Copyright

by

Mark Vincent Baum

2011

The Report Committee for Mark Vincent Baum
Certifies that this is the approved version of the following report:

Refactoring for Software Transactional Memory

APPROVED BY
SUPERVISING COMMITTEE:

Supervisor:

____
Miryung Kim

____
Dewayne Perry

Refactoring for Software Transactional Memory

by

Mark Vincent Baum, BSB

Report

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

December 2011

Dedication

This work is dedicated to my wife Fabiana, and my daughter Hanna.

Abstract

Refactoring for Software Transactional Memory

Mark Vincent Baum, M.S.E.
The University of Texas at Austin, 2011
Supervisor: Miryung Kim

Software transactional memory (STM) is an optimistic concurrent lock free mechanism that has the potential of positively transforming how concurrent programming is performed. STM, despite its many desirable attributes, is not yet a ubiquitous programming language feature in the commercial software domain. There are many implementation challenges with retrofitting STM into pre-existing language frameworks. Furthermore, existing software systems will also need to be refactored in order to take advantage of STM's unique benefits. As with other time consuming and error prone refactoring processes, refactoring for STM is best done with automated tool support; it is the aim of this paper to propose such a tool.

Table of Contents

List of Tables vii

List of Figures viii

1. Introduction 1
    1.1 Concurrent Programming Challenges 2
    1.2 Using Transactions 4
    1.3 Software Transactional Memory (STM) 5
        1.3.1 Abstraction 5
        1.3.2 Safety 6
        1.3.3 Concurrency 7
    1.4 Current State 8

2. Refactoring for STM 9
    2.1 Refactoring 9
    2.2 Design 9
    2.2 .1 Integrated Development Environment 11
    2.2 .2 STM Refactoring Engine 14
    2.2 .3 Precondition Checks 15
        2.2.3.1 Transform Program 15
        2.2.3.2 Throw Error 16

3. Related Work 17

4. Conclusion 24

5. Appendix 25
    5.1 Measuring Concurrency 25
        5.1.1 Amdahl's Law 25
        5.1.2 Related Work 28
        5.1.3 Conclusion 30

References 32

List of Tables

Table 3.1: Related concurrency refactoring tools and their respective usage domain. 23

List of Figures

Figure 2.1 Interactive Refactoring 10

Figure 2.2 Atomic Block Refactoring 11

Figure 2.3 Create Sub Object Refactoring 13

Figure 2.4 Add Atomic Retry Refactoring 14

Figure 3.1 ConvertToConcurrentHashMapRefactoring 19

Figure 3.2 Convert to Reentrant Lock refactoring 21

Figure 3.3 Complex numbers to a ParallelArray Refactoring 22

Figure 5.1 Amdahl's Law, speedup as a function of CPU cores 27

1. Introduction

Concurrent programming has been an active research topic for over thirty years, but only in the last five years has the industrial software community concerned itself with maximizing program concurrency to improve the scalability of software systems. There are several reasons for this, but the main factor is attributed to the work of Gordon Moore, to whom Moore's Law is attributed. Moore's Law, first published in 1965, predicts that chip component integration will double approximately every 2 years [1]. CPU clock speeds, execution, and cache optimizations have followed a similar trend for over a decade. It was the combination of these optimizations that enabled consistent performance gains in sequentially executing applications systems running on a single core CPU. The increases in single core CPU clock speeds ceased in 2005 after the release of the Intel Pentium 4 Processor. Since then, chip manufacturers have responded to the clock speed ceiling by increasing the capacity of CPU chips. This was done by adding more processor cores to each chip, effectively increasing the rate of executions per second without increasing clock speeds.

However, even with more CPU cores at an application's disposal, performance gains cannot inherently be realized without adopting an efficient concurrent programming model to enable the management of multiple concurrent thread executions. Concurrent programming models attempt to address the complex challenges of having two or more activities executing within the same interval of time by providing a set of tools in the form of algorithms and patterns to help manage the interleaving of concurrently executing threads.

1.1 CONCURRENT PROGRAMMING CHALLENGES

The challenges of concurrent programming are not immediately apparent. It is easy to conceptualize that one thread of execution can perform some computations while another performs some other computations independently. In some programming models, such as coordination and actor based message passing systems, this simple representation of parallel thread executions is completely valid because there are no global objects being accessed concurrently. While these thread safe programming models certainly have their benefits, they are rarely applied to solve the diverse and quickly changing set of challenges that exist in the larger commercial software industry. The most pervasive programming model in this domain is the object based, mutable shared memory model.

In a memory model which is immutable (i.e. unshared) and unchanging, a thread may read the statement x=1 which results in the allocation of memory on the program stack name x for which its value is 1. Where ever x is used by any thread, its value will always be 1. This is in contrast to a mutable shared memory model where x is the name of a location and not the value. In a mutable model, the value at location x may be changed by any thread, at any time. This can create inconsistencies in the value of x across different threads of execution, leading to unpredictable program behaviors and eventual incorrectness of the program. It is clear that a thread synchronization method is required to help the proper reading and writing of data to a shared memory location.

The most common thread synchronization method is the use of a locking protocol. A locking protocol is a mechanism to protect and to access protected data resources which are generally enabled in a set of language constructs. A protected data resource is placed inside what is considered a critical section to ensure that only one thread at a time may access and modify the data resource.

