Daniel Luis Landeiroto Parreira

Licenciado em Engenharia Informática

Data-Centric Concurrency Control on the Java Programming Language

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática

Orientador: Prof. Doutor Hervé Miguel Cordeiro Paulino

Júri:

Presidente: Profª. Doutora Margarida Paula Neves Mamede

Arguentes: Prof. Doutor Francisco Cipriano da Cunha Martins

Vogais: Prof. Doutor Hervé Miguel Cordeiro Paulino

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2013

Data-Centric Concurrency Control on the Java Programming Language

To all that supported me.

Whoever fights monsters should see to it that in the process he does not become a monster. And if you gaze long enough into an abyss, the abyss will gaze back into you.
- Friedrich Nietzsche

Acknowledgements

In first place, I would like to thank my coordinator, Prof. Hervé Paulino, for all the unconditional support given during the elaboration of this thesis. Without his guidance, this work would not be possible. I would also like to thank the Department of Informatics of the Faculty of Sciences and Technology of the New University of Lisbon for all the support, not only by the installations provided, but also from the staff.

A special thanks towards all my family for supporting me throughout all these years.

Additionally, a special thanks to Prof. Nuno Preguiça and the RepComp project (PTDC/EIA-EIA/108963/2008) for allowing me to access the machine used to run the benchmarks that were used to evaluate the performance of the work developed in this thesis.

Abstract

The multi-core paradigm has propelled shared-memory concurrent programming to an important role in software development. Its use is however limited by the constructs that provide a layer of abstraction for synchronizing access to shared resources. Reasoning with these constructs is not trivial due to their concurrent nature. Data-races and deadlocks occur in concurrent programs, encumbering the programmer and further reducing his productivity.

Even though the constructs should be as unobtrusive and intuitive as possible, performance must also be kept high compared to legacy lock-based mechanism. Failure to guarantee similar performance will hinder a system from adoption.

Recent research attempts to address these issues. However, the current state of the art in concurrency control mechanisms is mostly code-centric and not intuitive. Its code-centric nature requires the specification of the zones in the code that require synchronization, contributing to the decentralization of concurrency bugs and error-proneness of the programmer. On the other hand, the only data-centric approach, *AJ* [VTD06], exposes excessive detail to the programmer and fails to provide complete deadlock-freedom.

Given this state of the art, our proposal intends to provide the programmer a set of unobtrusive data-centric constructs. These will guarantee desirable security properties: composability, atomicity, and deadlock-freedom in all scenarios. For that purpose, a lower level mechanism (*ResourceGroups*) will be used. The model proposed resides on the known concept of atomic variables, the basis for our concurrency control mechanism.

To infer the efficiency of our work, it is compared to Java synchronized blocks, transactional memory and *AJ*, where our system demonstrates a competitive performance and an equivalent level of expressivity.

Keywords: Data-centric, Concurrency control, Deadlock-freedom, Atomicity

Resumo

O paradigma multi-core impulsionou a programação concorrente em memória partilhada para um lugar de destaque no desenvolvimento de software. O seu uso é limitado pela natureza e poder expressivo das construções de alto nível para sincronizar acesso a recursos partilhados. Raciocinar sobre estas construções não é trivial devido à sua natureza concorrente. Podem ocorrer *data-races* e *deadlocks*, reduzindo a produtividade.

Embora as construções devam ser intuitivas, o seu desempenho também deve ser semelhante a mecanismos baseado em *locks*. Neste contexto, a preservação de um desempenho equivalente é fundamental para a adopção generalizada de um dado sistema.

Estas questões têm sido alvo de bastante investigação nos anos recentes. No entanto, o actual estado da arte em controlo de concorrência é centrado no código e nem sempre intuitivo. A sua natureza centrada no código requer que se especifique onde se requer sincronização, contribuindo para a descentralização de erros de concorrência. Por outro lado, a única abordagem centrada nos dados , *AJ* [VTD06], expõe demasiados detalhes ao programador e não consegue assegurar a ausência de *deadlocks* em todos os cenários.

Dado este estado da arte, a nossa proposta pretende fornecer ao programador um conjunto de construções centradas nos dados para controlo de concorrência, que deverão garantir atomicidade, composicionalidade e ausência de *deadlocks* em todos os cenários. Para esse propósito, será usado um sistema de mais baixo nível (*ResourceGroups*). O modelo proposto reside no conhecido conceito de variável atómica, a base do nosso mecanismo de gestão de concorrência.

De modo a aferir a eficiência da nossa solução, esta é comparada com os blocos sincronizados do Java, memória transacional e o *AJ*, onde o nosso sistema demonstra uma performance competitiva e uma expressividade equivalente.

Palavras-chave: *Data-centric*, Controlo de concorrência, Livre de *deadlocks*, Atomicidade

Contents

List of Figures

List of Tables

Listings

1

Introduction

1.1 Motivation

Ever since the stalling of single core frequency scaling, largely due to physical limits of the materials [EBA+11], processor development has shifted to multi-core oriented architectures. In order to tap into the full potential of the new architectures, a change in mentalities was required. Algorithm efficiency is no longer the single most relevant aspect when measuring program performance: the ability to distribute the computation across multiple cores is gaining importance.

In the near future, consumer computers running 16 or 32 cores will be commonplace. In the meantime, distributed algorithms and concurrent programming will become increasingly more important. However, re-engineering an algorithm to run concurrently might not be viable, and certainly not a trivial task. Although using libraries that abstract hardware threading, such as POSIX threads, programmers are still encumbered by the burdens brought by concurrent programming.

Only recently have mainstream applications began actively supporting concurrency. Consider the *libjpeg* and *libjpeg-turbo* libraries. Even though they are almost omnipresent in modern computer systems, multi-threaded decoding is still not officially supported [Jin12].

Concurrent programming in a shared-memory environment brings the need for synchronization across multiple threads [Dij65], as access to shared data in a multi-threaded environment has to be sanitized. The non-deterministic nature of those accesses greatly contributes to the difficulty in debugging and proving the correctness of concurrent programs.

The arbitrary ordering of operations can easily result in bugs. They have origin in the unexpected interleaving of accesses to shared memory that was not foreseen by the programmer. Those events are commonly known as *data-races*.

Data-races are in the genesis of many errors in shared-memory programming. They occur when multiple threads are allowed to concurrently access shared data and one of them performs a write operation. These scenarios are not explicit in the code, and require special attention from the programmer when shared memory communication is in place. Since shared data, like global variables and shared references, can be accessed in various parts of the code, the programmer has to be aware of such access patterns or else data-races could occur.

A commonly used approach to prevent data-races is to force a sequence of statements to be evaluated in mutual exclusion regarding the other threads in the system, ensuring atomicity. However, mutual exclusion brings another problem: deadlocks. In a multiprogramming environment, several processes may compete for a finite number of resources. If the order in which they attempt to acquire them falls under a scenario where further progression is impossible, a deadlock has occurred.

Accordingly, the need for simple and effective constructs that avoid those pitfalls and allow the programmer to control accesses to shared memory is currently a pressing research topic.

1.2 Problem

Providing useful concurrency control mechanisms is challenging. The high level constructs should be intuitive to use by the programmer and guarantee a certain degree of scalability and security properties, such as atomicity and deadlock-freedom. However, the challenges themselves conflict with each other. Maximizing performance and concurrency might require additional information that is naturally conveyed through more annotations, resulting in less intuitive constructs.

Thus, these challenges cannot be addressed individually. New concurrency control mechanisms have to adopt a phase of experimentation and optimization for finding the best balance of trade-offs that the system can offer to the programmer.

At the present, mutex locks are still the *de facto* concurrency control mechanism. They allow the programmer to enforce mutual exclusion in the evaluation of a sequence of program statements. This mechanism further adds to the complexity of the code, without being an optimal solution. One of the problems associated with locks is *deadlock-proneness* and the decentralization of concurrency errors, particularly as the number of locks increases. Since locks have to be applied in all sections of code where the programmer wishes to protect a given computation from concurrent accesses, the application of concurrency control primitives is decentralized, leading to decentralization of the location of concurrency errors. Additionally, deadlocks that can arise in while using locks are tough to pinpoint and their usage entails a higher degree of error-proneness compared to unsynchronized programs. As such, research in the area for constructs that ensure deadlock-freedom is ongoing.

1.2.1 Deadlocks

Formally, when multiple threads T compete over the acquisition of a finite set of resources R, eventually an attempt to acquire a resource R_p will fail because it is already held by another thread $T_{i'}$. As a result, thread T_i blocks while waiting for resource R_p to become available. In the meanwhile, thread $T_{i'}$ might attempt to acquire resource R_q that is already held by thread T_i. This prompts thread $T_{i'}$ to block while waiting for the resource to become available, completing the circular-wait. This situation, illustrated in Figure 1.1, embodies a deadlock. A popular example of a deadlock situation was E. W. Dijkstra's *Dinning Philosophers* problem [Dij65].

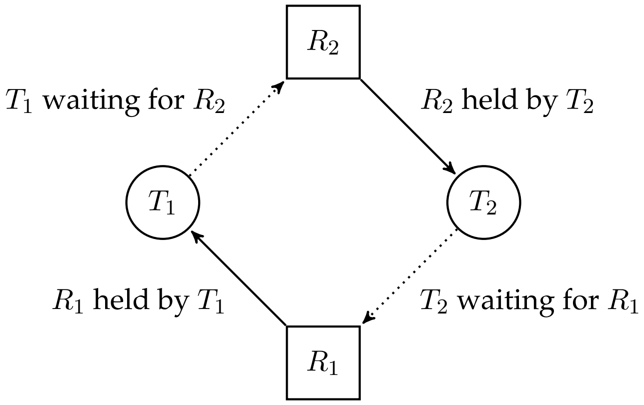


Figure 1.1: A circular-wait involving two threads holding one resource each.

The requirements for a deadlock to occur have long been identified in the literature [SGG05]. They are characterized by four conditions which have to be simultaneously verified:

mutual exclusion only one process can hold a resource at any given time

hold and wait a process holding at least one resource is waiting to acquire additional resources

no preemption only the process holding a resource can release it

circular wait a set of waiting processes $P_0, ..., P_n$ must exist such that each is waiting for a resource held by another process, e.g. P_0 waits for the resource held by P_1, ..., P_n waits for the resource held by P_0. Formally we have :

$$\forall_i \nexists_j : i, j \in \{0, ..., n\} \wedge i \neq j \wedge P_i \text{ waits on } P_j$$

To illustrate real scenarios in which deadlocks might occur, a banking application will be used as an example. Besides being an intuitive example to which the reader relates with, it provides an interesting case for discussing concurrency related concerns, being

```
void transfer(Account A, Account B, int amount)
{
    A.lock();
    B.lock();
    A.take(amount);
    B.give(amount);
    B.unlock();
    A.unlock();
}

...

transfer(accA, accB, 10); //executed by thread #1
...
transfer(accB, accA, 10); //executed by thread #2
```

Listing 1.1: Scenario exemplifying a deadlock during a transfer using mutex locks.

widely used as an example in the literature [Jon07, ST95]. This interest steams from the consistency and atomicity requirements when dealing with transactions between bank accounts. The system can never be in a state, externally observed, in which the money is already retrieved from one account but not yet deposited in another account.

To avoid an inconsistent external state, transfers have to be atomic. For this, the programmer could lock both accounts, ensuring mutual exclusion for the transfer, and releasing the locks at the end so that other transfers can take place, as shown in Listing 1.1.

However, even though atomicity is guaranteed, a deadlock could occur. The deadlock arises when the first thread only attains the `accA` lock and the second thread only attains the `accB` lock. Neither will be able to attain the next lock needed to process the transfer because the required lock is held by the other thread. This completes the circular-wait, entering a deadlock.

However, if the locks were attained in the same order, no deadlock could occur. When two or more resources are involved, locking them according to a global order is an effective way to prevent deadlocks. This approach eliminates deadlocks in cases where cyclic dependencies are present.

While applying locks to a small segment of code can appear trivial at first sight, the data that is guarded by those same locks may, naturally, be accessed by other threads across the program. This burdens the programmer with concerns that demand an evaluation of the impact of those accesses in the correctness of the code segment enclosed by the lock synchronization primitives. Improper evaluation of such accesses (and subsequent synchronization, if required) can lead to data-races. Such code-centric approach requires a non-local reasoning and a vast knowledge of the code by the programmer [VTD06].

On top of that, callers need to be aware of the concurrency management requirements of their callees, namely which locks they are taking. This implies knowing their implementation to avoid deadlock scenarios. As such, systems based on lock mechanisms are highly uncomposable and hard to maintain.

As expound, the extensive use of locks burdens the programmer with many decisions. The number of locks to use, where to place them, identifying potential deadlock-prone situations and how to deal with error-recovery inside regions delimited by locks. All these can affect the correctness and efficiency of the program.

1.2.2 Types of scenarios

The effectiveness of a concurrency control mechanism might depend on the type of scenario it is presented with: static or dynamic. Most notably, this theme will be referenced in the state of the art (Chapter 2) due to deadlock-freedom provided by the various systems varying according to the type of scenario considered. Due to the way they are implemented, some can only provide deadlock-freedom in static scenarios. In dynamic scenarios, providing for this property becomes inherently problematic.

Static scenarios represent the majority of scenarios in applications due to its fixed nature, as previously shown in Listing 1.1. They represent the scenarios where all the resources to be acquired before evaluating a critical section are possible to infer at compile time or at runtime, before entering the critical section.

Trivially, if the locks in Listing 1.1 were replaced by a higher level construct, the compiler could recognize the objects involved in the critical section, which in this scenario are account `accA` and `accB`, and acquire locks on both objects to ensure atomicity. In order to assure deadlock-freedom in this scenario, the locks would have to be acquired according to a previously established global order.

On the other end of the spectrum, in **dynamic scenarios**, no information can be attained at compile time about all the objects that take part in the critical section. A dynamic scenario in the removal of a node from a doubly-linked list is shown in Listing 1.2. Since the locking of more than one memory position is not atomic, a *guard* is required to prevent other threads from traversing the list at the same time. As a result, a deadlock-prone situation arises if in another part of the program a node is locked before the guard is invoked. The global nature of dynamic scenarios hinders its reasoning.

Some approaches, such as the one presented in [MZGB06], adopt a conservative approach and refuse to compile if deadlock-freedom cannot be guaranteed. In some cases, tools actually manage to support dynamic scenarios like *resource groups* (the system that will be the backbone of this thesis, as detailed in Section 2.2.2) and [CCG08, HFP06, EFJM07, WLK^{+}09] mentioned in the state of the art.

By contrast, transactional memory attempts to solve this at runtime by rolling back the execution of a thread whenever the memory operations it performs conflict with changes made by other threads or when a deadlock is detected (in pessimist approaches).

To take a conservative approach, the compiler can attempt to figure out a coarser alternative that guarantees the mutual exclusion for the possible objects in the critical section. One such approach is to force critical sections in dynamic scenarios to be serialized

```
void remove(List L, Node node) {
  atomic{
    for(Node cur in L) {
      if(cur==node) {
        Node prev = cur.prev ;
        Node next = cur.next ;
        prev.next = next ;
        next.prev = prev ;
      }
    }
  }
}
```

Listing 1.2: A dynamic scenario during the removal of a node from a doubly linked list

according to a global lock.

Another possible approach is acquiring a lock according to the type of the data that can be part of the dynamic set of unknown variables. For example, in a list constituted of nodes, a list/node global lock could be taken. That way, even though the actual data used in a dynamic scenario is unknown before entering the critical section, the type of the data passive of acquisition is known, hence the lesser known common denominator used to retain the atomicity.

However, these approaches intending to cope with dynamic scenarios result in a loss of concurrency. Compared to them, the lower level system *resource groups* achieves a minimal loss of concurrency.

1.2.3 Two approaches to synchronization: code-centric and data-centric

So far, this document has focused on **code-centric** synchronization. They are the most widely used constructs, which explicitly delimitate the sequence of instructions that should be evaluated while holding certain properties, which normally equates to atomicity.

Locks fall into this category by using the `lock()` and `unlock()` primitives on lock variables. Over the years, several constructs to express atomicity have been proposed. Of these, the most widely used is the *atomic section* proposed by Lomet [Lom77], which represents the guarantee of atomicity and isolation of the enclosed block. Atomic sections have been adopted by both pessimistic and optimistic models. For the sake of simplicity, this will be the construct used to symbolize atomicity (and transactions) throughout this document.

