component-seqs  (for [i (range d)] (map #(% i) data))
        component-sums  (map #(reduce + %) component-seqs)
        component-means (map #(/ % (count data)) component-sums)]
    (vec component-means)))
```

```
(defn assign-clusters!
  "Assigns each vector in data to a cluster in cluster-refs by first removing
  the vector from any previous cluster it may have been in and then adding it to
  the closest one."
  [data cluster-refs]
  (doseq [subset (partition-all 2 data)]
    ; add 2 points per transaction
    (dosync
      (doseq [v subset]
        ; add v to closest cluster
        (let [closest (closest-cluster v cluster-refs)]
          (alter closest update-in [:data] #(conj % v)))))))

(defn centroids-converged?
  "Determines whether the seq of centroids in a are equal to the seq of
  centroids in b. The order of the centroids is significant."
  [a b]
  (every? true? (map = a b)))

(defn assign-centroid!
  "Calculates and assigns a cluster's centroid."
  [cluster-ref]
  (dosync
    (let [cluster-data (:data @cluster-ref)]
      (when (seq cluster-data)
        (alter cluster-ref assoc :centroid (centroid cluster-data))))))

(defn complete-data-set
  "Returns a set containing all data vectors from all clusters in clusters.
  (NOTE: The clusters are expected not to be refs.)"
  [clusters]
  (let [data-sets (map :data clusters)]
    (apply union data-sets)))

(defn cluster-refs->centroids
  "Gets a seq of centroids from a seq of cluster refs."
  [cluster-refs]
  (doall (map #(:centroid (deref %)) cluster-refs)))

(defn initialize-benchmark
  "Initializes the benchmark."
  [k data]
  (alter-var-root (var clusters)
                  (fn [_]
                    (doall (map ref (initial-clusters k data)))))
  ; return nil for convenience (no printing of clusters)
  nil)

(defn run-benchmark
  "Runs the k-means benchmark, creating k clusters in a maximum of max-iter
  iterations using the specified data."
  [max-iter]
  (println "Running k-means on" *core-count* "cores")
  (let [cluster-refs      clusters
        data-set          (complete-data-set (map deref cluster-refs))
        data-set-size     (count data-set)
        centroid-assigner (fn [cluster-refs]
                            (dosync
                              (doseq [c cluster-refs]
                                (assign-centroid! c))))]
    (loop [iter 0]
      (if (< iter max-iter)
        (let [old-centroids   (cluster-refs->centroids cluster-refs)
              partition-size  (/ data-set-size *core-count*)
              partitions      (partition-all partition-size data-set)
              data-agents     (doall (map agent partitions))]

          ; empty clusters before assigning data to them
          (dosync
            (doseq [c cluster-refs]
              (alter c assoc-in [:data] #{})))
```

```
      ; first we assign the data to the right clusters
      (doseq [a data-agents]
        (send a #(assign-clusters! % cluster-refs)))
      (apply await data-agents)

      ; then we calculate the new centroids
      (let [partition-size  (/ (count cluster-refs) *core-count*)
            partitions      (partition-all (/ (count cluster-refs)
                                              *core-count*)
                                           cluster-refs)
            centroid-agents (map agent partitions)]
        (doseq [a centroid-agents]
          (send a centroid-assigner))
        (apply await centroid-agents))

      (let [new-centroids (cluster-refs->centroids cluster-refs)]
        (if (centroids-converged? old-centroids new-centroids)
          (do
            (println "Convergence! Iterations:" (inc iter))
            ; centroids didn't change - return resulting cluster-refs
            cluster-refs)
          ; another iteration is necessary and allowed
          (recur (inc iter))))))))
```

Monte Carlo Pi

```
(ns benchmarks.monte-carlo-pi
  (:use
     [clojure stacktrace]
     [benchmarks core]))


; The number of cells along each axis. The grid is a square.
(def *grid-size* 256)

(def grid nil)

(defn create-grid
  "Returns a square grid of the size n x n as a vector of vectors."
  [n]
  (for [_ (range n)]
    (for [_ (range n)]
      (ref ()))))

(defn insert-point
  "Inserts the point x,y in the grid."
  [x y grid]
  (let [col       (Math/floor x)
        row       (Math/floor y)
        cell-ref  (nth (nth grid col) row)]
    (alter cell-ref conj [x y])))

(defn in-circle
  "Returns true if the point x,y lies within the circle."
  [point]
  (let [center    (/ *grid-size* 2)
        [x y]     point
        dx        (- x center) ; adjust for displacement of coordinate system.
        dy        (- y center)
        distance  (Math/sqrt (+ (* dx dx) (* dy dy)))]
    (<= distance center)))