Example 1.1

```
myLock.acquire();
    /* start of critical section */
    if(myBalance<AmountIWant)
    throw OutOfMoneyError();
    else
    myBalance -= AmountIWant;
    /* end of critical section */
myLock.release();
```

There are two operations in the above sample of a critical section that require protection: the reading and writing of a value to *myBalance*. Using locks serializes the portion of the program where they are used by blocking all thread requests to acquire the lock while the lock is in use by another thread. Although this protocol does indeed work, a common issue in lock based protocols is the occurrence of deadlocks and livelocks. A deadlock is a condition where the threads in a system are unable to make progress because they require a resource they will never receive. For instance, if one thread acquires a lock and enters the critical section and an unhandled error is thrown while executing one of the statements in the critical section, the thread with the lock will abort before it can release the lock. When subsequent threads attempt to acquire the lock, they cannot, and therefore deadlock, or infinitely wait to acquire the lock. When a deadlock occurs the only remediation is the termination of the deadlocked threads. A livelock is similar to a deadlock, except that the state of the two threads involved in the livelock constantly

changes with regard to the other process. A more practical example of a livelock occurs when two people meeting in a narrow corridor, with each trying to be polite by moving aside to let the other pass, but instead both sway from side to side without making any progress because they always both move the same way at the same time wasting CPU cycles. These simplified examples convey the problems associated with locking protocols; implementing solutions to these problems correctly is however very complex and difficult.

1.2 USING TRANSACTIONS

Language designers recognize that database management systems have been successfully dealing with concurrent queries for several decades. Not only do database management systems achieve high degrees of performance and scalability, they do in ways that the synchronization work is completely invisible to the programmer providing the key attribute of abstraction.

The key concept in parallel database operations is the notion of a transaction. Transactions are language abstractions that possess particular properties [3]. Most notable of these properties is atomicity, which means that either all statements in a transaction execute successfully or none at all do so. A transaction executes as it is the only computation accessing a data source providing thread safety. While other transactions may be executing simultaneously, the transactional programming model in database management systems prevent transactions with dependencies on each other from executing concurrently. In other words, database software and hardware work together to serialize transactions that attempt to access and modify the same data while allowing other transactions to run in parallel which do not maximizing concurrency. Transactional

database systems are highly complex implementations hidden behind a relatively simple programming construct. Using this notion of transactions, language designers began developing Software Transactional Memory (STM) to implement the concepts of transactions into the application domain [4] [5] [6].

1.3 SOFTWARE TRANSACTIONAL MEMORY (STM)

STM allows a programmer to group a set of operations as a single atomic computation to update a program's volatile memory, i.e. program state. It is the role of the transactional memory system to provide the illusion that all operations of a committed transaction appear as if they were executed instantaneously to any other thread in the program. All operations of an aborted transaction, however, appear as if they never took place. Similar to transactional database systems, STM systems also provide the key attributes of *Abstraction*, *Safety*, and *Concurrency*.

1.3.1 Abstraction

At the minimum, implementing STM required a set of primitives [6] to access, read and write to shared system memory.

These include:

- A primitive to perform read from a shared (global) memory location and copy the value to a thread's private memory.
- A primitive to update private memory.
- A primitive to commit a transaction from private memory to shared memory, which enables all threads to see the changes at the same time.

- A primitive to validate a transaction.
- A primitive to abort a transaction, and perform a rollback of any changes.

In addition to these using these primitives, the STM compiler also keeps track of the memory at which values are being read from and written to, allowing multiple transactions to execute at the same time as long as there are no memory conflicts. If instead the programmer had to account for all of this, it would be a huge burden and would most likely lead to incorrect programs. However with STM, all of the transaction management is delegated to the compiler. The programmer simply needs to encapsulate statements that should be transactionalized in an *atomic* keyword as shown by Example 1.2.

Example 1.2

```
Atomic{
        if(myBalance<AmountIWant)
                throw YouAreOutOfMoneyError();
        else
        myBalance -= AmountIWant;
}
```

1.3.2 Safety

Implementing lock based thread synchronization as a protocol requires a precise ordering of lock acquisitions or the likelihood of deadlocks and livelocks is high. This level of diligence is difficult to realize for a few reasons. Software systems are often

written by a team of programmers; therefore, maintaining the precise ordering of lock acquires and releases becomes a difficult to follow and enforce across the entire team. Also, in attempting to fix data inconsistencies due to the lack of proper locking, programmers may overcompensate by removing too many thread interleavings, thus introducing the potential for deadlocks. Another consideration is in testing. Testing for deadlocks is notoriously difficult and costly because they happen under very specific thread schedules that may only be caught one time out of one thousand executions of the program code. STM enables safe access to protected data resources and eliminates the hazards of using a lock based protocol by completely removing the use locks and replacing them with atomic updates.

1.3.3 Concurrency

Lock based protocols follow the pessimistic locking model. A pessimistic model uses locking to prevent shared access to resources [7] under the assumption that multiple transactions concurrently accessing the same resource are likely to lead to consistency-threatening conflicts. Aptly named, the pessimistic model attempts to minimize concurrency through an approximation of conflict detection. It can, however, be overly conservative when the system goals are to maximize potential concurrency. Under a pessimistic model, a thread would lock an entire data structure, like an array or hash map, even if it was only accessing a single field in the data structure. With this example, it can be easily understood how pessimistic locking decreases a system's potential concurrency.