Even while being under the same category, locks have arguably lesser visibility for the programmer. If a section of the code must protect several distinct variables, it is not unusual to see several `lock()/unlock()` calls to each variable's respective lock. This *flood* of function calls that surround the actual code obstructs it, increasing the likelihood of programmer errors, such as forgetting to unlock one of the many variables initially locked.

```
atomic { //acquire lock over ’b’

    atomic { //acquire lock over ’a’
        ... //accesses ’a’
    }
    atomic {
        ...
    }

    //only when no more locks will be acquired, release phase can start
    //lock over ’a’ and ’b’ can be released
}
```

Listing 1.3: Two-phase locking involving atomic sections.

```
class Account {
    atomicset acc ;
    atomic(acc) int balance ;
}

void transfer(unitfor Account A, unitfor Account B, int amount) {
    A.take(amount);
    B.give(amount);
}
```

Listing 1.4: Example of a data-centric approach to concurrency as seen in [DHM+12].

Similarly, despite being a *cleaner* alternative to locks, it is still possible to wrongly delimit the sequence of statements to be enclosed by an atomic section, either by excess or deficiency. The former is more common, extending the atomic sections over statements that do not belong in the atomic section, reducing performance needlessly.

The regular approach is to apply two-phase-locking that features two distinct phases: acquisition and release phase. During acquisition phase, locks can only be acquired, but not released. Once the first lock is released, no locks can be further acquired until all locks are released.

With this in mind, the resources held by an inner section are potentially only released when the outer section terminates. Another posterior interior section would still acquire locks, negative the first inner section the right to release them due to the need to remain in the acquire phase. As shown in Listing 1.3, the resources held by the two nested atomic sections can only be released at the end of the outer atomic section in order to ensure deadlock-freedom.

Data-centric approaches contrast with these. The information is conveyed through "what data" the programmer wants to evaluate atomically, not where or how. Errors associated with misplacing constructs are lessened. Protecting a group of data is permanent for the whole program.

However, data-centric approaches place a heavier burden on the compiler, since it must infer more information. In order to optimize performance, even data-centric approaches might require additional annotations. In reality, a data-centric approach will have more than just annotations on the data that wants to be protected. In [DHM$^+$12], `unitfor` is used on function arguments to indicate that the method body should evaluate a set of data atomically, an *atomic set*.

An example from this work is exemplified in Listing 1.4. Comparing to previous code-centric examples, synchronization information is all centralized on the data, making its use trivial in the rest of the code, as shown in the implementation of `transfer()`.

Avoiding duplication of responsibilities in code is tutted as the first step towards reducing the occurrence of bugs and increasing legibility of the code. In a data-centric approach, this is assured. If in the course of program development a block of code needs to be treated atomically, and it relates to the same group of data that was previously protected, no changes are required. The annotations related to this group of data are already in the code. This approach centralizes the protection and makes it very non-intrusive and avoids repetition.

As during development, the approach used also influences the methodology in debugging applications. Consider a situation where the team developing an application finds a bug related to concurrency. If that application relies heavily on code-centric concurrency control, debugging a concurrency issue involves rechecking every code-centric annotation, where such set of data may be referenced. In contrast, in data-centric mechanisms it would suffice to review how the data is protected, where it is declared, instead of rechecking the whole code.

One problem associated with data-centric constructs is its novelty. Research in this area is still very recent, especially when comparing to code-centric alternatives, such as transactional memory. As a result, no data-centric system has seen widespread usage outside of the academia.

Unfortunately, the only data-centric approach [VTD06, DHM$^+$12] requires the programmer to explicitly assemble the resources into sets and does not guarantee deadlock-freedom in all dynamic scenarios. The ideal would be if the *grouping-sets* process would be automatically generated by the compiler. The programmer would then be strictly restricted to simple annotations that would convey its intent on which data to protect.

The absence of deadlock-freedom also affects more than meets the eye. Unlike locks where programmers are arguably more experienced in identifying deadlocks, in a new paradigm such as a data-centric system, that knowledge would have to be re-acquired.

As a consequence, the absence of deadlock-freedom in a new paradigm will greatly affect its adoption, emphasizing the need for a new system that provides it.

Our proposal intends to address these problems by providing simple constructs that will be mapped into an underlying low level API that provides deadlock-freedom and atomicity in a data-centric setting.

1.3 Objectives

As elaborated in the previous section, the currently used concurrency management systems have their inherent problems. No system can be said to be objectively superior to all other systems. While the data-centric paradigm has potentially the most benefits, it must be less obtrusive and provide more security properties in order to compete with the engrained code-centric, lock-based mechanisms.

As such, our objective is to provide the programmer with simple data-centric constructs to express atomicity, while relieving it from the burdens of deadlocks and data-races. Additionally, the performance penalization should be on par with comparable systems. Failing to achieve so would be detrimental to the adaptation of this work.

To achieve such, a low level data-centric concurrency control mechanism, named *ResourceGroups* [PD12], will be used to implement the higher level constructs that will be provided to the programmer. These data-centric constructs, provided by previous research work done by Paulino and Delgado, allows multiple resources to be combined into groups that can be acquired and released, granting the thread that holds the group the right to operate on the elements of the group in mutual exclusion regarding the other threads. They guarantee deadlock-freedom and composability, properties that should be preserved by the high level constructs.

The proposed high level constructs will consist in a set of data-centric Java annotations with no explicit reference to sets (or groups) of data, contrasting with *AJ*, the current state of the art in data-centric concurrency control. Comparatively to this work, the annotations will be reduced to the variables and parameters that require synchronization.

For that purpose, the work will consists in two parts: the model which will be provided to the programmer, and how to implementing it efficiently by mapping into the *ResourceGroups* API.

1.4 Contributions

The contributions of this work are threefold:

- Definition of a data-centric concurrency control system supported by a single data-centric annotation.

- Delineation of a lock inference algorithm that guarantees both deadlock-freedom and atomicity in our system, while minimizing the overhead.

- Implementation of a functional prototype in the Java programming language.

- Evaluation of the productivity of the system developed relative to current state of the art code-centric and data-centric concurrency control systems.

1.5 Structure

This document is split into six chapters. The current Chapter attempts to provide a motivation and setting for the need to develop further research in the field of deadlock-free concurrency control mechanisms and summarizes the contributions of this thesis. Chapter 2 presents a more in-depth view on the current state of the art in concurrency control mechanisms that provide some degree of deadlock-freedom and a final remark about them. Chapter 3 presents our proposal of a data-centric concurrency control model and its properties. Chapter 4 explains the implementations of our prototype. Chapter 5 features an evaluation of our model and prototype against other state of the art concurrency control mechanisms. Finally, Chapter 6 features a final overview of the work developed in this thesis.

2

State of the Art

The work on this thesis focuses on providing a data-centric approach to concurrency management that guarantees deadlock-freedom. Firstly, it presents a brief introduction of the three philosophies for dealing with deadlocks that are adopted by the systems presented throughout this chapter. Then, current state of the art in data-centric concurrency control will be presented along with the remaining systems studied that provide some degree of deadlock-freedom. Lastly, an overview comparing all the systems and their respective benefits and pitfalls.

2.1 On the handling of deadlock situations

Deadlocks represent a real danger to software and a constant concern for programmers. Research in the area is driven by the need to provide solutions to this problem, either through systems that can either guarantee deadlock-freedom or mitigate deadlock-proneness.

There are three distinct philosophies, well documented in the literature, to deal with deadlock situations [SGG05] :

deadlock avoidance a dynamic analysis, performed at run-time, that mediates the synchronization requests. The requests are only granted when they cannot generate a deadlock. Otherwise, they are rejected or delayed until safe. This method requires more information about the use given to the resources by each process.

deadlock prevention a static analysis, performed at compile time, ensures that the four necessary conditions for a deadlock to occur are never simultaneously verified in the system. At least one of the necessary conditions must not hold at any given time.

```
class Account {
    atomicset(account)
    atomic(account) int money;

    void transfer(Account A, Account B, int amount) {
        A.take(amount);
        B.give(amount);
    }
}
```

Listing 2.1: *AJ* `atomic(s)` construct.

deadlock detection using both static and dynamic analysis, the system monitors execution progress and when a deadlock is detected, measures are taken to recover from the deadlock, such as restarting from a previously checkpointed non-deadlock state. Optimistic transactional memory uses this approach.

Next we present the most relevant approaches for deadlock-free concurrency control, grouping them according to the philosophy used to provide a varying degrees of deadlock-freedom.

2.2 Data-centric Concurrency Control

2.2.1 Atomic Sets (*AJ*)

The following works are based on the notion of an atomic group that represents the ability to group a set of resources so that they can be evaluated atomically as a single unit. They represent the basis for data-centric concurrency control management mechanisms.

The proposal in [VTD06, DHM$^+$12] introduces the notion of *atomic sets*, a data-centric alternative to code-centric synchronization mechanisms. *AJ* avoids high and low-level data-races by delegating on the programmer the specification of consistency properties between certain data that should be updated atomically.

The paper goes on to demonstrate that data-races (both low and high level) can be statically categorized into 11 groups. They can be eliminated using the constructs proposed, resulting in fewer annotations required to provide equivalent synchronization in programs. Further static analysis is used in the paper to automatically infer the points in the code where synchronization primitives are required to avoid data-races.

In total, four constructs are proposed:

`atomicset(s)` creates a new atomic set. All operations over data in this atomic set should be done atomically, in a unit of work, as illustrated in Listing 2.1.

`atomic(s)` binds a variable to an atomic set `s`.

`owned(s)` in addition to binding a variable to an atomic set `s`, binds the reference it references to the same atomic set `s`, as illustrated in Listing 2.2. Recursively expressed

```
class LinkedList {
    atomicset(list);
    atomic(list) int size;
    owned(entry) atomic(list) Entry header;

    public Object set(int index, Object value) {
        Entry e = entry(index);
        Object oldVal = e.value;
        e.value = value;
        return oldVal;
    }
}

class Entry{
    atomic(entry) Object value;
    owned(entry) atomic(entry) Entry next;
}
```

Listing 2.2: *AJ* `owned(s)` construct.

```
class Bank {
    public void transfer(unitfor Account a, unitfor Account b, int n) {
        transfer(a, b, n);
    }
}
```

Listing 2.3: *AJ* `unitfor` construct.

data structures, such as linked lists, use this annotation to unify a chain of references in the same atomic set. In the case of a linked list, since each node of the list binds itself and the next node it points to, recursively, all nodes will be included in the same atomic set. This guarantees atomicity when dealing with the list.

`unitfor` specifies function parameters that should be manipulated atomically, as illustrated in Listing 2.3. In detail, each of the variables marked as `unitfor` in a function will be treated as if they were part of the same atomic set, in the function body.

`atomic` class-constructor makes a class thread-safe by putting all of its fields and the fields of its superclasses in a single atomic set. This allows us to skip synchronization wrappers classes.

s_1`=this.`s_2 indicates that the atomic set s_1 from the annotated type is aliased with atomic set s_2 from the current object.

The implementation relies on static analysis, using a data flow analysis over a programs call graph, that infers which locks need to be held for each unit of work, that represents atomic accesses to a group of atomic sets.

A lock is associated with each atomic set. Subsequently, for each method and all methods transitively called by that method, locks are acquired for all atomic sets they

```
class Node implements INode {
 atomicset(n);
 private atomic(n) Node left|this.n<n|;
 private atomic(n) Node right|this.n<n|;
 ...
  void insert(int v){
   ... left  = new Node|this.n<n|(v); ...
   ... right = new Node|this.n<n|(v); ...
  }
 }
```

Listing 2.4: Augmenting *AJ* with ordering annotations.

could possibly access. Due to performance concerns, the locks used in this proposal allow multiple readers but only one writer, commonly known as readers-writer lock.

The locks identified in each method are acquired according to a previously established global order, effectively eliminating deadlocks in static scenarios. However, deadlocks can still occur in dynamic scenarios if (transitive) cyclical dependences between two units of work exist. Currently these scenarios are not detected, but could be detected through static analysis. As such, the author points it as a possibility for future work.

Follow-up work [MHD$^+$] attempts to mitigate this possibility by the addition of another set of annotations that allow the programmer to clarify situations that potentially lead to deadlock situations from the perspective of the compiler. Due to the nature of the system, some situations are perceived as deadlock-prone because the system cannot guarantee that resources are acquired according to a global order. This type of situation is especially prevalent in dynamic scenarios and recursive data-structures.

s_1<s_2 specifies the order of acquisition regarding two instances of the same atomic set (s_1 precedes s_2).

However, those specific scenarios might represent false-positives. Even if it is not possible for the system to infer that locks are acquired in the correct order, the underlying structure being evaluated might already inadvertently enforce an order. In these situations, the primary culprit is the lack of information conveyed to the system by the annotations or the lack of a sufficiently complex static-analysis algorithm that would allow such inferences to be made. The proposed solution is to augment the system with additional annotations that can convey the actual order of the locks to the system.

An example of this situation is present in a binary tree, where a *de facto* node order exists: the root of a subtree always precedes the nodes of the left and right branches. However, its inference is not trivial, requiring the lock ordering annotations by the programmer, as illustrated on Listing 2.4.[1]

Notwithstanding, it should be noted that the authors only mention the application of these annotations to situations regarding false-positives. Applying this approach to

---

[1] The example is taken from Figure 7 in the original document [MHD$^+$]

```
atomicset(a)
atomic(a) int l = 0 ; //part of an atomic set

function fun1() {
    int local1 = l + 1 ;
    l = local ;
}

function fun2() {
    l = -1 ;
}
```

Listing 2.5: Example of a problematic data access pattern ($R_u(l), W_{u'}(l), W_u(l)$), as identified in [HDVT08].

dynamic scenarios were no such order exists would not result in deadlock-freedom. As such, the current state of this work still does not guarantee deadlock-freedom in all scenarios.

Related work by Christian Hammer et al. in [HDVT08] changed the approach by focusing on the identification of scenarios that should be annotated, due to the presence of data-races. The data-race detection relies on dynamic analysis of a program. Similarly to previous work by Vaziri et al. in [VTD06], data-races where categorized into groups that represent problematic data access patterns. In addition to the 11 problematic groups identified in [VTD06], 3 more were added, bringing the total to 14 problematic scenarios.

The correctness criterion used is *atomic-set serializability*, where a unit of work should be serializable regarding the atomic-set used. When units of work do not respect this criterion, data-races can occur within each unit of work. Each data-race scenario found in a program has to fall under one of the 14 identified by the author. Thus, one of the contributions is on how to make such detection during runtime as efficient as possible. The notation used to describe the order of events in these scenarios is $Op_u(var)$, where *Op* can be a read (R) or a write (W) from unit of work u to variable var.

Consider the following scenario illustrated in Listing 2.5: a shared variable l is part of an atomic set and two threads are executing two functions, `fun1()` and `fun2()`, each representing a unit of work. Unit of work u reads data from variable l into a local variable, possibly modifying it, and then writes it back to variable l. Meanwhile, between the read and the write from unit of work u, another unit of work, u', writes a value to variable l.

This scenario represents the first problematic data access pattern identified in this work, $R_u(l), W_{u'}(l), W_u(l)$, corresponds to a stale read. As a result of the first write by unit of work u', the value read in by u at the beginning is stale, invalidating its serializability.

Disregarding efficiency, each scenario can be represented by a state machine, as illustrated in Figure 2.1. The first state represents the beginning, when no access was detected, while the end state represents the detection of a data-race of the specific scenario.

In order to represent this state machine efficiently, the authors used a *bitset*. A bitset is an abstract data-structure that is regularly implemented using an integer of the target

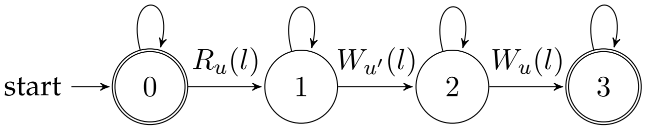


Figure 2.1: Finite state machine to represent scenario $R_u(l), W_{u'}(l), W_u(l)$.

language. In this work, each bit represents a state of the finite state machine. Since a scenario has at most 5 states, 3 bits are used to describe each state. As such, for each atomic-set a *long* value (42-bits) is used to represent all states for all scenarios.

During instrumentation, each atomic-set keeps monitoring accesses to data and updating the bitset state machines for the various scenarios. When a data-race is detected, i.e. when one of the state machines reaches an acceptance state, information about the conditions where the data-race appeared are logged into a file for posterior analysis.