(defn calculate-pi
  "Calculates the value of pi based on the ratio between the number of points
  that lie within the unit-circle and the total number of points in the grid.
  Assuming the points are normally distributed, the ratio will approach pi/4.
  Multiplying by 4 yields the approximated value of pi."
  [grid]
  (let [all-points        (apply concat (map deref (apply concat grid)))
```

```
        points-in-circle  (count (filter in-circle all-points))
        points-total      (count all-points)]
    (* 4 (/ points-in-circle points-total))))

(defn agent-action
  "This is the function that agents execute. An agent performs txs
  transactions that each insert points number of points in to grid."
  [_ grid txs points-count]
  (dotimes [_ txs]
    (let [points (doall (for [_ (range points-count)]
                          [(rand *grid-size*)
                           (rand *grid-size*)]))]
      (dosync
        (doseq [[x y] points]
          (insert-point x y grid))))))

(defn initialize-benchmark []
  (println "Initialized!")
  (alter-var-root (var grid)
                  (fn [_]
                    (create-grid *grid-size*)))
  ; return nil for convenience - no printing of grid
  nil)

(defn run-benchmark
  []
  (let [total-txs     4000
        txs-per-agent (/ total-txs *core-count*)
        points-per-tx 128
        agents        (for [_ (range *core-count*)] (agent nil))]
    (doseq [a agents]
      (send a agent-action grid txs-per-agent points-per-tx))
    (println *core-count* "agents working...")
    (flush)
    (apply await agents)))
```

Appendix B

Benchmark Tools

During the course of this project, we have developed several tools to run our benchmark. This appendix lists the source code for two important tools we have used, and we have included them to increase the transparency of our benchmark method.

Benchmark Runner Script

This is a Bash script used to run all the necessary benchmarks.

```
#!/bin/bash

CLASSPATH='clojure-contrib.jar:../src'
JAVA_OPTS='-server -Xmx14G -Xms14G -XX:NewSize=13G -XX:SurvivorRatio=64
           -XX:MinHeapFreeRatio=80 -XX:MaxHeapFreeRatio=95
           -XX:+UseParallelGC -XX:+PrintGCDetails -XX:+PrintGCTimeStamps'

CLOJURE_VARIATIONS=(aggressive karma priority clojure)
BENCHMARKS=(kmeans mcp vacation shared_int)
COV_WINDOW=4
MAX_ITERATIONS=20
CORES=(1 2 4 8)
MAX_JVM_INSTANCES=10
COEFF_VARIATION="0.01"
LOG_FILE=log.txt

for cm in ${CLOJURE_VARIATIONS[@]}
do
    for bm in ${BENCHMARKS[@]}
    do
        for c in ${CORES[@]}
        do

            for ((i=1; i<=$MAX_JVM_INSTANCES; i+=1))
            do
                (
                echo
                echo "=========================================="
                echo
                echo "  Contention manager: $cm"
                echo "  Benchmark:          $bm"
                echo "  Cores:              $c"
                echo "  JVM invocation:     $i"
                echo
                echo "==========================================="
                echo
```

```
                java $JAVA_OPTS -cp "$CLASSPATH:$cm.jar" \
                    clojure.main \
                    "run_$bm.clj" \
                    $cm \
                    $bm \
                    $c \
                    $i \
                    $COEFF_VARIATION \
                    $COV_WINDOW \
                    $MAX_ITERATIONS
                sleep 1
                ) | tee -a $cm-$LOG_FILE
            done

        done
    done
done
```

Scaffolding

To hook benchmarks up with the instrumentation the following scaffolding code is used. We first present a scaffolding API, and subsequently the scripts that call this API to initialize and run the benchmarks.

Scaffolding API

```
(ns benchmarks.scaffolding
  "Contains utilities to run benchmarks. This namespace requires that the
  customized, instrumented version of Clojure is on the classpath."
  (:use benchmarks.core)
  (:import d608a.Instrumentation)
  (:import d608a.Statistics)
  (:import java.util.Calendar))

(defn toInt [s]
  (Integer/parseInt s))

(defn toDouble [s]
  (Double/parseDouble s))

(def args *command-line-args*)

(def *cm-name* (nth args 0))
(def *bm-name* (nth args 1))
(def *cores* (toInt (nth args 2)))
(def *invocation* (toInt (nth args 3)))
(def *cov* (toDouble (nth args 4)))
(def *cov-window* (toInt (nth args 5)))
(def *max-iterations* (toInt (nth args 6)))

(defn initialize-instrumentation []
  "Initializes the instrumentation. Currently only allocates
  space for storing collected data."
  (Instrumentation/allocateLogSpace 100000))