By contrast, STM implements an optimistic concurrency control model. An optimistic model allows concurrent transactions to perform independent updates to the same data structure. The optimistic model assumes that the modification of the same field

in a multi-field data structure is not likely, but still possible. Therefore, it provides support for resolving detected conflicts when committing a transaction from private memory to shared memory. This may be implemented in many ways. One typical method for conflict resolution in the optimistic model is to abort the transaction, log the event in a file, and recommit the transaction at some time in the future.

1.4 CURRENT STATE

Software Transactional Memory is an ongoing research topic that has been gaining momentum over the last several years as a viable mechanism for facilitating lock free concurrent programming. There have been several notable attempts at implementing STM by research teams with varying degrees of success. Intel Corporation has created a C++ STM compiler prototype and published a draft specification for STM language constructs [8]. The Scalable Synchronization Research Group at Sun Microsystems has also published its version of STM language extensions for a C++ compiler [9]. Microsoft's research project to implement STM as a language feature on its .NET platform was, unlike the Intel and Sun Microsystems implementations, not released with the latest .NET 4.0 libraries [12]. The future of STM running on the .NET platform is currently unclear, but the one constant trend in programming language evolution has been the raising of the abstraction levels and STM as a language feature does this very thing. As the positive benefits remain unchanged, in time it is realistic to expect that the difficulties of implementing STM on .NET will have been resolved and become part of the language specification.

2. Refactoring for STM

The aim of this paper is to propose the notion of an IDE refactoring browser plug-in that will support the conversion of program code to code that incorporates STM language features. An elementary design is provided in this paper for such a tool.

Similar to *Concurrencer* [15], the STM refactoring tool will be an interactive, Integrated Development Environment (IDE) plug-in. In this case, it shall integrate with the Visual Studio IDE instead of Eclipse, and provide automated support for refactoring for STM using references provided in Microsoft's experimental transactional memory framework, STM.NET.

2.1 Refactoring

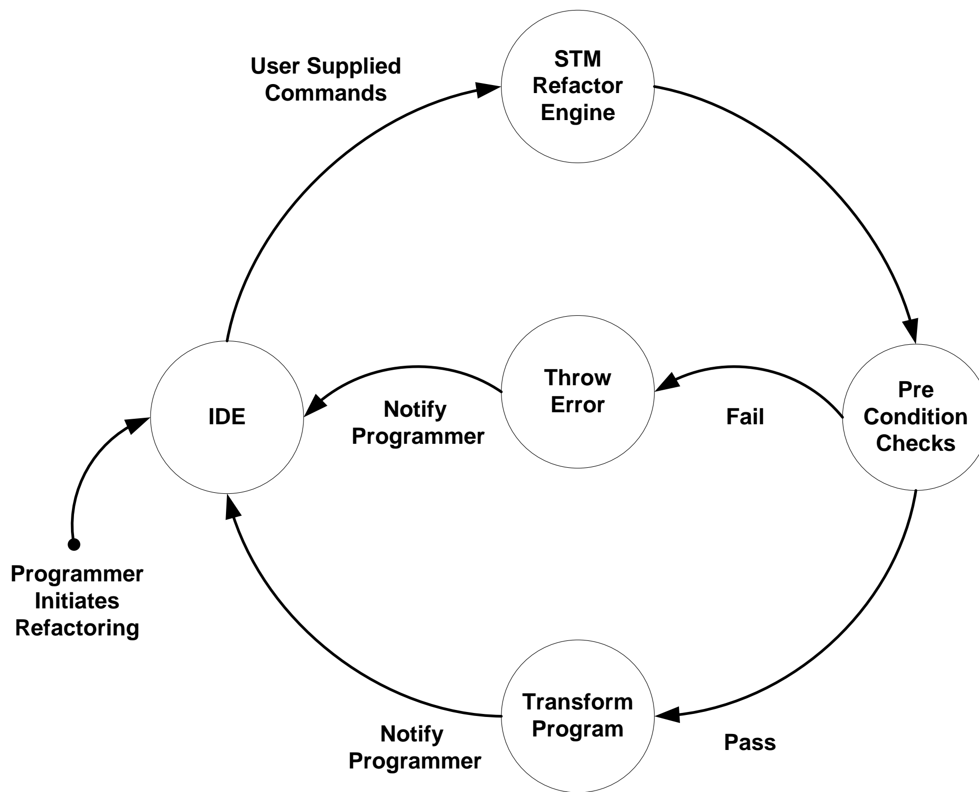
Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure [10]. This traditional definition of refactoring centers on the notion of *behavior preservation,* so that given the same inputs before and after refactoring, the outputs remain consistent. The goal of a refactoring effort is to produce better software. Incremental refactoring efforts seek to improve the internal structure of the code by improving the design through abstraction and simplification techniques. Thus, refactoring will lead to a more understandable, maintainable, and testable code base.

2.2 Design

As previously mentioned, the STM refactoring tool is an interactive tool which requires a programmer to initiate the refactoring process through the IDE as well as

approve the final changes to the program which are ultimately reflected in the IDE. In contrast, non-interactive refactoring engines may autonomously change parts of the program that satisfy the preconditions for refactoring. The Figure 2.1 diagram illustrates the interactive refactoring process.