2.2.2 Resource Groups (*RG*)

In [DP12], Delgado and Paulino propose a data-centric pessimistic concurrency control mechanism, *Resource Groups* or *RGs* for short.

In this work, the *resource group*, a set of data that should be manipulated atomically, is the basic concurrency management unit. These are dynamic: can change composition, be acquired, or released. Table 2.1 lists the corresponding API. The acquire and release define the scope of atomic evaluation of its resources.

As such, concerns about lock acquisition order in each resource group is abstracted by the system's API, making this work an interesting platform to be the compilation target of higher level constructs.

`newGroup` g	*Creates a new group*
g`.add(`$\bar{a}$`)`	*Add a sequence of resources to group*
g`.addRead(`$\bar{a}$`)`	*Add a sequence of resources to group using a read lock*
g`.remove(`$\bar{a}$`)`	*Remove a sequence of resources from group*
g`.acquire()/lock()`	*Acquire resource group*
g`.release()/unlock()`	*Release resource group*
g`.setPartition(`$\bar{a}$`)`	*Set the group's partition*
g`.close()`	*Close resource group*

Table 2.1: Constructs provided to manipulate resource groups.

Static scenarios represent situations when the resources to be acquired in the critical section are known before evaluating it. Accordingly, a group has to be created, added resources to and be acquired by the thread, as exemplified in Listing 2.6. Since this group is fixed, i.e. does not change size, successful acquisition of a group guarantees that the thread can safely execute the critical section without further considerations. Furthermore, since all resources are known, the system attempts to lock each resource respecting

```
void transfer(Account A, Account B, int amount)
{
  newGroup group ;
  group.add(A) ;
  group.add(B) ;
  group.acquire() ;

  A.take(amount);
  B.give(amount);

  group.release() ;
}
```

Listing 2.6: Usage of resource groups in static scenarios.

```
void remove(List L, Node node)
{
  newGroup group ;
  group.add(L.head) ; //add the head of the list
  group.acquire() ; //acquire partition

  Node current = L.head.getNext() ;

  while(current != node) {
    current = current.getNext() ;
  }

  Node previous = current.getPrevious() ;
  Node next = current.getNext() ;
  group.add(previous) ;
  group.add(next) ;

  previous.setNext(next) ;
  next.setPrevious(previous) ;

  group.release() ;
}
```

Listing 2.7: Usage of resource groups in dynamic scenarios.

a previously defined global resource order, effectively eliminating deadlocks in these scenarios.

However, in dynamic scenarios, assuring deadlock-freedom is not so simple. Groups may increase and decrease in size during critical sections and be nested inside other resource groups, effectively invalidating the previous approach taken in static scenarios. As such, the resources that potentially might be used during a critical section are not able to be identified before actually evaluating the critical section. Listing 2.7 illustrates such scenarios.

When removing an element from a doubly-linked list whose position in the list is unknown, the nodes that surround it are, by transitivity, also unknown. Although the removal should be atomic, the system does not know which nodes should be locked before the beginning of the operation. This models a fully dynamic scenario.

To avoid this, static analysis splits the application state into *partitions*. Each partition contains the elements whose interaction can lead to a deadlock. Each group is then associated with the partitions that its resources belong to, forcing the group acquisition to also acquire the partitions associated with it. Logically, since an additional layer of bonding between resources is added to groups, concurrency will be more restrained in dynamic scenarios.

However, since acquiring a resource group involves acquiring beforehand its associated partition, at most one thread will hold a partition. As such, no two threads will hold a resource group whose elements belong to the same partition, effectively assuring deadlock-freedom in dynamic scenarios.

The partitioning algorithm puts in the same partition the resources that are accessible to multiple threads and provoke circular-wait conditions. Effectively, resources are put into a partition if they generate circular chains. In addition, their acquisition either can be local (when restricted to a given object) or global. For the sake of brevity, this scope represents a way to impose different granularities on partitions in order to maximize concurrency.

Conflicts between static and dynamic acquisitions are solved by guaranteeing that a thread obliged to block during a group acquisition operation releases the remainder resources, of that same group, that it already holds. The thread will then block on the sole resource that it failed to acquire last. When that resource is again available, the thread releases it and attempts to re-acquire all locks from the start, respecting the previously established global lock order. The ordered re-acquisition of the resources guarantees that at least one thread is making progress. As a result, livelock-freedom is also guaranteed.

Composability is also provided by these constructs: nested groups can re-acquire resources or partitions acquired in outermost groups due to the use of reentrant locks.

In conclusion, this work offers a data-centric API for concurrency management that guarantees composability, atomicity, livelock-freedom and deadlock-freedom in both static and dynamic scenarios.

Nevertheless, due to its low level nature, two major problems make it infeasible for a direct replacement of more popular concurrency control mechanisms, such as locks and STMs. First, the amount of annotations required make it error-prone. If the programmer forgets to free a group after using it, the system will not work correctly. Second, the properties assured by the API depend on its correct use. That responsibility should be delegated into a compiler instead of being another concern for the programmer.

As such, creating new high-level constructs that avoid these pitfalls while retaining the positive properties provided by *ResourceGroups* will be the focus of this thesis.

2.3 Code-Centric Concurrency Control

Atomic sections are one of the most widely used constructs to express atomicity in programming languages. Systems that rely on pessimistic approaches are especially useful due to supporting irreversible operations, such as I/O, to be used atomically. The most common method to guarantee deadlock-freedom is prevention.

The following systems are motivated by the lack of these properties in legacy lock-based systems and by the fact that no runtime system is required, as opposed to approaches that use deadlock avoidance or detection. Internally, the atomic sections resort to lock-based lower-level concurrency control primitives to manage concurrency. However, these locks will need to be inferred since in most systems the only information supplied by the programmer is the position of the atomic section in the code.

The atomicity provided by these systems is only guaranteed as long as the code inside the atomic sections do not terminate unexpectedly due to an exception. If such were to happen, changes made in shared memory would not be reverted, in opposition to STMs, where the transaction would be aborted and retried.

Each system has its own method for inferring which locks should be acquired by a given atomic section. The main goal is to guarantee atomicity and composability while at the same time attempting to guarantee deadlock-freedom. While the level of deadlock-freedom guaranteed varies according to each system, most systems do not guarantee it in dynamic scenarios.

2.3.1 AutoLocker

AutoLocker [MZGB06] focus its contribution on the ability to provide superior performance compared to STMs. Nonetheless, benchmarks show that performance is slightly worse than manual implementations using coarse and fine-grained policies. Achieving equal performance would require a much more complex lock-inference system and more annotations to eliminate ambiguities that forbid optimizations used by programmers.

Concurrency control is expressed at two levels. The programmer must delimit atomic sections and, additionally, explicitly associate a lock variable to annotate the data with a lock variable, through the `protected_by` annotation.

Atomic sections in AutoLocker only guarantee exclusive access to the memory positions that have been associated to lock variables. Listing 2.8 illustrates this approach in the implementation of a function, `put`, that stores an element `v` (with given key `k`) in a hash table.

Due to the lack of annotations inside the atomic section, the programmer can fine-tune the locking mechanism, enforcing it to be more fine or coarse-grained. Tuning the granularity does not interfere with the underlying program because both annotations (`mutex` and `protected_by`) are only used by Autolocker. Listing 2.9 demonstrates the use of an individual lock per bucket instead of a global lock for the whole table, improving performance in the hash table by enabling concurrent access to distinct buckets.

The addition of a method for hash table resize would require another `protected_by` annotation to protect the whole hash table. The global lock would be at a higher level than the individual bucket locks, only allowing the use of individual locks when the

```
struct entry { int k; int v; struct entry* next; };

mutex table_lock;
struct entry *table[SIZE] protected_by(table_lock);

void put(int k, int v) {
    int hashcode = ...;
    struct entry *e = malloc(...);
    e->k = k; e->v = v;
    atomic {
        e->next = table[hashcode];
        table[hashcode] = e;
    }
}
```

Listing 2.8: The annotations required to implement an hash table using AutoLocker.

```
struct bucket {
    mutex lock;
    struct entry* head protected_by(lock);
};
struct bucket table[SIZE];
```

Listing 2.9: Augmenting the hash table from Listing 2.8 using fine-grained locks.

global lock is in read-only mode. This lock organization scheme, called multi-granularity locking, comes from database systems literature [GLPT76]. Overall, having the possibility to use such organization provides the programmer with almost full control over the code generated with minimal use of annotations.

Deadlock-freedom is also guaranteed. During the translation from the annotated code into pure C code, a correction algorithm is run to ensure that all the code produced is deadlock free. If that property cannot be guaranteed, compilation will fail. Thus, the tool might refuse to compile the code if it is unable to order locks in a way that deadlocks are not present, leading to a false-positive. This is due in part to the algorithm's conservative approach. However, the authors argue that most of the cases that produce a compilation error can be manually fixed by replacing local locks with global locks.

Since AutoLocker makes use of two-phase-locking, limitations are imposed on algorithms implemented with it. Consider a tree data structure. Deadlocks that appear obvious to the compiler might not exist due to the way the structure is accessed. A simple tree traversal algorithm in which each node traversed is locked will not compile. At each level, a lock to the previous node would be released and another to the next node would have to be acquired, triggering the false positive by the compiler.

2.3.2 Lock Inference for Atomic Sections

The work proposed by Michael Hicks et al. in [HFP06] provides a similar mechanism to AutoLocker but without requiring lock annotations.

A type-based analysis infers the abstract locations of each memory location shared between threads through pointer backtracking. For each atomic section, the compiler backtracks all pointer values to their point of origin in the atomic section. The closed set of memory locations computed represents the minimal set that has to be protected with a lock before executing the atomic section. This information is then used to generate the locks that have to be acquired to ensure atomicity in each atomic section. These locks are then released before exiting the atomic section, i.e. two-phase locking previously discussed for AutoLocker (Section 2.3.1). Locks should be reentrant since outer atomic sections also protect the data inside the inner atomic sections.

Since the number of locks potentially acquired can introduce unwanted overhead, when two or more locations are always accessed together, their locks are coalesced into a single lock. Additionally, if a location a is always accessed after acquiring the lock to another location b, the location a lock is *dominated* by location b lock. Accordingly, the lock for location a is dropped.

Deadlock-freedom in both dynamic and static scenarios is attained by enforcing a global lock order and acquiring them accordingly. As far as we can conclude, due to the need to assess statically all references, deadlock-freedom in dynamic scenarios is assured by sacrificing performance. When a relation of dominance is not able to be established due to circular dependencies, the algorithm reverts to a global lock that protects the affected atomic sections. This detail is not explicitly mentioned in the work, but a similar technique is assumed to be used.

Consider a linked list. Even thought the amount of nodes it can contain is unbounded, the algorithm will only protect the list backbone. If a method returned a random node from the list, its lock would not be the same as the list backbone due to a different memory location and violate atomicity. But if the previously mentioned global lock is used, atomicity would be preserved.

2.3.3 Lock Allocation

The work proposed by Emmi et al. in [EFJM07] computes the locks required through backtracking of pointers locations. Each atomic section is then associated with locks that represent resources used in that section. Resources that are unknown or that might have multiple locations are said to be *may-alias*. The locks are then modelled as a constraint optimization problem in order to reduce the number of locks used while maintaining atomicity and deadlock-freedom.

The nesting of atomic sections is, once again, supported by enforcing a two-phase-locking in similar fashion to AutoLocker (Section 2.3.1). The resources acquired by the inner atomic sections are only released when the outer atomic section finishes.

In static scenarios, imposing a global lock order guarantees the absence of deadlocks due to the orderly acquisition of the locks at the entrance of an atomic section. However, this approach is not enough to guarantee deadlock-freedom in dynamic scenarios since the set of locks represented by *may-alias* variables has to be dynamically determined. In those cases, *may-alias* variables are assigned a single global lock. Given that they can point to multiple locations during the execution of the program, assigning them to a single global lock guarantees both atomicity and deadlock-freedom in dynamic scenarios. Hence, concurrency is sacrificed in order to guarantee deadlock-freedom in dynamic scenarios.

In conclusion, deadlock-freedom is assured in both static and dynamic scenarios.

2.3.4 Inferring locks for atomic sections

The work proposed by Cherem in [CCG08] makes use of multi-granular locking seen in database systems to provide weak atomicity and deadlock-freedom.

Consider a *fine lock* that represents a resource that is acquired inside an atomic section where a *coarse lock* is already held. Multi-granular locks permit the locking of individual finer locks without holding the coarser lock, instead marking it with an *intention*. The intention conveys that at least one fine lock is currently held by a thread, so the coarse lock cannot be acquired. When all fine locks are released, the coarse lock loses its *intention*, regaining the ability to be acquired by a thread.

The association between resources and the locks that should be acquired is done through pointer backtracking analysis, as described in Section 2.3.2.

If the number of locations in an atomic section is fixed and under a certain threshold, each memory location has a fine-grain lock associated. Otherwise, such as in dynamic scenarios where the number of elements accessed in a atomic section is unbounded, a single coarse-grain lock is used. This technique reduces the contention when the number of locks is high and ensures deadlock-freedom in dynamic scenarios by using a coarser lock that encases all potentially accessed elements.

Deadlock-freedom is guaranteed in both scenarios as in [EFJM07, HFP06].

2.3.5 Deadlock Avoidance

The deadlock avoidance strategy relies on having a runtime that avoids transitions in the system's state that might result in a deadlock.The system delays the acquisition of locks deemed deadlock-prone until they are considered safe. As such, the two systems here overviewed resort to static analysis and a runtime for deciding lock acquisitions. However, they resort to different methods to evaluate the *safeness* of a lock acquisition.

The purpose of the static analysis is to identify cases where deadlocks can be present, delegating a significant part of the computation offline. This allows the runtime code to focus solely on avoiding them as the execution unfolds, minimizing runtime overhead.

2.3.5.1 Shared Memory Deadlock-Free Semantics

In the work by Boudol presented in [Bou09], the runtime ensures that states that might lead into a deadlock are not allowed by refusing to lock a memory location when it anticipates to take a memory location that is held by another thread.

The information on which memory locations that are held by a certain thread is anticipated by using a type and effect system, that uses prudent semantics to prove the deadlock-freedom of the lock. The runtime then disallows the locking of a pointer whenever it anticipates to take some other pointer that is currently held by another thread, effectively preventing deadlocks.

This is achieved by translating an extension of CoreML with a type and effect system into a runtime language. For each expression, the system computes the finite set of pointers that it might have to lock during its execution, and adds a lock over the set of pointers before evaluating the expression. Since all locks possible of acquisition are already acquired when the expression is evaluated, deadlocks are avoided. As a consequence, for every closed expression of the source language that is typable, its translation for the target language is effectively free from deadlocks. However, this can only be applied to static scenarios.

2.3.5.2 The theory of deadlock avoidance via discrete control

Contrasting with Boudol's work, Wang et. al [WLK+09] introduces the use of Discrete Control Theory (DCT) as a tool for dynamic deadlock avoidance in concurrent programs that use lock-based synchronization primitives, without requiring additional annotations. Not requiring additional annotation presents an interesting possibility, in which the system would be used on legacy lock-based code to provide deadlock-freedom without modifications.

DCT is a branch from control theory that describes system states and transitions, normally modelled as Petri Nets. These models can later be subjected to conditions to identify states that are prone to deadlock. For this purpose, the Petri Nets are augmented with synchronization primitives and conditions that govern state transition. In a Petri net, a siphon represents a group of states where the set of possible output transitions is empty. Since no transitions to outside of the siphon are possible, the system state is effectively trapped inside the siphon. For this reason, the authors argue that siphons represent deadlock-prone states.

Instrumentation code is injected to take place immediately before synchronization primitives, such as locks, are invoked. Before allowing a synchronization primitive to execute, the runtime determines whether the resulting target state is deadlock-prone. The transition is then stalled until the target state is deemed deadlock-free.

In dynamic scenarios, when the locks to be acquired are unknown, the system takes a conservative approach and holds a lock using its own type. In the worst case scenario, when all locks share the same type, the performance is downgraded to that of a

```
void transfer(Object Object, Object Object, int amount)
{
  atomic{
    A.take(amount);
    B.give(amount);
  }
}
```

Listing 2.10: Atomic section in a transactional memory environment.

type-wide lock. As a result, deadlock-freedom is still assured in dynamic scenarios by compromising performance.