(defn get-filename [iteration]
  "Returns the name of the file in which to store the data from a benchmark."
  (str *cm-name* "-" *bm-name* "-" *cores* "-" *invocation* "-" iteration ".txt")
      )

(defn timed-run [init-fn benchmark-to-run iteration]
  "Executes benchmark-to-run once, writes the output to a file and finally
   returns the duration of the benchmark in nanoseconds."
  (init-fn)
```

```
  (Instrumentation/resetLog)
  (System/gc)
  (let [t (.. (Calendar/getInstance) (getTime))]
    (println "*** BENCHMARK BEGIN @ "
             (.getHours t) ":"
             (.getMinutes t) ":"
             (.getSeconds t)))
  (Instrumentation/beginBenchmark)
  (benchmark-to-run)
  (Instrumentation/endBenchmark)
  (let [t (.. (Calendar/getInstance) (getTime))]
    (println "*** BENCHMARK END @ "
             (.getHours t) ":"
             (.getMinutes t) ":"
             (.getSeconds t)))
  (Instrumentation/dumpData (get-filename iteration))
  (- Instrumentation/benchmarkEnd Instrumentation/benchmarkBegin))

(defn multiple-timed-runs [init-fn benchmark-to-run k]
  "Executes benchmark-to-run k times and returns a vector the durations
  for each of the k executions of the benchmark."
  (doall (vec
    (for [i (map inc (range k))]
      (timed-run init-fn benchmark-to-run i)))))

(defn steady-state?
  "Determines if steady state has been reached by calculating the coefficient of
      variation
  for durations. Steady state is achieved when the coefficient of variation of
      the durations
  is below the value specified in the command line."
  [durations]
  (let [threshold *cov*
        current-cov (Statistics/coefficientOfVariation (long-array durations))]
    (println "Current coefficient of variation: " current-cov)
    (< current-cov threshold)))

(defn run-until-steady
  "Executes benchmark-to-run until the *cov-window* last iterations are
      considered steady state
  executions or until *max-iterations* has been performed."
  [init-fn benchmark-to-run]
  (loop [durations (multiple-timed-runs init-fn benchmark-to-run *cov-window*)
         iteration (inc *cov-window*)]
    (println "Durations: " durations)
    (if-not (steady-state? durations)
      (if (<= iteration *max-iterations*)
        (recur (conj (subvec durations 1)
                     (timed-run init-fn
                                benchmark-to-run
                                iteration))
               (+ iteration 1))))))

(defn dobenchmark
  "Run a benchmark represented by benchmark-fn. The init-fn function is called
  before each run."
  [init-fn benchmark-fn]
  (initialize-instrumentation)
  (run-until-steady init-fn benchmark-fn)
  (System/exit 0))
```

Clojure Script for running Shared Integer

```
; Shared integer benchmark

(use 'benchmarks.shared-int
     'benchmarks.scaffolding)

(require '[benchmarks.core :as bmcore])

(binding [bmcore/*core-count* benchmarks.scaffolding/*cores*]
```

```
  (dobenchmark (fn []) run-benchmark))
```

Clojure Script for running Vacation

```
; Vacation benchmark

(use 'benchmarks.vacation
     'benchmarks.scaffolding)

(require '[benchmarks.core :as bmcore])

(binding [bmcore/*core-count* benchmarks.scaffolding/*cores*]
  (initialize-benchmark)
  (dobenchmark (fn []) run-benchmark))
```

Clojure Script for running K-means

```
; K-means Benchmark

(use 'benchmarks.kmeans
     'benchmarks.scaffolding
     'clojure.contrib.duck-streams)

(import 'java.io.File
        'java.io.PushbackReader)

(require '[benchmarks.core :as bmcore])

(declare *data*)

(defn run []
  (binding [bmcore/*core-count* benchmarks.scaffolding/*cores*]
    (let [init-fn #(initialize-benchmark 128 *data*)]
      (dobenchmark init-fn #(run-benchmark 3)))))

(let [data-file (File. "kmeans.dat")]
  (if (.exists data-file)
    (binding [*data* (read (PushbackReader. (reader data-file)))]
      (run))
    (let [data (generate-random-data 1000 1000 1000)]
      (with-out-writer data-file
         (pr data))
      (binding [*data* data]
        (run)))))
```

Clojure Script for running Monte Carlo Pi

```
; Monte Carlo Pi Benchmark

(use 'benchmarks.monte-carlo-pi
     'benchmarks.scaffolding)

(require '[benchmarks.core :as bmcore])

(binding [bmcore/*core-count* benchmarks.scaffolding/*cores*]
  (dobenchmark initialize-benchmark run-benchmark))
```