Figure 2.1 Interactive Refactoring



2.2 .1 INTEGRATED DEVELOPMENT ENVIRONMENT

To support the addition of STM refactorings to the current suite of refactorings supported in Visual Studio [11], the IDE should have the drop down menu options to initiate the STM refactorings. A few of these refactorings and examples may include:

I. *Atomic Block Refactoring* - Atomic block allows programmers to declaratively specify which sections of the code should run in isolation. By highlighting the code in the IDE and right clicking, the STM Refactoring menu should appear giving the programmer an option to make the highlighted code *Atomic.* The result of this action encapsulates this highlighted code within an Atomic.Do()=>{…} block [13]. Figure 2.2 illustrates and example of what the programmer may be presented with when conducting interactive *Atomic Block Refactoring* in the development IDE.

Figure 2.2 Atomic Block Refactoring

```
private class objAtomic
        {
         private string m_string1 = "1";
         private string m_string2 = "2";

         public objAtomic() { }

             public bool Validate() {
             bool result;

                 result = (m_string1.Equals(m_string2) == false);
                 return result;

                }


             public void SetStrings(string s1, string s2){

                 m_string1 = s1;
                 Thread.Sleep(1);
                 m_string2 = s2;


                 }
           }
```

```
public objAtomic() { }

    public bool Validate() {
    bool result;

      Atomic.Do(() =>
        {
        result = (m_string1.Equals(m_string2) == false);
        return result;
        }

    }
    public void SetStrings(string s1, string s2){

     Atomic.Do(() =>
        {
        m_string1 = s1;
        Thread.Sleep(1);
        m_string2 = s2;
        }
    }
  }
```

II. *Create Sub Object Refactoring* - Even when appropriate using the *Atomic* blocks around data that requires isolation, of an object maybe modified within transactions while other fields of the same object are some fields modified outside of the transaction, in STM.NET this may lead to a data inconsistency. To eliminate the problem in such cases, it is necessary to break up the object into two sub-objects where one object contains the data that is accessed from within a transaction, and one object contains the data that is not accessed from a transaction. When highlighting an instance of an object in the IDE and choosing the *Create Sub Object* refactoring, a new object will be created of the same type but different name. Figure 2.3 illustrates and example of what the programmer may be presented with when conducting interactive *Create Sub Object Refactoring.*

Figure 2.3 Create Sub Object Refactoring

```
private class objAtomic
    {
     private string objName;
     private int objZipCode;
     private string objAddress;

     public int getAddress(string objName){
         return objAddress;
         }
     public void setAddress(string address){
             objAddress=address;
             }
       }
     public void setName(string name) {
             objName=name;
             }
       }
 static void Main(string[] args){
     objAtomic obj = new objAtomic();
     Thread t1 = new Thread(new ThreadStart(delegate
     {
             Atomic.Do(() =>
             {
                 obj.setAddress("One Microsoft Way");
                 Atomic.Retry();
             });
     }));
     Thread t2 = new Thread(new ThreadStart(delegate
     {
            obj.setName("Microsoft");
     }));
```

```
private class objAtomic
    {
     private string objName;

     public void setName(string name) {
             objName=name;
             }
       }

 private class SubObjAtomic:objAtomic
    {
     private string objAddress;

     public int getAddress(string objName){
         return objAddress;
         }
     public void setAddress(string address){
             objAddress=address;
             }
       }
 static void Main(string[] args){
     objAtomic obj = new objAtomic();
     Thread t1 = new Thread(new ThreadStart(delegate
     {
             Atomic.Do(() =>
             {
                 obj.setAddress("One Microsoft Way");
                 Atomic.Retry();
             });
     }));
     Thread t2 = new Thread(new ThreadStart(delegate
     {
            obj.setName("Microsoft");
     }));
```

III. *Add Atomic Retry Refactoring* - STM.NET implements the “retry” coordination mechanism that can be added inside of transaction blocks to specify that in the event of failure the transaction rolls back and re-executes. By placing the cursor on a line in the IDE, the *Add Atomic Retry* refactoring will result in the placement of an *Atomic.Retry()* method at the selected line. Figure 2.4 illustrates and example of what the programmer may be presented with when conducting interactive *Add Atomic Retry Refactoring* after successfully completing *Atomic Block Refactoring* in the development IDE.

Figure 2.4 Add Atomic Retry Refactoring

```
public objAtomic() { }

    public bool Validate() {
    bool result;

      Atomic.Do(() =>
        {
        result = (m_string1.Equals(m_string2) == false);
        return result;
       }

    }

    public void SetStrings(string s1, string s2){

     Atomic.Do(() =>
        {
        m_string1 = s1;
        Thread.Sleep(1);
        m_string2 = s2;
       }
    }
  }
```

```
private class objAtomic
        {
         private string m_string1 = "1";
         private string m_string2 = "2";

         public objAtomic() { }

             public bool Validate() {
             bool result;

               Atomic.Do(() =>
                 {
                 result = (m_string1.Equals(m_string2) == false);
                 return result;
                 Atomic.Retry();
                }

             }

             public void SetStrings(string s1, string s2){

              Atomic.Do(() =>
                 {
                 m_string1 = s1;
                 Thread.Sleep(1);
                 m_string2 = s2;
                 Atomic.Retry();
                }
             }
           }
```

2.2 .2 STM REFACTORING ENGINE

The STM refactoring engine performs lexical analysis and parsing of the program source to a form of representation using an abstract syntax tree (AST). An AST is created from the original program source representing the as-is state, and the second AST is created when a refactoring action is initiated representing a to-be state of the program. The second AST is created from the following user supplied commands:

I. *MakeAtomic()* adds an *Atomic.Do()=>* {…} block with the program code. This is legal everywhere in the program code however only statements that change mutable memory state are allowed to be within the atomic block. Input/Output (IO) operations to disk or console are not allowed.