2.3.6 Transactional Memory (*TM*)

Software transactional memory (STM) as a concurrency control mechanism grows from the transaction concept found in database systems [HLR10]. Namely, from the optimistic model used in databases that enables concurrent transactions and consequently taking advantage of the new multi-core architectural paradigm. Furthermore, synchronization intensive applications exhibiting a low conflict rate tend to benefit the most from the scaling provided by concurrent transactions [PW10].

The STM paradigm favours compact semantics and ease of use in a concurrent environment, as exemplified in Listing 2.10. The support for nested atomic sections enables code compositionality. For instance, in an object-oriented programming language, the implementation of a given method is not influenced by the use of atomic sections in lower levels of the call stack.

Of the properties originally guaranteed by database transactions (ACID) [HR83], transactional memory provides two of them:

Atomicity Changes performed by the transaction are made visible to the outside at the same time, when the transaction commits successfully. STMs normally offer *weak atomicity*, where atomicity is only guaranteed among transactions [HLR10].

Isolation Concurrent transactions do not observe the internal state of other transactions, and therefore do not interfere with each other's result.

The remainder two, consistency and durability, are not guaranteed in transactional memory since supporting them would imply a considerable overhead similar to that of database systems [MBM+06].

Statements in the atomic section have to maintain these properties or abort, and retry later. The behaviour in the abort scenario can differ across distinct STM implementations. The unclear semantics associated with transactions across multiple implementations poses one of the reasons why STM only achieved negligible use in real applications [Boe09].

Transaction properties are either managed by a runtime system during the execution or statically handled by the compiler, depending on the system in question. Normally, the static approach is not as flexible but has less of an impact on the performance.

The transaction normally relies on two important mechanisms to assure atomicity: commits and rollbacks. Commits are performed at the end of a transaction, and defines the point when the changes performed by such transaction are made visible to the outside (the commit itself also has to be atomic). A transaction cannot commit when the changes it wants to apply to shared data conflicts with changes made in the data since the time the transaction started.

Since STMS roll back when a deadlock or conflict is detected, transactional memory is deadlock-free in both static and dynamic scenarios. At most, it will keep rolling back due to conflicts, that might lead to a livelock, but never enter a state of deadlock.

Two main approaches can be taken regarding the concurrency control between transactions: optimistic and pessimistic.

2.3.6.1 Conflict detection and resolution

With pessimistic concurrency control, if a conflict occurs, it is detected and resolved at the same moment. Once a transaction is initiated, all further possible conflicts are detected before they interact with the data, avoiding future conflicts. In other words, once a transaction starts, it holds exclusive access to that data. In this case, deadlocks that occur have to be resolved and STM does it through deadlock-detection.

On the other side of the spectrum, optimistic concurrency control permits for delayed detection and resolution of conflicts. This allows concurrent transactions to take place, leaving to the runtime system the detection of possible conflicts before committing changes.

When contention is high, the pessimistic approach will provide for better performance due to the decreased number of rollbacks made compared to the optimistic alternative. In that case, most transactions would execute only to the aborted because they were contending with other transactions for the same data and the conflict was only found in the end of execution.

On the other hand, the optimistic implementation is best suited for scenarios where multiple transactions are not expected to compete over the same data.

However, some instructions cannot be rolled back, such as I/O routines or system calls, and as a consequence are not allowed inside transactions. Even if I/O buffers where to be copied and rolled back, sent packets would already be sent, regardless of whether the transaction rolls back or not. Works in this direction have tried to mend this flaw and provide for unrestricted transactions that don't rely on complex implementations.

2.3.6.2 Supporting I/O and System Calls within Transactions

The work in [BLM06] proposes a method for supporting I/O operation and system calls in a unbounded conventional transactional memory system.

For that purpose it specifies two modes of execution:

restricted transactions limits transaction size, duration and content but is highly concurrent.

unrestricted transactions unbounded, but allows I/O and system calls. Only one is allowed at any time, limiting concurrency.

Such support for I/O and system calls is based on the assumption that the oldest transaction in the system will never roll-back or abort, because in the case of conflict, younger transactions are aborted instead. This conflict resolution algorithm is used on several transactional proposals [AAK+05, MBM+06, RG02].

Even though the authors provide two distinct implementations, a naïve and an optimized one, in this document we will restrict ourselves to the latter. The optimized implementation allows for concurrent execution of multiple restricted transactions and a single unrestricted transaction. This provides for highly concurrent execution of transactions, assuming that the transactions using unrestricted mode due to I/O or system calls in a program are a minority.

This proposal, however, requires dedicated hardware support: an interface that includes pairs of instructions for initiating and terminating each of the two modes of transactions, and two word-sized registers (*STSW* and *PTSW*) used for controlling transactions without introducing significant overhead.

2.3.7 Adaptive Locks

Adaptive Locks [UBES09] takes a completely different approach from previous works and plays on the fact that performance of pessimistic and optimistic concurrency control varies according to the execution scenario. While it relies on regular atomic sections that ensure atomicity, their mode of execution is dynamically alternated between optimistic and pessimistic in order to achieve maximum performance. Runtime holds the responsibility of choosing and dynamically applying the mechanism that should yield the best performance in each occasion.

For that purpose, the two modes represent mutex locks or transactional memory. Decision on whether to run an atomic section using a certain mode is based on the observed behaviour of the program. During runtime, three factors are statistically evaluated for each atomic section to make such decision:

nominal contention (*c*) number of threads blocked on the lock in mutex mode. It represents the benefit of executing in transaction mode rather than in mutex mode.

actual contention (a) number of times each transaction retries in transactional mode. It represents the competition by other threads over a shared data on the critical section.

transactional overhead (o) a measure of the overhead associated with executing a transaction in this critical section, calculated based on the proportion of shared memory operations in a transaction.

These three variables are taken into account by the balancing algorithm, which decides whether or not to change the execution mode based on potential performance gain. The cost-benefit analysis abides to the following formula:

$$a.o \geq c$$

In case of tie, the system opts for the mutex mode.

The statistics' acquisition overhead is almost negligible due to several optimizations, like using CAS (compare-and-swap) instructions in the decision algorithm and updating the statistic counters in a non-atomic way, allowing data-races in the statistics' updates. Updating the statistic counters would represent an important bottleneck if it had to be serialized in all critical sections. For this reason, the authors decided that the skew arising from concurrent updates is sufficiently negligible, and thus have opt to ignore it.

However, since any of the two mechanisms can be enforced at runtime, only common features can be taken into account. As such, while isolation is guaranteed, atomicity and deadlock-freedom are not. Additionally, due to the interchange between STM and mutex mode, composability is also not guaranteed.

2.4 Final remarks

As possible to infer from Table 2.2, the most uniform advantage of the presented systems compared to the *de facto* concurrency control mechanism, mutex locks, is composability. This property alone greatly helps programmers build modular software.

The weak atomicity provided by systems other than transactional memory is only applicable if the code being executed during an atomic section does not terminate unexpectedly due to an exception, as mentioned in Section 2.3.

In regards to their implementation, only works based on deadlock-prevention are static. The remaining works, based on deadlock-detection and avoidance, rely on a runtime system to manage the presence of deadlocks and atomicity where applicable.

While some code-centric approaches [HFP06, EFJM07, CCG08] manage to provide a full spectrum of properties that benefit the programmer, only one data-centric approach [PD12] manages to guarantee deadlock-freedom in all dynamic scenarios. As such, this system will represent the basis on which the work in this thesis will be built upon.

	Deadlock-free scenarios					
Approach	Static	Dynamic	Livelock-free	Atomicity	Isolation	Composable
Mutex locks	No	No	Yes	No	Yes	No
Deadlock-detection						
Trans. Memory	Yes	Yes	No	Weak	Yes	Yes
Adaptive Locks	No	No	No	No	Yes	No
Deadlock-prevention						
AutoLocker	Yes	Fails to compile	Yes	Weak-	Yes	Yes
Inferring Locks	Yes	Yes	Yes	Weak-	Yes	Yes
Lock Inference	Yes	Yes	Yes	Weak-	Yes	Yes
Lock Allocation	Yes	Yes	Yes	Weak-	Yes	Yes
AJ	Yes	Mostly yes	Yes	Weak-	Yes	Yes
Resource Groups	Yes	Yes	Yes	Weak-	Yes	Yes
Deadlock-avoidance						
Boudol	Yes	No	Yes	No	Yes	Yes
DCT	Yes	Yes	Yes	No	Yes	Yes

Table 2.2: Comparison of legacy and state of the art systems.

3

From Atomic Variables to Data-Centric Concurrency Control

In this chapter, we introduce a data-centric concurrency management model. We provide an informal presentation of both the model's syntax and semantics, followed by a discussion about the impact of our construct in the underlying type system, along with its reasoning and the properties we can derive from it.

3.1 Syntax

Our proposal relies on the identification of objects in memory whose value should be accessed atomically, in both read and write accesses. In essence, the declaration of these objects should convey to the system that intent, so the latter can enforce the desired properties when the program is subsequently executed.

An issue to consider is the simplicity of the proposed constructs, which has to be carefully balanced. On the one hand, several distinct annotations can provide the compiler with additional information to further optimize the generated code, but burdens the programmer with additional responsibilities. On the other hand, a simpler, less complex approach, releases the programmer from these concerns and delegates it on the underlying automated system. Yet, this approach requires additional static analysis and a stronger lock inference engine. The simple use of the model must not, however, limit the programmer's ability to naturally express concurrency in both simple and complex scenarios. Conversely, the programming construct must supply enough information for the compiler to generate efficient concurrent code while guaranteeing two important properties: **atomicity** and **deadlock-freedom in all scenarios**.

In the face of this trade-off, our approach is to subsume as much concerns in the compiler and runtime system as possible and, hence, relief the programmer from those concerns. As such, we propose the use of a single data-centric concurrency control construction for the target programming language. The main asset provided is the ability to express concurrency using a simpler mental process compared to legacy, code-centric systems due to the delegation of concerns to the underlying system.

The specification of concurrency control in our model is thus entirely built around a single annotation: `@Atomic`. This annotation is applicable to all variable declaration, namely, class fields, function parameters, and local variables declarations.

The act of using a single annotation, with a single meaning, eases considerably the development of software compared to systems that provide to the programmer several constructs with distinct semantic meaning [VTD06]. As a consequence, we claim that our model is easier to use and provides the programmer with higher productivity than the aforementioned models, while maintaining a high degree of expressivity.

3.2 Semantics

The reasoning of the proposed concurrency control model revolves around the annotation of variables that must be evaluated atomically in a unit of work. In the scope of this document, an annotated variable will be referenced to as an *atomic variable* and a memory object referenced from an atomic variable as an *atomic resource*, or simply *resource*.

Definition 1 (Atomic Variable). *A variable annotated with* `@Atomic`*.*

Definition 2 (Atomic Resource or Resource). *A memory object reachable from an atomic variable.*

The semantics of the model builds from another construction traversal to most programming paradigms, the function. This construction will be our basic scope of atomicity, its body will delimit the scope of atomic evaluation for all memory objects reachable from atomically annotated variables within that same body. It is the equivalent to the unit of work in *AJ*. The scope of atomic execution is limited by either the return of a function or the raising of an exception.

Definition 3 (Unit of work). *The function defines the scope of atomic evaluation of a set of atomic resources, and thus it is the model's unit of work.*

Definition 4 (Semantics). *Let $\mathcal{R}$ be the set of atomic resources referenced by the atomic variables in the scope of a function f. The model guarantees all atomic resources belonging to $\mathcal{R}$ are atomically evaluated, as a whole, in the entire scope of f.*

As such, to perform an atomic operation over a set of resources, the programmer only has to reference them in the same function.

Note that in order not to limit the use of nested units of work, resources are reentrant, in the sense that they may be acquired multiple times in nested functions. As expected, a resource acquired in a function is considered acquired in the context of inner functions.

Axiom 1 (Resource thread isolation)**.** *Message-oriented communication may not exchange references to resources.*

A precondition for the application of the model is the absence of communication inside a unit of work that involves the exchange of references to resources between threads. This limitation prevents the occurrence of circular dependency scenarios regarding resource acquisition that cannot be inferred from static analysis due to threads exchanging references to resources.

Note that the atomic resources referenced in a unit of work can change during its execution. However, as described in Definition 4, the model guarantees that all the resources referenced in a unit of work at any point in time are atomically evaluated.

To illustrate the use of our system, two main examples will be presented and revisited throughout the document: a **bank** simulation and the implementation of a (simply-)**linked list**. The bank allows the withdraw and deposit of money in accounts, and the transfer of money between two accounts. The linked list on the other hand represents a fairly straightforward implementation, using a single forward reference in each node. This implementation supports adding, removing or checking the presence of an element in the list.

In the **bank program**, illustrated in Listing 3.1, the intent is to ensure that all accounts are accessed under atomicity. Respectively, the structure that stores the accounts has its `Account` polymorphic type annotated with `@Atomic` (line 2). Since the map is not itself annotated, it is not atomically evaluated, but rather the accounts attained from it. At this point, the accounts attained from this map are effectively atomic resources. As such, when assigned to variables, they should be also annotated with `@Atomic` to convey that the (atomic) variable holds an atomic resource (lines 6 and 7).

In the `transfer` function of the `Bank` class, the scope of atomicity enforced only includes the third statement (line 8). It represents the first access to an atomic variable in the function, since the previous two declaration of local atomic variables (lines 6 and 7) do not contain accesses to atomic variables (the map is not considered an atomic variable). However, in the `transfer` function in the `Account` class, the programmer also has to annotate the parameter of type Account with `@Atomic` (line 17) because the accessed accounts, `dst` and `this` (line 18) should also be atomically evaluated in the scope of this transfer function. Additionally, since the function is invoked with an atomic variable of type `Account`, the parameter also has to be an atomic variable, hence requiring the `@Atomic` annotation.

The **linked list** implementation shares a similar intent to the previous example: guaranteeing that the linked list is traversed and modified atomically. To achieve this, the programmer starts by annotating with `@Atomic` the head of the list, `head` (line 16), and

```
class Bank{
    Map<Integer, @Atomic Account> accounts;

    void transfer(int srcAccNumber, int dstAccNumber, int amount)
    {
        @Atomic Account src = accounts.get(srcAccNumber) ;
        @Atomic Account dst = accounts.get(dstAccNumber) ;
        src.tranfer(dst, amount);
    }

    (...)
}

class Account{
    float m_balance ;

    void transfer(@Atomic Account dst, int amount) {
        dst.deposit(this.withdraw(amount));
    }

    (...)
}
```

Listing 3.1: Syntactic application of our system to a bank transfer between two accounts.

the forward reference in each node, `next` (line 2). However, these two annotations do not suffice. The functions of the `LinkedList` class are still dealing with non-atomic local variables that reference nodes of the list. To ensure the linked list is traversed atomically in those functions (`add`, `remove`, and `contains`), their local variables should also be annotated with `@Atomic`.

Focusing on the `add` function, the scope of atomicity in this unit of work ranges from the first access to the atomic variable `head` until the last access to the atomic variable `prev` (line 31 and 43, respectively). This scope is especially interesting since it illustrates a scenario where the composition of the resource set in the function evolves in time. Both the `next` and `prev` atomic variables are assigned different atomic resources to the `next` and `prev` atomic variables as the algorithm iterates over the list (lines 23 and 37).

3.2.1 Deadlock-freedom

While the model itself makes no guarantee regarding deadlock or livelock-freedom, the lower level concurrency mechanism that will be used to implement this model already guarantees livelock-freedom. Assuring deadlock-freedom, on the other hand, requires the identification of deadlock-prone resource acquisitions (represented by circular dependencies between them) and associating them a coarse-grained lock, assuring their serialization and respective deadlock-freedom. As such, and given the semantics presented, we claim that it is possible to build a system based on pessimistic concurrency control mechanism that fulfils these requirements while assuring that at least one thread is making progress at any given point in time.

```
class Node {
  @Atomic Node next ;

  void setNext(@Atomic Node node) {
    next = node ;
  }

  Node getNext() {
    return next ;
  }

  (...)
}

class LinkedList {
  @Atomic Node head;

  public boolean contains(int value) {
    @Atomic Node next = head.getNext();

    int v;
    while ((v=next.getValue()) < value) {
      next = prev.getNext();
    }

    return (v == value);
  }

  public boolean add(int value) {
    boolean result ;
    @Atomic Node prev = head;
    @Atomic Node next = head.getNext();

    int v;
    while ((v=next.getValue()) < value) {
      prev = next ;
      next = prev.getNext();
    }

    result = (v != value) ;
    if (result) {
      @Atomic Node node = new Node(value,next);
      prev.setNext(node);
    }

    return result;
  }

  (...)
}
```

Listing 3.2: Syntactic application of our system to the implementation of a simply-linked list.

3.3 Typing

We want our annotation to have an impact on the type of the resources reachable from atomically annotated variables. We denote by $\tau^{@}$, the type of an atomically annotated variable of type τ.

Definition 5 (Typing)**.** *Let* $<:$ *denote the subtyping relation in the hosting language type system.* τ *and* $\tau^{@}$ *share the same interface but* $\tau \not<: \tau^{@}$ *and* $\tau^{@} \not<: \tau$*.*

The omission of any of the annotations will produce a typing error, warning the programmer to the missing annotation and refusing to compile. Additionally, whatever is the super or subtype of the annotated type, there is a corresponding atomic type.

Proposition 1 (Subtyping)**.** *For a given type* τ*,*

$$\forall \tau' : \tau' <: \tau \Rightarrow \exists \tau'^{@} : \tau'^{@} <: \tau^{@}$$
$$and\, \forall \tau' : \tau <: \tau' \Rightarrow \exists \tau'^{@} : \tau^{@} <: \tau'^{@}$$

Thus, $\tau^{@}$ *cannot be cast to a non-atomic type.*

Both types τ and $\tau^{@}$ share the same interface and implementation, but are not subtypes of each other. Therefore, the type is not interchangeable between an atomic variable and its non-atomic correspondent. The type system does not allow invocation of function by passing an atomic type argument to a non-atomic type parameter, and vice-versa. It is also not possible to type-cast an atomic type to a non-atomic type without producing an (compile or runtime) error. As such, an atomic type cannot be accessed outside an atomic scope. Additional, all aliases of an annotated variable will also share the same type, therefore sharing its atomic type. Likewise, any partial access is also atomic.

Incrementally, atomic types also implicitly impact the return type of functions and the type of the constructors. A function that returns a type value of $\tau^{@}$ must have its type modified accordingly. Likewise, an assignment from a constructor of type τ to a variable of type $\tau^{@}$ must also be converted. Since the original function returned a non-atomic type (τ), returning an atomic type ($\tau^{@}$) would result in a failed compilation. As such, the return type of the function has to be also converted (Equation (4.5)). Constructors also have to be converted if a constructor of a non-atomic type is being assigned to an atomic variable (Equation (4.3) and (4.4)).

In overview, an atomic resource is always accessed via an atomic variable. As such, an atomic resource is always evaluated in the scope of an atomic function. This property is the backbone for the centrality of our concurrency control mechanism.

Theorem 1 (Correction)**.** *A well-typed program does not access atomic resources from outside an atomic function.*

Proof. Definition 5 and Proposition 1 imply that an atomic type is not interchangeable with a non-atomic type. Type-casts from an atomic type to a non-atomic type will also

```
1  class X {
2    @Atomic T var = ... ;
3    T normal-var = ... ;
4
5    ...
6
7    void go() {
8      start(var) ; //TYPE ERROR
9      end(normal-var) ; //TYPE ERROR
10   }
11
12   void start(T param1) {
13     @Atomic T local1 = param1 ; //TYPE ERROR
14   }
15
16   void end(@Atomic T param2) {
17     T local2 = param2 ; //TYPE ERROR
18   }
19 }
```

Listing 3.3: Compilation failures of a program where the atomic types were not propagated correctly.

result in an error. Additionally, Definition 4 implies that an atomic variable is always evaluated atomically in the scope of the function that accesses it. Thus, an atomic resource can only be accessed in the scope of an atomic function, where it will always be atomically evaluated. □

Corollary 2 (Strong Atomicity). *An access to an atomic resource is guaranteed to be atomic against all remainder accesses to that same resource.*

To illustrate this concept, Listing 3.3 represents a sample program that fails to compile due to incorrect propagation of our atomic annotations. In line 8, an atomic variable `T` is passed as argument to a function that takes a non-atomic argument. Even though their original type is the same (`T`), the type of the annotated variable ($\mathtt{T}^{@}$) is not a subtype of `T`, which results in a type error in this invocation. The inverse takes place in line 9, where a non-atomic variable fails to typecheck against an atomic variable parameter. Lastly, both lines 13 and 17 feature another type error. This time, both local variables feature opposed atomic-statuses regarding the parameter they are aliasing, effectively failing the typecheck.

This propagation of atomic types could easily be inferred by the compiler using a call-graph analysis. However, we argue that its explicit annotation by the programmer improves the legibility of the code and notion that the programmer has about where the concurrency control is being effectively enforced.

To exemplify the inference of atomic types possible in our system, let us return to the bank application and linked list examples, now after performing the respective type transformations discussed. Here we illustrate the concept of *primary* and *derived* annotations.

```
class Bank{
    Map<Integer , Account@> accounts;

    void transfer(int srcAccNumber , int dstAccNumber , int amount)
    {
        Account@ src = accounts.get(srcAccNumber) ;
        Account@ dst = accounts.get(dstAccNumber) ;
        src.tranfer(dst, amount);
    }

    (...)
}

class Account{
    float m_balance ;

    void transfer(Account@ dst, int amount) {
        dst.deposit(this.withdraw(amount));
    }

    (...)
}
```

Listing 3.4: Type mutation applied to the bank transfer example.

Both Listing 3.4 and Listing 3.5 represent, respectively, the state of both the bank program and the linked list examples after going through type mutation. An erroneous application of our annotation would generate type errors.

A simple exercise using the linked list illustrates the inherent centrality of our system. The starting point in this exercise is the linked list implementation without no annotations whatsoever. Then, we start by annotating the head of the list (`head`) at line 16. After attempting to compile, the compiler would then quickly return a type error in line 19 and 31 due to the assignment of an atomic variable (`AtomicNode`) to the non-atomic variable (`Node`). The programmer would then correct these by annotating these two variables. This procedure would go on until the programmer eventually reached the correct state of the code previously presented in Listing 3.2. As such, the `head` annotation represents our *primary* annotation, the one from which all other *derived* annotations can be inferred.

The example also showcases the application of the type conversion rules to the return of method results required to maintain type coherency. The return type at line 8 had to be converted from `Node` to `AtomicNode` because the returned variable is atomic. At line 41, the constructor had to be converted from `Node` to `AtomicNode` because it is now being assigned to an atomic variable.

3.4 Final remarks

At the end of this chapter, we have defined a model for data-centric concurrency mechanism based on a single annotation. The following chapter will address the implementation of this model on the Java programming language.

```
class Node {
  Node@ next ;

  void setNext(Node@ node) {
    next = node ;
  }

  Node@ getNext() {
    return next ;
  }

  (...)
}

class LinkedList {
  Node@ head;

  public boolean contains(int value) {
    Node@ next = head.getNext();

    int v;
    while ((v=next.getValue()) < value) {
      next = next.getNext();
    }

    return (v == value);
  }

  public boolean add(int value) {
    boolean result ;
    Node@ prev = head;
    Node@ next = head.getNext();

    int v;
    while ((v=next.getValue()) < value) {
      prev = next ;
      next = prev.getNext();
    }

    result = (v != value)    if (result) {
      Node@ node = new Node@(value,next);
      prev.setNext(node);
    }

    return result;
  }

  (...)
}
```

Listing 3.5: Type mutation applied to the implementation of a simply-linked list.

4

Implementation

The proposed data-centric concurrency control model will be implemented through the mapping of the higher-level `@Atomic` annotation to the lower-level concurrency control system *ResourceGroups* [DP12]. This system permits the deadlock-free acquisition of resources, but requires their explicit aggregation in groups. Part of our work will attempt to delegate this reasoning into the compiler.

By itself, the objective of the compilation process is to generate Java *bytecode* with invocations to the *ResourceGroups* API. Two approaches could be taken to achieve this: compile the program normally and then use a framework for bytecode manipulation and modify the required code, or carry out this modifications in the source language directly. The latter was selected, due to the fact that the end result would be closely related to an implementation on an actual compiler, meaning it could be integrated directly into the workflow of programmers without reliance on external applications or frameworks. Another advantage of working at the source level is the ability to manipulate the type system, that is partially lost once the program is compiled. Furthermore, it would be possible to compile bad-typed programs. However, this approach was completely undocumented in the literature, amassing inertia to the initial development. The only source of knowledge was the actual source code of the implementation of the chosen compiler. As a result, the documentation of this procedure along with the particularities of the compiler used will also represent a minor contributions of this thesis.

The compiler chosen for this purpose was *OpenJDK8*[1], since it represents the most popular open-source Java compiler that is actively developed. The eighth version of the Java language was chosen due to the added support for type annotations in polymorphic types.

---

[1] Currently in testing phases, official release scheduled for 2014.

The introduction of our concept is done by using the new plugin system in *OpenJDK8*, similar to annotation processing support in past versions of the Java language. The new plugin system allows the binding of user classes to a view of the abstract syntax tree (AST) in the compiler (JSR 199[2] and 269[3]). However, this view is obfuscated by design. No modifications can be done and several properties are hidden, such as the symbols attached to each AST node. In order to achieve this obfuscation, the classes provided to the programmer to explore the AST are mere facades of the actual internal compiler classes used to hide their full functionalities.

Those limitations, albeit intended by the developers of the compiler, can be overcome, as partially documented in [EK08]. By casting those facade classes into their inner types, the compilers internal (modifiable) AST nodes and related helper classes are uncovered. The framework developed during this work to access the hidden functionalities of the *javac* compiler is available as an open-source project at `https://github.com/metabrain/openjdk8-javac-plugin-framework`.

As such, this chapter will begin by introducing the selected *javac* compiler, its internal functioning and stages of compilation.

4.1 A Brief Introduction to *javac*

Before dwelling further inside the implementation, one must understand how the compilation in *javac* works, more specifically, the distinct stages of compilation and their properties. But first, some core aspects and terms have to be defined. Namely the distinction between the abstract syntax tree and the symbols.

4.1.1 Nodes and symbols

The abstract syntax tree represents, more or less faithfully, the syntactic constructs used. Each language construction has a node, with several distinct capabilities depending on the type of node. But more importantly, most nodes have associated a symbol, and a type object, if applicable. However, during most of the compilation, this type object is effectively uninitialized: its initialization only occurs during the late *Analysis* stage.

Regarding the symbols, they are analogous to the AST nodes, only poorer. Less information is delegated onto these symbols, in part because the symbols represent the compiled state of a node. More prominent AST nodes, such as class or method nodes, have rich symbols. This is justified by the fact that classes and methods are the most relevant structures in the actual bytecode. In specific, they contain the classes' names, methods' names, parameters and their types, field variables with their types and offset on the stack frame, relevant flags, and so on.

Another thought, and a reasonable one at that, would be that those symbols are basically derived from their AST nodes, and as such, only AST nodes require modification.

---

[2] *JSR 199: JavaTM Compiler API* `http://www.jcp.org/en/jsr/detail?id=199`

[3] *JSR 269: Pluggable Annotation Processing API* `http://www.jcp.org/en/jsr/detail?id=269`

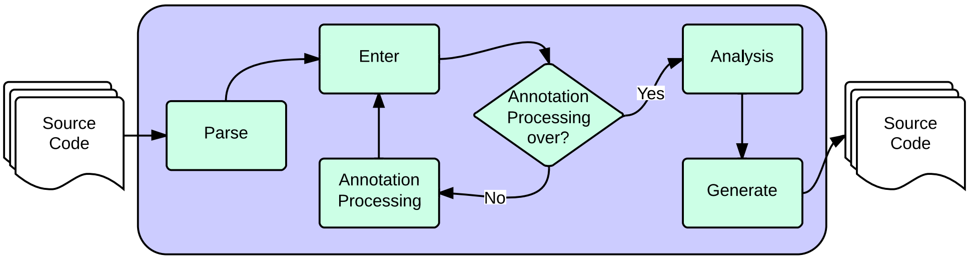


Figure 4.1: Compilation stages in the *javac* compiler.

But that is not strictly true. Some symbols already have information saved during the *Parsing*. For example, the stack offsets for the fields in each class is calculated during this phase. Therefore, if for example, we wanted to inject a field into a class, we would have to modify those stack offsets manually. As such, a complex process is further complicated due to the decentralization of responsibilities across both symbols and AST nodes.

The manipulation of symbols that represent already compiled classes is also limited. Those limitations are imposed due to the fact that they are the representatives of compiled bytecode. As such, it is not possible to modify the types of functions and their parameters in previously compiled code.

4.1.2 Compilation stages and considerations

The compilation process of the *javac* compiler, illustrated in Figure 4.1, is divided into five main stages:

- *Parse* - source *.java files are parsed into the AST, and symbols from Java standard libraries, external libraries and other *.class files from the classpath are loaded into the compiler.

- *Enter* - each AST node is augmented with information regarding variable scoping and shadowing. Some symbols are created for some more prominent AST nodes.

- *Annotation Processing* - not a real compilation stage since the compiler does not do anything in this stage unless the user specifies an annotation processor. It is an artificial stage proposed by JSR 175[4] and formalized by JSR 269[5] integrated into *javac* version 1.6. It allows the use of an annotation processor that can be used to generate new source files and explore a reduced, obfuscated, view of the AST.

---

[4] *JSR 175: A Metadata Facility for the JavaTM Programming Language* `http://www.jcp.org/en/jsr/detail?id=175`

[5] *JSR 269: Pluggable Annotation Processing API* `http://www.jcp.org/en/jsr/detail?id=269`

- *Analysis* - one of the more important phases where more verifications take place. To be more specific, most type checking, polymorphic expansions, cyclic dependency verification and desugaring take place here. At the start of this phase, the AST nodes have it's type calculated and saved. This information is used on the verifications performed in this stage.

- *Generate* - the last phase of compilation. Each AST corresponding to a class is compiled into its respective class file. During this process, the AST nodes are consumed, leaving behind a skeleton of its former self, representing the information attainable from the class files. In other words, they become a symbol.

These stages do not take place in a strict order. The *Annotation Processing* takes place after *Enter*, after which the *Parse* phase is repeated for the newly generated files, if present. Afterwards, the *Enter* phase is again repeated on all files, now including the newly generated ones, so the changes from the new files are propagated to older classes. This cycle goes on until the annotation processor signals the compiler that it has finished.

The rest of the stages are fairly straightforward. The new plugin system allows us to attach our plugin at the start or end points of each stage of compilation. As such, a decision has to be made about where to perform our modifications. But alas, it is not an easy decision. The obvious answer would be the *Analysis* stage: our model requires the use of types, to have their information available would be useful. Since all expressions in this stage have a type calculated and associated by the compiler, even a small modification would require the re-calculation and re-assignment of the types of all the expressions in the program, since they might have been affected by the change. However, the modification of those types is not trivial and would add further complexity to our implementation. In the *Enter* stage this type information is only present in variable declarations instead of all expressions. This means that changing the type of a variable will not require the recalculation of the expression's types, simplifying the type transformation process. Additionally, we have to imperatively perform some tasks before the *AnnotationProcessing* stage, such as the identification of atomic types, forcing us to use the *Enter* stage for that particular part of our plugin. As such, choosing the *Analysis* stage for the remaining parts would effectively force us to split the knowledge base between two distinct compiler phases.

All things considered, we have decided to focus our changes in a single stage, the *Enter* stage. The more *bare bones* state of the AST allowed us to make, as well, bare bones modifications. This leaves the most complex stages to the later stages of compilation, such as type calculations. However, this means that at this point, no type information given by the compiler available for all the expressions: only variable declarations and classes have their type associated at this point. If the need to attain the type of, for example, an expression or method invocation, we have to calculate it ourselves, a non-trivial process. This represents the main drawback of making modifications during the *Enter* phase.

4.1.3 Plugin binding points

Our system will be composed of three main phases that have a strict order of dependency between them:

Atomic Type Identification collect information about all the variables and parameters annotated with `@Atomic`. In concrete, the extraction of its type. Can be performed on all compilation stages after the *Parse* stage;

Generation of Atomic Types produce the new class files of the respective atomic types previously identified. This phase can only be performed on the *Annotation Processing* stage of the compiler;

Remaining modifications composed of several phases detailed further ahead, but can be part of the same compilation stage.