II. *CreateSubObject(class,objName)* adds a new, unreferenced object to the specified class. This is legal if there are no other existing objects of the same name.

III. *AddAtomicRetry()* adds an instance of the *Atomic.Retry()* method to the program. This is legal as long as it occurs as a statement within an atomic block.

2.2 .3 PRECONDITION CHECKS

A refactoring is essentially composed of a precondition and a program transformation so therefore be characterized as $R = (Pre; T)$ [18], where the refactoring R is valid if and only if the precondition for transformation T is satisfied. When the programmer applies a STM refactoring to the program, the *to-be* AST is analyzed to validate the preconditions of the refactoring. As previously mentioned, the key property of refactorings is the notion of behavior preservation. For any refactoring to be valid it must satisfy the following: a refactoring with precondition represented by predicate *Pre*, transformation T for program p then $\lambda p,$ if $(Pre\ p)$ then $(T\ \mathrm{p})$, else p. Letting $\equiv$ denote the behavioral equivalence between programs the refactoring must satisfy $\forall p.\ (Prep) \longrightarrow (T\ p) \equiv p$.

2.2.3.1 Transform Program

If all the preconditions are satisfied for the chosen refactoring, the internal representation of the program (the AST) is transformed according to the refactoring. The *to-be* AST is then presented to the programmer in program source form with any

invariants and comments of the original program preserved. The programmer at this point may accept the refactoring or undo the refactoring.

2.2.3.2 Throw Error

If the preconditions are not satisfied, the refactoring process stops and the original program will remain unchanged, and a notification is provided in the IDE to the programmer.

3. Related Work

Despite the significant research work in the area of refactoring, only a small fraction of that work is in the domain of refactoring concurrent programs. In research conducted by Danny Dig [14] [15], refactoring is proposed as a means to gain performance, scalability, and throughput in sequentially executing programs. Tool support for refactorings may fall into the one of two broad categories: fully automated or interactive. In fully automatic refactorings no programmer action is required and the compiler automatically parallelizes the code; in interactive refactoring, the programmer initializes the refactoring in the code and can view or undo changes as necessary. Both of these methods have their benefits and drawbacks, but in most cases it is preferable to allow the programmer, who is also the domain expert, to initiate these refactorings with IDE support. In this way, programmers may ensure the appropriate methods or functions are parallelized while preserving any program invariants. IDEs like Visual Studio may then be used to perform the tedious automated work such as maintain code consistency and modifying references across the entire project/solution.

In this model of programmer initiated automated refactoring there are several points of interaction with the programmer. The programmer selects some code as the refactoring target, and the tool analyzes the safety of the transformation. It is then the programmer's responsibility to identify all shared data or compute-intensive code and target it with the appropriate refactorings. If some of the refactoring preconditions are not met, the tool raises warnings of the code in question. The programmer may decide to cancel the refactoring, fix the code and then re-run the refactoring, or roll back any changes. By default, the refactoring tool applies the changes only when its analysis

determines that it is safe to do so. However, the programmer has the choice to ignore the warnings and apply the changes anyway.

Concurrencer [15] is a tool that supports this vision. *Concurrencer* provides support to refactor sequential code to concurrent code using the java.util.concurrent (j.u.c.) library available in Java version 5. Dig's proof of concept tool supports three refactorings: Convert INT to ATOMICINTEGER, Convert HASHMAP to CONCURRENTHASHMAP, and Convert RECURSION to FJTASK (Fork Join Task). Each instance of these refactorings would start by selecting the field and invoking the relevant refactoring. *Concurrencer* performs find and replace action on all field updates with their corresponding concurrent API methods, e.g. INT to ATOMIC INTEGER. *Concurrencer* will convert a sequential program into one which is thread-safe by applying thread-safe functions in place of functions that would require the use of locks to maintain safety and warn the programmer if the update expression cannot be made thread-safe, other than by using a lock. Figure 2.5 is an example of the ConvertToConcurrentHashMap refactoring [15].

Figure 3.1 ConvertToConcurrentHashMapRefactoring

```
private Map<Locale, String[]> timeZoneLists;
private String[] timeZoneIds;

public String[] getTimeZoneList() {
        Locale jiveLocale = JiveGlobals.getLocale();
        String[] timeZoneList = timeZoneLists.get(jiveLocale);

        if (timeZoneList == null) {
                timeZoneList = new String[timeZoneIds.length];

        for (int i = 0; i < timeZoneList.length; i++) {

                }

        timeZoneLists.put(jiveLocale, timeZoneList);
        }
        return timeZoneList;

}
```

```
private ConcurrentHashMap<Locale, String[]> timeZoneLists;

      private String[] timeZoneIds;
      public String[] getTimeZoneList() {
      Locale jiveLocale = JiveGlobals.getLocale();

      String[] timeZoneList = timeZoneLists.get(jiveLocale);
      String[] createdTimeZoneList = createTimeZoneList(jiveLocale);

      if (timeZoneLists.putIfAbsent(jiveLocale, createdTimeZoneList) == null{
              timeZoneList = createdTimeZoneList;
              }

      return timeZoneList;
      }

      private String[] createTimeZoneList(Locale jiveLocale) {
      String[] timeZoneList;
      timeZoneList = new String[timeZoneIds.length];

      for (int i = 0; i < timeZoneList.length; i++) {

      }

      return timeZoneList;
}
```

In related work by Max Schäfer et al [19], he presents a tool to convert Java Synchronize blocks (locks) to ReentrantLocks [20] & ReadWriteLocks [21]. The *Relocker* tool presented is an automated tool for changing Java source code by using a set of conversion algorithms. There were some implementation concerns cited in this work, but in most benchmarks, the conversion was effective. What is undetermined from these benchmarks is whether the increased presence of ReadWriteLocks does actually improve concurrency because the test results centered on validating the actual conversion algorithms, and not the net impact of the conversion on concurrency. Figure 2.6 gives a pseudo code description of the main conversion algorithm for the Convert to Reentrant Lock refactoring. Given an abstract monitor M to refactor, it creates a corresponding lock field using procedure createLockField, and then iterates over all monitor actions (a) in the program [19].

Figure 3.2 Convert to Reentrant Lock refactoring

```
1: procedure CONVERT TO REENTRANT LOCK (AbstractMonitor M):
2: createLockField(M)
3: for all monitor actions a do
4: if M(a) ↕ M then
5: transformAction(a)
6: else
7: assert M(a) ? M
8: procedure createLockField (AbstractMonitor M):
9: if M is FM(f) then
10: assert f is declared in a modifiable type
11: create lock field l as sibling field of f
12: for all assignments a to f do
13: insert assignment l = new ReentrantLock() after a
14: else if M is CM(C) then
15: assert C is a modifiable type
16: create static lock field l in C
17: else /* M must be of the form TM(C) */
18: assert C is a modifiable class
19: create lock field l in C
20: procedure transformAction (MonitorAction a):
21: assert a is from source code
22: l  mkLockAccess(a)
23: if a is synchronized(e) { ... } then
24: replace a with l.lock(); try { ... } finally { l.unlock(); }
25: else if a is a synchronized method then
26: remove synchronized modifier from a
27: replace body of a with l.lock(); try { ... } finally { l.unlock(); }
28: else /* a must be call to wait or notify */
29: . . .
30: function mkLockAccess (MonitorAction a):
31: e   me(a)
32: if M(a) is FM(f) then /* e is of the form e0:f */
33: return e0:l
34: else if M(a) is CM(C) then
35: return C:l
36: else /* M(a) must be of the form TM(C) */
37: return e:l
```

Similarly, *Relooper* [22], is an automated tool, and integrates with the Eclipse IDE to replace traditional for array data structures with a ParallelArray. The goal of this is to aid in retrofitting sequential code loop/array structures into a thread-safe data structure to improve a program's parallelism. Figure 2.7 [22] illustrates how to convert an

array of Complex numbers to a ParallelArray with the original code on the top and the refactored code on the bottom.

Figure 3.3 Complex numbers to a ParallelArray Refactoring

```
public class TestComplex{

        private Complex[] numbers;

        public void test(){
                numbers= new Complex[10000000];
                for (int i=0;i<numbers.length;i++){
                        numbers[i] = Complex.createRandom();
                }

                for (int i=0;i<numbers.length;i++){
                        numbers[i].makeSquare();
                }

                Complex sum = new Complex(0,0);
                for (int i=0;i<numbers.length;i++){
                        sum = new Complex(sum.a+numbers[i].a,sum.b+numbers[i].b);
                }

        }
}
```

```
public class TestComplex{

        private ParallelArray<Complex> numbers;

        public void test(){
                numbers= ParallelArray.create(10000000, Complex.class,ParallelArray.defaultExecuter());

                numbers.replaceWithGeneratedValue(new Ops.Generator<Complex>(){
                        public Complex op(){
                                Complex elt;
                                elt = Complex.createRandom();
                        }
                });

                numbers.apply(new Ops.Procedure<Complex>() {
                        public void op(Complex elt) {
                                elt.makeSquare();
                        }
                });

                Complex sum = new Complex(0,0);
                sum = numbers.reduce(new Ops.Reducer<Complex>() {
                        public Complex op(Complex sum, Complex elt) {
                                sum = new Complex(sum.a + elt.a, sum.b + elt.b);
                                return sum;
                        }
                }, sum);
        }
}
```

Table 3.1: Related concurrency refactoring tools and their respective usage domain.

Tools	Purpose	Domain
Concurrencer	Data structure refactoring	Java
Relocker	Lock refactoring	Java
Relooper	Loop/Array refactoring	Java

4. Conclusion

Automated tool support will be required to facilitate the transition to implementing Software Transactional Memory language constructs in existing software systems. This paper presented a basic design of a tool that leverages the idea of interactive refactoring browsers to enable programmers the opportunity to make incremental changes to their programs. All proposed STM refactorings get validated by a set of preconditions for a chosen refactoring that when met will defer the final decision to the programmer to go through with the refactoring or not. The presented design could be the stepping stone in developing an appropriate implementation plan for a fully operational tool; such a tool would yield a significant value and may help guide and educate programmers on how to use STM language features in their programs in a way that is succinct and promotes healthy code design.