Both the atomic type identification and the remaining modifications can be made at almost any stage of the compilation process, given that the parsing is already performed. However, the generation of atomic types can only be made at the *Annotation Processing*. In order to harbour this middle phase, the plugin has to be bound at three points:

- At the end of the first iteration of the *Enter* stage, atomic types are identified;
- In the following *Annotation Processing* stage, the respective atomic types' classes are generated;
- At the end of the the second iteration of the *Enter* stage, the remaining modifications are performed.

Using these binding points will allow us to achieve all our objectives, while remaining focused on a single compilation stage, *Enter*.

4.1.4 Decomposing the implementation phases

Given a general overview of the *javac* compiler, we can present the fundamental components in which our implementation is divided. They will be referred to as *phases*, not to be confused with *javac* compilation stages. In all, we can identify twelve distinct phases:

- **Atomic Type Identification**
  1. IdentifyPhase
- **Generation of Atomic Types**
  1. GeneratePhase (with RW and exclusive lock variants)
  2. GenerateGlobalPartitionsFilePhase

- **Conversion to Atomic Types**
    1. ConvertTypePhase
    2. ConvertAssignmentsPhase
    3. ConvertMethodReturnTypePhase
    4. ConvertMethodInvocationsTypePhase
- **Partitioning**
    1. BlockifyPhase
    2. PartitioningPhase
    3. InjectPartitionsPhase
- **Mapping into *ResourceGroups***
    1. MappingPhase (with RW and exclusive lock variants)
    2. ImportPhase

Each of these phases represents a self-contained set of processes, whose name is derived from the source file name that implements it. Additionally, all phases (excluding the generation of atomic types) are implemented according to a visitor pattern [PJ98]. The classes used for this purpose are both `TreeScanner` and `TreeTranslator` visitor helper classes. The first is supplied as part of the compiler API, while the later is part of the developed framework. The `TreeTranslator` extends the `TreeScanner` and takes care of the boilerplate code required to access the inner components of the compiler, allowing the programmer to access them freely. These phases will be detailed further ahead in this chapter along with their implementation strategy and possible limitations.

4.2 Atomic Type Identification and Integration

4.2.1 Identification

The very first phase of our plugin, **IdentifyPhase**, has a very simple objective: to identify the atomic types necessary for the latter phases. In other words, it has to inspect all variable declarations and functions parameters in the AST and verify if the `@Atomic` annotation is being used or not. If so, that variable is an *atomic variable*. The type of that variable, conveniently stored in a field of the AST node, is then added to a list that stores all the types of the atomic variables found.

Oddly, this procedure is fairly straightforward to perform, unlike the ones showcased further ahead in this document. Both function parameters and variable declarations (whether local, fields, or static fields) are conveniently represented by a single AST node type, `JCVariableDecl`. The type associated to each of these nodes has been previously set by the compiler, being available in the `vartype` field. The only detail here concerns arrays: the type returned will contain the brackets `[]` that have to be removed beforehand.

Unlike presented in the model, the annotation of polymorphic types is absent from this implementation. This decision was made on the ground that adding support for the polymorphic atomic types falls beyond the contributions of this thesis. Our knowledge of the compiler system was centred on the *Enter* stage, while polymorphic types are only expanded during the *Analysis* compilation stage. Implementing this feature would disperse our knowledge between these two phases, slowing down our progress and being forced to cut out other features to stay on schedule.

4.2.2 Creation

After having identified all atomic types, the annotation processing stage takes place. A small quirk, important to document here, is that the plugin system and the annotation processor are both, somehow, *sandboxed* by the compiler. Probably by launching another JVM and running the annotation processor there, but such could not be confirmed. As such, knowledge of the atomic types to be generated has to be transferred between the plugin system and the annotation processor through the use of a temporary file. Before terminating the *Enter* stage and proceeding to the *Annotation Processing* stage, the list holding all the atomic types found has to be serialized and written into the temporary file. After starting the annotation processor, the file is deserialized and the atomic types recovered.

The annotation processor, **GeneratePhaseRW**, is used to create the new atomic types. This creation, since it requires a new Java class source file, can only be done at the *Annotation Processing* stage. For each type, a new class will be created that symbolizes its atomic type. Listing 4.1 illustrates the atomic class equivalent of the `Node` class from the *linked list* benchmark. This example will be mentioned throughout this section to exhibit the transformations described.

Two distinct alternatives existed for generating the classes for the atomic types. Extending the base type class or *cloning* the behaviour of the base type class into the new atomic type class. Copying of the old class' behaviour into the new class, without extending it, would ensure that the implementation remain strictly faithful to our model.

However, the cloning of AST nodes containing the base class behaviour raises several implementation issues that would eventually remove us from our main objective. On the other hand, resorting to the `extends` clause implies the inability to create an atomic type if the base class has been declared as `final` (such as `String`) or the primitive type classes (such as `Integer`). Since the system only works with objects, primitives (`int`, `long`, etc.) are also unqualifiable for atomic types.

Despite these limitations, the choice of extending the base class was made. Nevertheless, we believe that with the deeper understanding of the compiler developed during the latter phases of this work, the excluded alternative can be regarded as a possibility

```
public class AtomicNode extends Node implements RWResource {
    protected final ReentrantReadWriteLock __lock = new
        ReentrantReadWriteLock();
    private final Lock __writeLock = __lock.writeLock();
    private final Lock __readLock = __lock.readLock();

    public AtomicNode(int arg0) {
        super(arg0);
    }

    public AtomicNode(int arg0, AtomicNode arg1) {
        super(arg0, arg1);
    }

    // Implementation of the Resource Interface
    public int compareTo(Resource o) {
        return this.hashCode() - o.hashCode();
    }

    public void lock() {
        __writeLock.lock();
    }

    public void unlock() {
        try { __writeLock.unlock(); }
        catch (Exception e) {}
    }

    public boolean tryLock() {
        return __writeLock.tryLock();
    }

    public void lockRead() {
        __readLock.lock();
    }

    public void unlockRead() {
        try { __readLock.unlock(); }
        catch (Exception e) {}
    }

    public boolean tryLockRead() {
        return __readLock.tryLock();
    }
}
```

Listing 4.1: Example of an atomic type class generated for the `Node` class of the *linked list* benchmark.

```
public class GlobalPartitionsHolderClass {

    public GlobalPartitionsHolderClass() {
        super();
    }

    //Where partitions will be inserted in the future
}
```

Listing 4.2: Empty `GlobalPartitionsHolderClass` created to hold global partitions.

for future work.

The main side-effect of our choice is that the extension of the base class allows the assignment of an atomic type to its non-atomic equivalent due to inheritance, i.e. the atomic type is a subtype of the non-atomic type, a behaviour that goes against our model. In an attempt to eliminate this behaviour, our system detects the occurrence of these assignments. A type rule was artificially injected in order to strongly enforce the propagation of atomic types. Scenarios where the assignment of an atomic variable to a non-atomic variable are verified, a custom type error issued by the compiler. This allows us to partially avoid breaking the premise presented in Definition 5. However, this rule can still be broken if the programmer uses a supertype of the atomic type to convert it to a non-atomic type. For this reason, figuring how to implement the atomic types flawlessly should be the priority in future work.

Aside from extending the base class, the new atomic type class also has to support a lock-based interface (`Resource`), that allows the *ResourceGroup* API to interact with an atomic resource. Thus, we add two objects to the new class representing a read lock and write lock (lines 3 and 4). Then, the respective methods required from the *ResourceGroup* API are injected and implemented using delegation to the aforementioned two lock objects (lines 15, 19, 23, 28, 32, 36, and 41).

Next, we need to inject into the new class the constructors from the base class and implement them through delegation (lines 6 and 10). For this, we find the symbol containing the base class and extract all the constructors in it. This is done by using the `JavacElements` helper class in the compiler.

The package of the new atomic type class will be the same as the base class. The name of the new atomic class is composed of the old name prefixed with a common prefix attributed to atomic types, `Atomic`.

4.2.3 Creation of a class for global partitions

After all atomic type classes have been generated, the process proceeds to a rather simple phase entitled **GenerateGlobalPartitionsFilePhase**. A single, empty, placeholder class, illustrated in Listing 4.2, is generated to store the global partitions required in the future. This allows us to keep all the global partitions in a single class, ready for importing to the classes that require it.

4.3 Converting Annotated Variables into Atomic Variables

This section will encompass several phases whose final objective is to convert the type of atomically annotated variables into the respective atomic type, and to perform the necessary program transformations in order to re-establish the type coherency eventually disrupted by the preceding conversion. Additionally, the equations mentioned throughout this section are illustrated in Figure 4.2. In it, `T` represents a type, x a variable, ϕ our transformation function, f a function, e an expression, and i a statement. $\tilde{e}$ and $\tilde{\imath}$ represent a sequence of expressions and statements, respectively.

$$\phi(\texttt{@Atomic T}\ x) \Rightarrow \texttt{AtomicT}\ x \tag{4.1}$$

$$\phi(\texttt{@Atomic T[]}\ x) \Rightarrow \texttt{AtomicT[]}\ x \tag{4.2}$$

$$\phi(\texttt{new T}[\tilde{e}]) \Rightarrow \texttt{new AtomicT}[\phi(\tilde{e})] \qquad \text{if assigned to a variable of type } \texttt{AtomicT} \tag{4.3}$$

$$\phi(\texttt{new T}(\tilde{e})) \Rightarrow \texttt{new AtomicT}(\phi(\tilde{e})) \qquad \text{if assigned to a variable of type } \texttt{AtomicT} \tag{4.4}$$

$$\phi(\texttt{T}\ f(\tilde{x})\ \{\tilde{\imath}\}) \Rightarrow \texttt{AtomicT}\ f(\phi(\tilde{x}))\ \{\phi(\tilde{\imath})\} \qquad \text{if returnType}(\tilde{\imath}) \text{ is } \texttt{AtomicT} \tag{4.5}$$

$$\phi(\alpha) \Rightarrow \alpha \tag{4.6}$$

Figure 4.2: Type transformations performed.

4.3.1 Converting to atomic types

The initial type conversion, **ConvertTypePhase**, is fairly straightforward: convert type of the variables and function parameters annotated with `@Atomic` into its correspondent atomic type (Equation (4.1)). As such, for each annotated variable declaration found, we replace the *vartype* field of each `JCVariableDecl` with the new correspondent atomic type. Respectively, array types also have their types converted if annotated (Equation (4.2)).

In the linked list example illustrated in Listing 4.3, it is possible to observe the conversion of types that occurred in the `root` (line 2) and both local atomic variables `prev` and `next` (line 6 and 7, respectively).

4.3.2 Fixing atomic type constructors

At this point, while program has the atomic variables with the correct type, its assignments are incorrect. In assignments, before we had a non-atomic type variable on the left-hand side and a non-atomic type variable or constructor on the right-hand side. However, now it is possible to have an atomic type variable on the left-hand side and the old non-atomic type on the right-side. Fixing this right-side of the assignments will be the objective of the following phase, **ConvertAssignmentsPhase**.

It starts by traversing the AST, similarly to other phases, and checking all assignments. Three scenarios can be identified in the assignments, since each scenario requires a slightly different approach, but they have the same foundations. First, we calculate

```
class LinkedList {
 AtomicNode root;

 boolean contain(int value) {
  boolean result;
  AtomicNode prev = root;
  AtomicNode next = prev.getNext();
  int v;
  while ((v=next.getValue()) < value) {
   prev = next;
   next = prev.getNext();
  }
  result = (v == value);
  return result;
 }
}
```

Listing 4.3: Conversion of atomic types applied to the linked list example.

the types of the left and right-hand side expressions. Second, if the left-hand side corresponds to an atomic type and the right-hand side to a non-atomic type, we are in the presence of a type incoherency introduced by the previous stages. In the other cases, we do not do anything because it is not our responsibility. Third, we check if the right-hand side's type is the base type of the atomic type represented in the left-hand side. If so, we have to convert the right-hand side to match its atomic correspondent. Therefore, and for each scenario:

A variable is being assigned a new object (new class constructor) the constructor must be converted into the correspondent atomic type (Equation (4.4)).

An array position is being assigned a new object again, the constructor must be converted into the correspondent atomic type. (Equation (4.4))

An array is being assigned a new array the constructor of the array must be transformed into the array of the correspondent atomic type. (Equation (4.3))

4.3.3 Fixing method return types

The next phase, **ConvertMethodReturnTypePhase** fixes another kind of incoherency. Namely, methods that have their return type as a base type when in fact, the returned variable is an atomic type. As such, we need to convert the return type of that method into the correspondent atomic type.

During the traversal of the AST, for each method found, the type of the variable returned is calculated. If it is an atomic type and the return type of the method represents its base type, conversion is required. The conversion is performed in-place, by replacing the `restype` class identifier (that represents the type returned) of the `JCMethodDecl` AST node (that represents the method in question) with the new identifier that represents the atomic type.

Again, in the example illustrated in Listing 4.4, the `getNext` method has its return type

```
1 public class Node {
2     private AtomicNode m_next;
3 
4     (...)
5 
6     public AtomicNode getNext() {
7         return m_next;
8     }
9 }
```

Listing 4.4: Conversion of the return type of the `getNext` method of the `Node` class from the linked list example.

```
public void meth() {
    @Atomic T var = new T(...);
    go(var);
}

public void go(@Atomic T param) {
    ...
}
```

Listing 4.5: Example of the superfluous variable declaration avoided by the ConvertMethodInvocationsTypePhase phase.

converted from `Node` to `AtomicNode` (line 7). After the type conversion, the expression being returned no longer represents a `Node`, but rather its atomic counterpart.

4.3.4 Inferring the single use of atomic type constructors as arguments

The following and final phase shares a distinct motivation. Rather than ruled by the strict enforcement of the model and the type coherency in the program, it is guided by more practical reasons. Its purpose is to avoid a distinct variable declaration for all objects created even if they are only used once, in a method invocation. Take the example illustrated in Listing 4.5. The normal use of the method invocation would be `go(new T(...))`, because it is only referenced once and it is only used for the method invocation. But such would not be allowed, since the constructor is not on the right-hand side of an explicit assignment, so it would not be converted into its atomic type equivalent. The scenario previously described in Section 4.3.2 represents the same transformations, albeit in an explicit setting. Here, the transformations are implicit since the newly created objects are not part of direct assignments, but used as arguments to the target function.

This would effectively force the programmer to declare the variable. This phase does precisely that. Fix all method invocations in the AST where a constructor of a base type is used as an argument and the respective parameter is the atomic type.

Note that, from this phase onward, the type of expressions might have to be calculated. However, calculating the type of an expression is not as simple as the variable case, since it must be derived from the expression's components, which in turn may

```
1  public class Node {
2    private AtomicNode m_next;
3
4    (...)
5
6    public boolean add(int value) {
7      (...)
8
9      if (result) {
10       prev.setNext(new Node(value,next));
11     }
12
13     return result;
14   }
15
16   public AtomicGENNode setNext(AtomicNode next) {
17     m_next = next;
18   }
```

Listing 4.6: Conversion of the constructor type used in the `setNext` method of the linked list example.

themselves be expressions. This process is currently very complex, and not yet perfect. There are many corner cases and distinct scenarios that can occur during the calculation of the type of an expression that are not documented. Discussion about the strategy and limitations of the type inference system developed would deserve a section on its own. However, due to not being directly correlated with the implementation of the plugin, it will not be further discussed. While the type inference system developed worked as far as it was tested, it is surely the weakest link in this implementation and a priority for future work.

Having said that, the first step of this phase, **ConvertMethodInvocationsTypePhase**, is to calculate the type of constructor used as an argument and the type of the expression over which this function is invoked. The type of the expression that precedes the function invocation is needed so we can pinpoint to which class the function declaration belongs to. Accordingly, the class that owns the declaration of that function is attained so the type of the correspondent parameter of that function can be retrieved. Now, in the presence of a constructor of a non-atomic type being used as an argument to a function invocation whose respective parameter is the atomic type correspondent, the constructor is transformed into its respective atomic type.

This phase effectively avoids the isolated declaration of an atomic local variable (with constructor) just to pass it posteriorly into a method invocation. This is illustrated in Listing 4.6, where the `previous.setNext(node)` method can be invoked without declaring explicitly the new atomic node beforehand (line 10).