5. Appendix

5.1 MEASURING CONCURRENCY

Concurrent programming models attempt address the complex challenges of having two or more activities executing within the same interval of time by providing a set of tools in the form of algorithms and patterns to help manage these concurrently executing activities. Integrating a mature concurrent programming model into the development process of a new application surely will help application architects realize their scalability goals across multi-core platforms. To realize equivalent gains in sequential legacy code systems, sequential code dependencies need to be identified and refactored accordingly. This task is not only complex but it presupposes that the legacy code can indeed be refactored for concurrent thread executions. Before taking on this challenge, it is not only important to evaluate whether any of the time and effort spent in analysis and refactoring can in fact yield a positive result. The aim of the research conducted in the area of measuring program parallelism is to answer two questions:

1) "How much of my program can run in parallel?"
2) "Has refactoring improved my program's parallelism"

5.1.1 Amdahl's Law

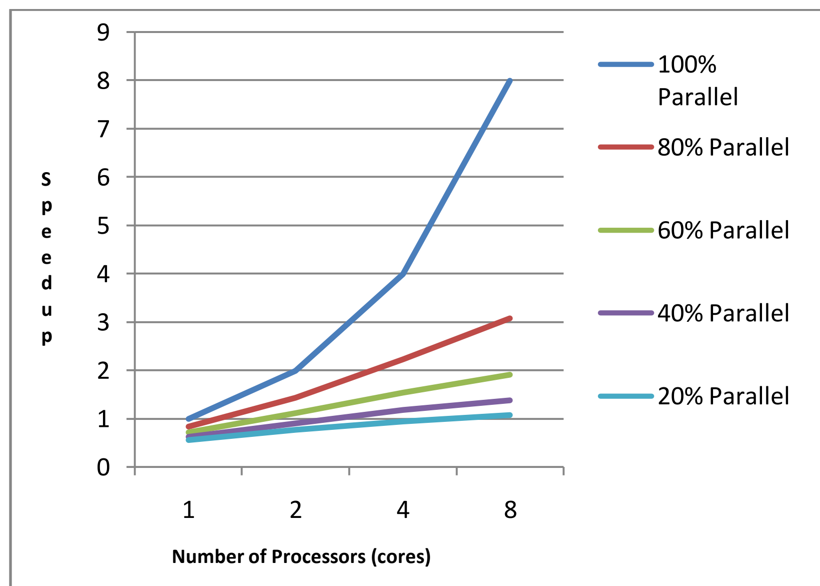
Amdahl's Law [23][24][25][26][27] is one of the most cited sources for measuring the scalability of sequential programs by adding additional processors. Amdahl stated that the increase in execution time of a given program due to an increase

in parallelism is expressed by *Speedup* (*P*). Where N is the number of processors (processor cores), f is the fraction of the code that is parallelizable while the remaining fraction $(1 - f)$ is permanently sequential. This equation is written as:

$$Speedup\ (P) = \frac{1}{(1-f) + \frac{f}{N}}$$

The following graph depicts the relationship between number of processor cores and parallel code when applying Amdahl's formula.

Figure 5.1 Amdahl's Law, speedup as a function of CPU cores



When analyzing the output from Amdahl's calculation, specifically the 100% parallel case, the speedup factor increases linearly with the addition of processor cores and is unbounded. Empirically, we know that this does not happen due to the overhead of thread creation, communication, and scheduling. John Gustafson noted a less obvious problem with Amdahl's Law in 1988 [25] and proposed an alternative he referred to as a *Scaled Speedup* in contrast to Amdahl's *Fixed Speedup*. Gustafson noticed that some workloads scaled more efficiently than others and that the flaw in Amdahl's formula was that it assumed a fixed, homogenous computational workload. His proposed modification to Amdahl's original equation states that the amount of serial (s) and parallel (p) time spent on a parallel system, would require s + p * N time to perform the task using a serial

processor. The derivation of this formula is given as:

$$Scaled\ Speedup\ (P) = \frac{s + p * N}{(s + p)}$$

$$= s + p * N$$

$$= N + (1 - N)x\ s$$

An interesting re-evaluation of Amdahl's and Gustafson's assumptions was conducted by Yuan Shi [27] in 1996, where he claimed that Amdahl's and Gustafson's work was in fact the same law with two different formulations. In the paper, Yuan Shi shows the mathematical equivalence between the two formulations and claims that although the two laws are theoretically correct; deriving the serial percentage of parallelism is not practically obtainable. Much of the analysis of Amdahl's Law has served to point out its limitations without necessarily providing viable alternatives.

5.1.2 Related Work

In his early work, [28] D.J. Kuck proposed performing static analysis on program traces. The program execution was analyzed statically to determine which statements could be executed in parallel. The average total parallelism of the program was derived from the average measured parallelism of all of traces analyzed. Static analysis methods are still approximations because of the potentially large number of program states that cannot be observed statically. Furthermore, it is noted that this analysis tends to give a conservative estimate for available parallelism because the flow of a value between two

operations has to be assumed whenever the absence of this flow cannot be proved analytically, as pointed out by Manoj Kumar [29].

Noting the shortcomings of static program analysis, Manoj Kumar developed a tool to measure parallelism dynamically, i.e. at runtime. COMET, *Concurrency Measurement Tool*, worked by monitoring the statements executed at runtime and the flow of values between them, deducing the maximum concurrency possible [29]. In execution, COMET creates a program extension *P'* and modifies *P*' while preserving the behavior of the original program *P* during runtime. For each variable in *P*, a shadow variable in *P'* is created. Shadow variables are tuples that consist of a representation of the variable in *P*, and a timestamp of when it was created. As *P* executes its core statements, tracking statements are created in *P'* that trace the dynamic execution sequence of the core statements in order to compute the earliest time at which this core statement would execute. In *P'*, control variables are also created. Control variables of a statement *S* in *P* mark the time that *S*, in *P* should execute. By capturing the points in program time from which the statements in *P* can execute, the earliest time when dependent successor statements can execute is known. From this, COMET deduces sequential statement dependencies, thus measuring a programs degree of parallelism.