4.3.5 Homogenization of the abstract syntax tree

One of the big problems brought by the variation allowed in the AST is the various combinations of nodes possible to reach the same end result. The biggest proponent of this aspect are blocks, whose AST node type is `JCBlock`. They delimit a sequence of instructions and are used mostly in methods, classes, and control flow statements. However, in many cases, such as in the use of control flow statements, they are not mandatory. As such, it is possible to use a `for` loop with only one statement following it without the using a block. This statement is not an `JCBlock`, but a `JCExpressionStatement`. This brings two types of problems. First, additional complexity is required when iterating over the syntax tree. To find the body of control flow statements we have to check both scenarios: either a solo `JCExpressionStatement` without a block or a `JCBlock` that contains inside multiple statements. Second, any modification of the AST tree that requires adding additional statements would have to firstly create the block if missing, adding unneeded and unrelated complexity to the algorithm.

This phase, **BlockifyPhase**, effectively erases both type of problems by replacing all potentially missing blocks with blocks. It checks for missing bodies in most control flow statements. Both `if-else` statements, `foreach` and regular `for` loops are *blockified*. Its implementation is rather simple. If the expected control flow statement's body is not a `JCBlock` but a sole `JCExpressionStatement`, it creates a new `JCBlock`, copies the previous `JCExpressionStatement` to the new block and unwraps the `JCExpression` contained in the `JCExpressionStament`. Finally, the new block is assigned to the control flow statements AST node.

4.4 Deadlock-free Lock Inference

In this section, we detail the lock inference performed that guarantees deadlock-freedom in our system. This lock inference diverges from previous approaches in most code-centric state of the art because those works have to infer which locks to associate with each memory position. On the other hand, in our approach, we already have the locks associated with each resource accessible through an atomic variable. In order to guarantee deadlock-freedom, we use the *partitioner*, a tool introduced further ahead that detect deadlock-prone situations in a program, allowing us to prevent them from occurring. At this stage, the program is already deadlock-free. However, a considerable amount of locks in the current program might not be required, introducing unnecessary overhead. As such, two additional steps are performed to reduce the number of locks that have to be acquired at runtime, improving its performance. This optimization phase entails two steps: the elimination of *dominated* and *redundant* locks.

4.4.1 Partitioning

The partitioner, developed as part of the previous *ResourceGroups* project [PD12] developed by Professor Hervé Paulino and student Nuno Delgado, is an external tool that requires knowledge of the AST of the program to detect which resources acquisitions can lead to cyclic dependencies, and hence, a deadlock. In specific, the classes present in the program and their respective methods, the declaration of atomic variables, where an atomic variable is referenced, and finally, the dependencies between methods are required by this tool. The later is needed so the partitioner can build a call-graph of the program. As such, the goal of this step is to traverse the AST and transmit that information about the AST to the partitioner, through the use of the partitioner's API, illustrated at Table 4.1. This API features methods oriented to the visitor pattern. The visitor only has to inform the partitioner of its current position (using either the `newClass` or `newMethod` methods) for the partitioner to progressively build an internal representation of the AST as the visitor traverses the tree, while informing the partitioner about other relevant tree nodes.

```
void newClass(String clazz)
void addPublicField(String resourceIdentifier)
void addPrivateField(String resourceIdentifier)
void newPrivateMethod(String methodod, int arity)
void newAcquisition()
void newMethod(String methodod)
void addPublicResourceType(String type)
void addPrivateResourceType(String type)
void addLocalResource(String resourceIdentifier)
void addMethodDependency(String clazz, String method, int arity)
PartitionList computeLocalPartitions(String clazz)
PartitionList computeGlobalPartition()
```

Table 4.1: API provided by the partitioner.

When the declaration of an atomic variables is found by the visitor, the corresponding method must be invoked. For class fields, the `addPublicField` or `addPrivateField` methods is used, depending if the field in question is, respectively, public or private to the current class. For local atomic variables, `addLocalResource` is used if declared on the spot (using `new` constructor), otherwise[6] using `addPrivateResourceType` with its atomic type. Given that, at compilation time, it is not always possible to infer to which resource an atomic variable will point to, the minimum known denominator must be used, which is its type. This last case occurs throughout the linked list example, such as in `@Atomic var = previous.getNext()`, where `addPrivateResourceType` is used with the `AtomicNode` type. For atomic parameters, `addPublicResourceType` method is also used with the type of the parameter. Finally, for each expression encountered with a method invocation,

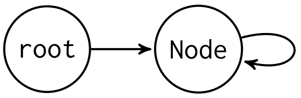
---

[6] Namely, if it derives from the invocation of a method that returns an atomic type or is an assignment to a previously declared local variable.

`addMethodDependency`) indicating the method invoked and its respective class. In the end, the partitioner is able to use the information collected to build a simpler image of the AST containing all the information required to fulfil its purpose.

The internal representation of the program built by the partitioner is composed by four distinct components: *DependencyChainSets*, *DependencyChains*, *Groups* and finally *Locks*. The program evaluated is composed by as many *DependencyChainSets* as the number of classes present in it. As such, a **DependencyChainSet** correspond to a class and encompasses all the *DependencyChains* in it. In turn, each **DependencyChain** represents a possible path of execution of a method in the current class, composed by a list of *Groups* with a specific order representing the order of group acquisitions in chained method invocations. A single **Group** represents a set of *Locks* which must be acquired atomically. Finally, a **Lock** represents an atomic variable in the source code.

A set of *DependencyChain* may be interpreted as a graph, whose vertex set represents all group acquisitions and the edge set is given by the union of the Cartesian products of every pair of successive group acquisitions. In the example of the *linked list* implementation illustrated in Listing A.2, the graph representing the *DependencyChains* of the methods of class `Node` would be:



In this graph, strongly connected components (SCC) correspond to cyclic dependencies between resource acquisitions, symbolizing deadlock-prone situations.

The partitioner then applies Tarjan's algorithm [APT79] to identify the SCCs. Each of the identified SCCs is associated with a partition, an artificial resource created by the partitioner that will force the serialization of the methods that incur the SCCs. This prevents the acquisition of resources in unordered fashion by multiple threads and consequently, prevents deadlocks. Again, in the case of the linked list example illustrated in Listing A.2, a partition would be associated with the three main methods: `add`, `remove` and `contains`, since they access a type included in the partition, `AtomicNode`. Effectively, this partition forces the serialization of the execution of those methods, as illustrated in Figure 4.3.

The partitions identified can be part of two distinct scopes: local and global partitions. Local partitions are local to each object of a class since the SCC associated only includes resources local to that class. Global partitions, on the other hand, are shared across all objects of the program because the SCC associated includes resources from multiple classes.

The end result given by the partitioner to our plugin at this point is the set of locks that should be acquired by each method. However, this preliminary result will be trimmed down by the lock coalescing performed. These were developed in the context of our work.

4.4.2 Lock coalescing

Firstly, in order to **eliminate dominated locks**, a relation of *dominance* between locks has to be established. While this concept is similar to the one presented in [HFP06], the latter does not have the notion of groups, only of standalone locks.

Definition 6. *(Dominance) A lock l_1 is said to dominate lock l_2 when all occurrences of l_2 are either preceded by l_1 or acquired at the same time (i.e. occur in the same Group) as l_1. By analogy, l_2 is said to be dominated by l_1.*

As such, when a lock l dominates lock l', the acquisition of l' is not required due to it being *contained* by the previous acquisition of lock l. Thus, a dominated lock can be safely removed, eliminating the overhead associated with its (superfluous) acquisition. However, since resources can be aliased to other resources, problems regarding aliasing of locks could arise. To avoid these, we can only eliminate a lock if its dominant lock is deemed *concrete*. A concrete lock is a lock from a variable whose pointer is never altered by another thread during execution. Furthermore, since initially the acquisition of dominated locks is guaranteed to be preceded by the acquisition of its dominant lock, the elimination of dominated locks will not result in a loss of parallelism.

In order to achieve maximum performance, our work will eliminate all dominated locks. The implementation of the algorithm that performs this task, illustrated in Listing 4.7, is fairly straightforward. First, we attain the ordered list of all locks in the program supplied (`allChains`) using the method `flatMapLocks(allChains)` (line 4). The locks should be ordered according to a deterministic criterion. Our implementation uses the lexicographic order of the locks. Then, for all pairs (l_1, l_2) of distinct locks l_1 and l_2, we check if l_1 dominates l_2 using method `dominates(l1,l2,allChains)` (line 12). This method returns `true` when lock l_1 dominates lock l_2, i.e. when l_1 precedes or occurs in the same group as l_2 in all chains where l_2 occurs. If so, and if l_2 represents a concrete

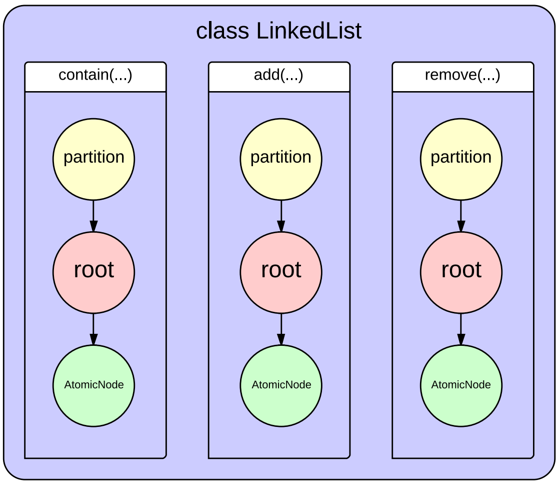


Figure 4.3: Order of resource acquisition after the attribution of a partition to the *linked list* example.

lock, we eliminate all occurrences of l_2, the dominated lock, from the current program (line 15). Accordingly, l_2 is also removed from the list of all locks. When an elimination is performed, the algorithm restarts (line 19). This process is repeated until the algorithm converges, when an iteration of the algorithm does not eliminate any lock, i.e. no dominated locks remain in the program.

To exemplify the lock inference process, let us focus on the *hash set* benchmark, illustrated in Listing A.1, with just the `add`, `findIndex` and `rehash` methods for simplicity. In the partitioner, three chains exist for the three respective methods, as illustrated in Figure 4.4.

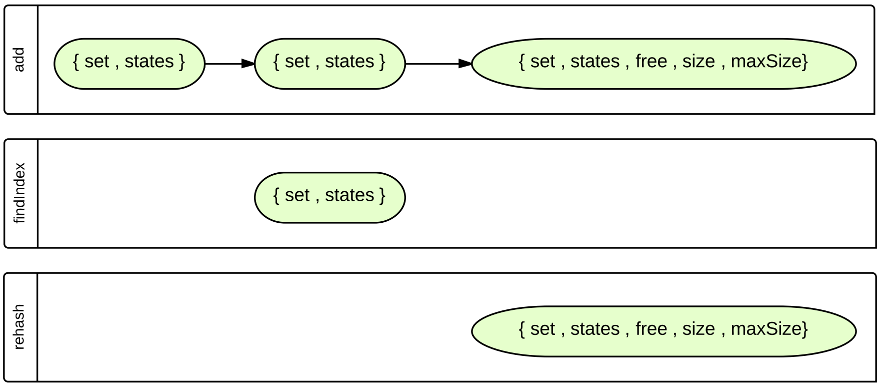

add
{ set , states }
{ set , states }
{ set , states , free , size , maxSize}
findIndex
{ set , states }
rehash
{ set , states , free , size , maxSize}

Figure 4.4: Graph representation of dependency chains in a simplified *hash set* containing only the `add`, `findIndex` and `rehash` methods.

The `add` method invokes two methods, first `findIndex` then `rehash`. In the `add` method chain, the first group belongs to the `add` method itself, since it accesses both the `set` and the `states` atomic variables. The second group belongs to the `findIndex` invocation and the third to the `rehash` invocation. By itself, the `rehash` method accesses five atomic variables: `set`, `states`, `free`, `size`, and `max`. The `findIndex` method chain only accesses two atomic variables, `set`, and `states`.

Before the start of the algorithm, we have the following chains:

```
Chain add:        -> (set,states)->(set,states)->(free,max,set,size,states)
Chain findIndex:  -> (set,states)
Chain rehash:     -> (free,max,set,size,states)
```

The algorithm then starts testing all possible pairs of distinct locks in the program, and checking if one dominates the other. In our implementation, the first pair detected in which a lock dominates the other is `(free,max)`. In this case, `free` occurs in two chains: chain `add` and chain `rehash`. In both chains, `free` occurs in the same group as `max`. Since `free` either precedes or occurs in the same group in the chains that contain `max`, `free` dominates `max`. We proceed with the elimination of the dominated lock, `max`.

```java

public static void lockCoalescing(Set<DependencyChainSet> allChains) {
  //get all locks of the program
  List<Lock> allLocksInProgram = flatMapLocks(allChains) ;
  //remove one lock at a time until it converges
  repeat:while(true) {
    //for each pair of distinct locks
    for(Lock l1 : allLocksInProgram) {
      for(Lock l2 : allLocksInProgram) {
        if(!l1.equals(l2) && l1.isConcrete()) {
          //if l1 dominates all ocurrences of l2
          if(dominates(l1,l2, allChains)) {
            //remove dominated lock l2 from the program
            Lock dominated = l2 ;
            removeAllFromAllChains(dominated,allChains);
            //...and from our list of locks
            allLocksInProgram.remove(dominated) ;
            //finally, repeat algorithm since we eliminated a lock
            continue repeat;
          }
        }
      }
    }
    return ; //no removal done, converged
  }
}

private static boolean dominates(Lock l1, Lock l2, Set<DependencyChainSet>
    allChains) {
  for(DependencyChainSet css : allChains) {
    for(DependencyChain c : css) {
      if(c.containsLock(l2)) {
        for(Group g : c) {
          if(g.contains(l1))
            break ; //precedes OR in same group in this chain
          if(g.contains(l2))
            return false ; //l2 found before l1, so l1 doesnt precede
        }
      }
    }
  }
  //lock l1 precedes (or occurs in same group) as lock l2 in our program
  return true;
}

class DependencyChainSet extends Set<DependencyChain> {...}
class DependencyChain extends Set<Group> {...}
class Group extends Set<Lock> {...}
class Lock {...}
```

Listing 4.7: Lock inference algorithm.

```
Chain add:        -> (set,states)->(set,states)->(free,set,size,states)
Chain findIndex:  -> (set,states)
Chain rehash:     -> (free,set,size,states)
```

In the next iteration, the first pair where a lock dominates the other is the `(free,size)`. Since `free` and `size` always occur in the same group together, `free` dominates `size`. The same could be said for the pair `(size,free)`, where `size` also dominates `free`. Thus, regarding the correction of the algorithm, `free` could indeed be eliminated instead of `size`. However, following the ordering of pairs tested in our algorithm, the `(free,size)` is tested before the `(size,free)`. As such, `size` is eliminated.

```
Chain add:        -> (set,states)->(set,states)->(free,set,states)
Chain findIndex:  -> (set,states)
Chain rehash:     -> (free,set,states)
```

In the next iteration, the pair found is `(set,free)`. Here, `set` dominates `free`. Accordingly, `free` is removed.

```
Chain add:        -> (set,states)->(set,states)->(set,states)
Chain findIndex:  -> (set,states)
Chain rehash:     -> (set,states)
```

In the next iteration, only two locks are left: `set` and `states`. The first pair to be checked is `(set,states)`. Since they always occur in the same group, `set` dominates `states`. Accordingly, the lock `states` is eliminated from all the chains where it occurs.

```
Chain add:        -> (set)->(set)->(set)
Chain findIndex:  -> (set)
Chain rehash:     -> (set)
```

The algorithm then converges because in the next iteration no lock is dominated by another lock. No lock can further be removed and the algorithm terminates.

When a partition is introduced, such as the linked list example illustrated in Figure 4.3, the algorithm behaves the same way. In this example, all three methods (`add`, `remove`, and `contains`) share similar chains:

```
Chain add:        -> (partition)->(root)->(AtomicNode)->(AtomicNode)
Chain remove:     -> (partition)->(root)->(AtomicNode)->(AtomicNode)
Chain contains:   -> (partition)->(root)->(AtomicNode)->(AtomicNode)
```

Here, the resources represented by `AtomicNode` cannot be acquired at the start of the method, when the partition and `root` are acquired. As such, `AtomicNode` is part of a distinct group.

In the first iteration, the pair `(partition,AtomicNode)` is tested. Here, since `partition` always precedes `AtomicNode` in the chains where `AtomicNode` occurs, `partition` dominates `AtomicNode`. As such, the locks `AtomicNode` are eliminated.

```
Chain add:        -> (partition)->(root)->()->()
Chain remove:     -> (partition)->(root)->()->()
Chain contains:   -> (partition)->(root)->()->()
```

In the second iteration, the pair `(partition,root)` is tested. Again, since `partition` always precedes `root`, `partition` dominates `root`. Accordingly, `root` is eliminated from the program.

```
Chain add:        -> (partition)->()->()->()
Chain remove:     -> (partition)->()->()->()
Chain contains:   -> (partition)->()->()->()
```

In the final iteration, no dominated locks are found. Thereby, the algorithm terminates.