Inspired by Kumar's work, Rountev et al [30] used a similar model for measuring parallelism. Dynamic program analysis is used to compute a timestamp for each executed statement in order to determine the ratio between the total number of statements and the largest timestamp computed. This derives as *N* being the number of executed program statements, and *T* being the best case parallel time for the program, and then the ideal increase in time gained through parallel execution is characterized as $\frac{N}{T}$. It is important to note that this measure is what Rountev refers to as a program's *available parallelism* and

not the actual speedup that may be achievable through parallelism. A significant difference between Rountev's and Kumar's measurements are that Rountev focuses on method level parallelism while Kumar's COMET tool was designed for instruction level analysis of scalar values and arrays.

5.1.3 Conclusion

The aforementioned research attempts to provide an objective measure of potential parallelism through both theoretical computation and program analysis (static and dynamic). Across the research there is an apparent tradeoff between simplicity and potential accuracy when attempting to derive a measurement. Amdahl's Law being the simplest form of measurement relies on a subjective measurement, presumably through static code analysis, of the amount of code that can be parallelized in the program. Due to this subjectivity, it is appropriate to assume that the measure would differ substantially while under examination by different people. The static analysis methods proposed by D.J. Kuck, mitigates the subjectivity of analysis by averaging measurements across all program traces analyzed. This method has the potential to be accurate in situations where the number of possible program states is very few but this is seldom the case in reality. Assuming the number of program states may be extremely large, a greater number of program traces would need to be averaged to maintain accuracy and consistency, which is not practical.

With the methods proposed by Kumar and Rountev, manual efforts are replaced by the dynamic analysis conducted my software tools at runtime. Dynamic analysis

methods have promise because they are able to capture and analyze large number of program states and gain better insights on how the program executes.

Both static and dynamic analyses have the potential to help answer the two questions posed earlier in this review:

1) “How much of my program can run in parallel?”
2) “Has refactoring improved my program’s parallelism”

Tool supported static analysis may be better applied to answer question 1, while dynamic analysis methods may be applied to question 1 and would be better suited to answer question 2 because this method would actually observe the program behavior during runtime, as opposed to predicting its behavior as a static analysis method would.

The analysis of program parallelism does not take into account system overhead such as thread creation, scheduling, and coordination. This separation in the analysis of the application code from the application system is undoubtedly required considering the number of variants it would inject into problem space, making any measurement difficult at best to attain consistency across platforms. However, in a practical sense, system overhead can be a huge limitation to application scalability. Due to this, an approach to measure and assess an application’s total scalability may be decomposing the system parts and applying measurement analysis to each part. The system part with the lowest degree of parallelism represents the maximum parallelism achievable.

References

[1] “Cramming more components onto integrated circuits”, Gordon E. Moore 1965

[2] "The Deadlock problem: a classifying bibliography", Zobel, 1983

[3] http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx

[4] “Process Structuring, Synchronization, and Recovery using Atomic Actions”, D.B. Lomet, 1977

[5] “Specification and implementation of resilient, atomic data types”, Weihl and Liskov, 1983

[6] “Implement Atomic Actions on Decentralized Data”, David Reed, 1983

[7] “Transactional Memory: Architectural Support for Lock-Free Data Structures”, Maurice Herlihy et al

[8] http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/

[9] http://labs.oracle.com/projects/scalable/scalable.html

[10] “Refactoring: Improving the Design of Existing Code”, Martin Fowler et al, 1999

[11] http://msdn.microsoft.com/en-us/vstudio/ff718165.aspx

[12] http://blogs.msdn.com/b/stmteam/

[13] “STM Programmers guide”, Microsoft July 24, 2009

[14] “A Refactoring Approach to Parallelism”, Danny Dig

[15] “Refactoring Sequential Java Code for Concurrency via Concurrent Libraries”, Danny Dig et al

[16] “Reengineering for Parallelism: An Entry Point into PLPP (Pattern Language for Parallel Programming) for Legacy Applications, Berna L. Massingill et al

[17] “Correct Refactoring of Concurrent Java Code”, Max Schäfer et al

[18] “Practical Analysis for Refactoring” Donald Bradley Roberts, 1999

[19] “Refactoring Java Programs for Flexible Locking”, Max Schäfer et al, 2010

[20] http://download.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/ReentrantLock.html

[21] http://download.oracle.com/javase/1,5.0/docs/api/java/util/concurrent/locks/ReadWriteLock.html

[22] “ReLooper: Refactoring for Loop Parallelism in Java”, Danny Dig et al, 2010

[23] “Validity of the single processor approach to achieving large scale computing capabilities” Gene M Amdahl 1967

[24] “Amdahl’s Law in the Multicore Era”, Mark D. Hill et al

[25] “Multicore Programming”, Akhter& Roberts, Intel Press 2006

[26] “Reevaluating Amdahl’s Law”, John L. Gustafson 1988

[27] “Reevaluating Amdahl’s Law and Gustafson’s Law”, Yuan Shi 1996

[28] “Measurements of parallelism in ordinary FORTRAN programs”,D. J. Kuck et al, 1974

[29]”MeasuringParallelism in Computation-Intensive Scientific/Engineering Applications” Manoj Kumar, IEEE 1988

[30] “Understanding Parallelism-Inhibiting Dependences in Sequential Java Programs”,AtanasRountev et Al