4.4.3 Lock removal

Afterwards, the **elimination of redundant locks** takes place. The main objective of this step is improving performance by eliminating the acquisition of locks that, through a call-graph analysis, are guaranteed to be already previously acquired. For each method, we check its callers.

Given a method m and a lock l, if all the callers of m acquire l then the acquisition of l in m is omitted. A literal implementation of this algorithm would only eliminate locks acquired in consecutive methods. In order to *bubble down* the locks previously removed, when checking the presence of lock in the callers of a method, we also check if that caller already had that lock removed. If it did, it was already acquired by its callees. Thus, we can also remove it from the present method. This procedure is then repeated over all methods until all removals are bubbled down.

In the previous *hash set* example, after the elimination of dominated locks, some redundant locks are still present. Both `findIndex` and `rehash` methods are only invoked from within the `add` method. As such, the `set` lock acquisition in both `findIndex` and `rehash` are redundant since it is guaranteed to be previously acquired in the `add` method. As such, the locks that are acquired by the `findIndex` and `rehash` methods can be eliminated from the chains. In the end, only the `set` lock has to be acquired in the `add` method.

```
Chain add:        -> (set)->()->()
Chain findIndex:  -> ()
Chain rehash:     -> ()
```

4.5 Mapping onto the *ResourceGroups* API

This section will detail the phase that will acquire the resources identified by the partitioner using the *ResourceGroups* API in each method. Before that, another simplification of the AST to ease this phase will be presented.

```
public boolean contains(int key) {
        AtomicRBNode node = getRoot();
        while (node != sentinelNode) {
            if (key == node.getValue()) {
                boolean returnExpression = true;
                return returnExpression;
            } else {
                if (key < node.getValue()) {
                    node = node.getLeft();
                } else {
                    node = node.getRight();
                }
            }
        }
        boolean returnExpression = false;
        return returnExpression;
    }
```

Listing 4.8: Example of the return expression isolation in the `contain` method of the `RBTree` benchmark.

4.5.1 Isolating return expressions

One of the fundamental conditions present in our model is that a function releases all the locks it has acquired. However, in situations where the `return` expression still accesses atomic variables, release is not possible since the return marks the last statement executed in the method. As such, one has to isolate those expressions that are returned into a new variable, that is in turn returned. This job is done by the **IsolateReturnExpressionsPhase**. At this point, and due to the previous *blockification* of the AST, all control flow statements are guaranteed to use a `JCBlock`, representing the brackets at syntax level. Since all `return` statements are guaranteed to be inside a `JCBlock`, searching all blocks will uncover all `return` statements in the AST. For each `return` statement found, if it does not return a single variable, the later is relocated into a new assignment that is injected before the `return` statement. Next, the expression in the return statement is replaced by the reference to the newly created variable.

This transformation preserves the behaviour of the method, while enabling the subsequent phases to freely insert statements before any return, without worrying about the presence of atomic variable references in the return expression, allowing the release of the resources acquired in that method before the `return` statement. The final result is exemplified in Listing 4.8, where both `return` statements in the `if` block and in the method body got isolated to allow the inject of additional statements between them.

4.5.2 Injecting partitions

While the partitions are already defined by the partitioner, they only represent artificial resources that only exist inside the partitioner. Thus, they have to be inserted in the actual program. This task is performed by the **InjectPartitionsPhase**. The local partitions have

```
public class GlobalPartitionsHolderClass {

    public GlobalPartitionsHolderClass() {
        super();
    }

    public static Partition GLOBAL_partition_00 = new Partition();
    public static Partition GLOBAL_partition_01 = new Partition();
}
```

Listing 4.9: `GlobalPartitionsHolderClass` with two global partition injected.

```
public class IntSetLinkedList implements IntSet {

    private AtomicNode m_first = new AtomicNode(0);
    private Partition LOCAL_partition_00 = new Partition();

    (...)
}
```

Listing 4.10: `LinkedList` class with a local partition injected.

to be inserted in the scope of the selected classes and global partitions will have to be inserted in the special class created to store the global partitions, `GlobalPartitionsHolderClass` (Section 4.2.3).

For it, we need to inject new fields inside those classes that represent the partition. However, adding a new field to a pre-existing class in the AST is not trivial. One would think that adding the respective variable declaration inside the AST of the class would be sufficient. The code will compile, and the field will be apparently generated. But at runtime, the JVM would crash. The reason for this is that the offsets from the fields in a class are calculated after the *Parse* phase. Injecting a variable declaration in the class AST would not automatically update the offsets, leading to corrupted bytecode. Those offsets have to be updated manually after the injection of a new field (a variable declaration) in classes.[7]

The variable declarations are created to hold an object of type *Partition*. This object inherits from *Resource*, and as such, supports lock based operations similar to the atomic type objects created. Illustrated in Listing 4.9 is the partition holder class with two global partitions injected and in Listing 4.10 is an example of a class that had a local partition allocated by the partitioner and thus, injected into it. These partitions will be used by the next phases, as explained in the this section.

---

[7] This search for the correct injection of new fields in pre-existing classes in a *javac* plugin lead to the creation of a small proof-of-concept. A simple *javac* plugin that offers support for persistent local variables. `https://github.com/metabrain/java8-plugin-persitent-local-vars`.

4.5.3 Mapping

This phase, the **MappingPhase**, is where all concurrency control will be effectively enforced after the preparation performed in previous phases. Firstly, it is important to state two variants of this phase exist: one that makes use of exclusive locks and another that uses read-write locks. Only the read-write implementation will be detailed due to it being more interesting, complex and, as explained ahead in Section 5.2, allowing a strictly better performance.

All the reasoning in this phase is done at method-level, since it represents our basic unit of work. Due to the complexity of this phase, it will be further split into three distinct parts: attaining the dominant locks from the partitioner, locating the AST nodes representing locks and finally the actual mapping. The mapper itself, in turn, is also composed of two distinct modes: single lock mapper or multi-lock mapper.

4.5.3.1 Filtering resources to acquire from the atomic variables

At this point, a method might have several types of atomic variables. They can be class fields (partition or a normal variable), method parameters or local variables. As such, the first step is to query the partitioner about what locks should be acquired in this method. This is done by invoking the `m.getDominators()` method in the partitioner, where `m` represents the current method, from which we obtain a list of the dominant locks. After collecting in a list all AST nodes of the atomic variables referenced in the current method, the local partitions in the current class and all the global partitions, we remove those that are not present in the list of dominant locks for this method. Now that we have the AST nodes of the dominant locks, we have to choose which mapper to execute.

4.5.3.2 Multi-resource Mapper

The functioning of the mapper (read-write lock version) will be explained in parallel with the example illustrated in Listing 4.11. The example represents the *hash set* benchmark used in Section 5.2. For this example, the only resource that must be acquired according to the compiler is `set`.

The first step is inject the creation of the `ResourceGroup` object at the beginning of the method (line 2). This object will be used according to the *ResourceGroups* API, illustrated in Table 2.1, in order to add or remove resources, lock or unlock the group, throughout the method. Following, if a partition is part of the set of dominant locks to acquire, we set as the current partition by injecting, immediately after the declaration of the `ResourceGroup`, a statement with `rg.setPartition(`p`)`, where p represents the partition variable.

Next, the acquisition of the group should be made before the first access to an atomic variable. This means that scope of atomicity in a function has an upper boundary delimited by the first statement that accesses an atomic variable. In our example, the first access occurs at line 5. Accordingly, the `lock` of the resource group is injected at line 4, before the first access.

```
1  private int index(int value) {
2      RWResourceGroup rg = new RWResourceGroup();
3      rg.addRead(set);
4      rg.lock();
5      int length = states.value.length;
6      int hash = (value * 31) & 2147483647;
7      int index = hash % length;
8      int probe;
9      if (states.value[index] != FREE && (states.value[index] == REMOVED ||
           set.value[index] != value)) {
10         probe = 1 + (hash % (length - 2));
11         do {
12             index -= probe;
13             if (index < 0) {
14                 index += length;
15             }
16         }
17         while (states.value[index] != FREE && (states.value[index] ==
               REMOVED || set.value[index] != value));
18     }
19     int returnExpression = states.value[index] == FREE ? -1 : index;
20     rg.unlock();
21     return returnExpression;
22 }
```

Listing 4.11: Final result of the multi-resource mapper for the `index` method of the *hash set* benchmark.

The following steps entail the initial addition of the atomic variables used to the `RWResourceGroup` object. In this situation we can identify two groups of atomic variables: those that are known at the beginning of the method and those that are not. The first group contains both class fields and parameters while the second represents local variables in the method.

Accordingly, the additions of the atomic variables that are known at the start are also acquired at the start, immediately after the creation of the `ResourceGroups` object. In our example, `set` has to be acquired as previously stated, and as such, is acquired at the beginning of the method (line 3).

On the other hand, the variables that are not known at the start can only be added to the group when they are declared. As such, their addition to the group is only injected after their initial declaration.

The aliasing of atomic variables is also extremely relevant for the mapper. Each time an atomic variable is aliased, the resource referenced is potentially changed, requiring an acquisition. As such, the mapper injects the acquisition of the new resource in the statement following its aliasing. Basically, the mapper treats an alias of an atomic variable as the declaration of a local variable.

The read-write lock version of the mapper also allows the use of read resources, as observed in line 3 of the *hash set* example. However, its application must respect some rules.

Since the Java read-write locks used, `ReentrantReadWriteLock`, do not support promotion, we cannot add a resource as read if a nested function will perform a write access upon that same resource.

To discover if an atomic variable can be acquired in read mode, two conditions must be verified. First, the atomic variable is used solely in *read accesses* in the context of the current method.

Here, our definition of *read access* is that, basically, the variable is only used as a reference. This entails no aliasing and no invocations of methods on the referenced object. In the case of atomic variable arrays, accessing a position of an array is considered a read access, but creating or assigning another array is classified as a write access. Second, the methods invocable from the current method and their respective descendent also only use the atomic variable, if present, in read accesses. If any of these conditions is broken, the system defaults to acquire the variable in write mode.

The inference of read and write accesses might be further relaxed, but to do so it would be necessary to perform a deeper analysis over the whole call-graph. Due to time constrains, we think that the current read/write set detection was sufficient to demonstrate the application of readers-writer locks to our system.

At last, all resources acquired throughout the method have to be released before it terminates, or else resources would remain eternally acquired in the system. Therefore, the `rg.unlock()` statement has to be injected at the end of all possible exit points of the function. For this, the mapper traverses the AST of the method, exploring all possible paths of execution and injecting this statement before its end. The end of the paths of execution are represented by either `return` statements, or in its absence, the end of the method body.

Since the release of the resources acquired throughout the method is always done at its exit points, two conclusions can be drawn. First, the end of a method represents the lower boundary of atomicity in a method. This boundary could, arguably, be further narrowed to the last statement that accesses an atomic variable, representing future work that could be done to further optimize the performance of the system.

However, this implementation does not currently support the release of the locks in case of an exception. The solution would be to, in every method, surround the contents of the body after the declaration of the `ResourceGroup` object with a `try-catch`. The `try` section would delimit the entire code of the method, while the `catch` block would include two statements: one to release the resources acquired and another to throw the exception caught upwards on the call-stack. That way, an exception would not leave pending locks.

Note that currently 2PL is not enforced, given that a top-level method does not acquire resources that are reachable by a nested method invocation but not in its actual body. These acquisitions (and paired releases) are delegated on the invoked method. Accordingly, during a method's execution locks may be released before others are acquired.

4.5.3.3 Single Resource Mapper

The single resource mapper, *SingleResourceMapMethod*, is an optimization for the scenario where only one atomic variable needs acquisition in a method. Instead of creating a new `ResourceGroup` object just to add a single atomic variable along with the other methods, we apply the `lock` and `unlock` primitives directly on that atomic variable, possible due to `Resource` interface inherited by atomic types, as explained in Section 4.2.2. This optimization effectively avoids the overhead associated with the creation of a new `ResourceGroup` object. However, there is a particular scenario in which this optimization cannot be employed. Due to the nature of this optimization, if the atomic variable is aliased throughout the method, the resource initially `locked` at the start might not be the same resource being `unlocked` at the end. As such, the single resource mapper can only be employed when only one atomic variable is to be acquired in the method and that atomic variable is never aliased throughout the method, i.e. the resource referenced by that atomic variable is always the same during the execution of the method. In all other cases, the normal mapper is employed.

Let `l` be a lock associated with an atomic variable or a partition. The single resource mapper diverges from the mapper in the sense that only two statements have to be injected: `l.lock()` and `l.unlock()`. The `lock` statement is injected at the start of the method. If the atomic variable is not declared at the start of the method, the injection of the `lock` statement has to be accordingly delayed until it is declared. Consequently, the `unlock` statement shares the same mapping as `rg.unlock()` in the multi-resource mapper implementation: inject at all possible exit points of the method. An example of the utilization of this mapper is illustrated in Listing 4.12, where a global partition is the single resource in the method.

The single-resource mapper shares the same limitations (and respective solutions) of its multi-resource counterpart, namely lack of support for releasing locks in case of an exception.

4.5.4 Injecting the missing `imports`

After all the injection of statements in the AST, some of the classes modified by the mapper require additional `import` statements in the class declaration. These additional imports required derive from three sources: the use of the *ResourceGroups* API, the use of the global partition holder class and the use of the new atomic types. This phase, **ImportPhase**, takes requests from other phases to add the dependencies required for certain classes. Every time the mapper injects a `ResourceGroup` object, a request is sent to add the *ResourceGroup* class to the list of imports, if missing. Similarly, when the mapper injects a global partition, a request sent to add the `GlobalPartitionHolderClass` to the list of imports, if missing. The other source of requests is the *ConvertPhase*. When it converts a type into its atomic equivalent, a request is sent to add the atomic class to the list of imports, if missing.

```
public boolean contains(int key) {
        GlobalPartitionsHolderClass.GLOBAL_partition_00.lock();
        AtomicRBNode node = getRoot();
        while (node != sentinelNode) {
            if (key == node.getValue()) {
                boolean returnExpression = true;
                GlobalPartitionsHolderClass.GLOBAL_partition_00.unlock();
                return returnExpression;
            } else {
                if (key < node.getValue()) {
                    node = node.getLeft();
                } else {
                    node = node.getRight();
                }
            }
        }
        boolean returnExpression = false;
        GlobalPartitionsHolderClass.GLOBAL_partition_00.unlock();
        return returnExpression;
    }
```

Listing 4.12: Example of `contain` method of the `RBTree` benchmark after the use of the single-resource mapper.

5

Evaluation

In this chapter we perform an evaluation of our model and implementation. The model will be compared against competitor models according to their productivity and the properties guaranteed by each system. The implementation's performance will be compared in a set of benchmarks against the state of the art in concurrency control mechanisms.

5.1 Productivity

In this section, we will present an analysis of the productivity enabled by our system. It will be compared against the current state of the art in data-centric concurrency control, *AJ (atomic sets)*, and against the state of the art code-centric approaches based on atomic sections.

5.1.1 *versus AJ*

A clear advantage our system provides against *AJ* in the simplicity department is the reduced number of constructs used by our system, one, against the five constructs used by *AJ*. These five constructs also hold distinct semantic meanings as exposed in Section 2.2.1, while our system's solo annotation has a single semantic meaning. Furthermore, the programmer has to manually specify the atomic set that each variable belongs to, which is not required in our model.

We claim that most of these annotations should be delegated to the underlying system. The programmer should not be encumbered by having such a considerable number of annotations with distinct use-cases.

	AJ						Our system
	atomicset	atomic	owned	unitfor	alias (/*this.L=L*/)	Total	@Atomic
TSP	2	9	0	0	0	**11**	**9**
Elevator	1	4	0	8	8	**21**	**4**
Weblech	2	4	0	0	0	**6**	**4**
Jcursez1	5	15	0	16	29	**65**	**25**
Cewolf	4	5	0	0	1	**10**	**5**
Collections	5	53	0	40	330	**428**	**146**

Table 5.1: Comparison between the number of annotations required in *AJ* and our system.

A direct comparison of the number of constructs required with *AJ* against our system is illustrated in Table 5.1. The programs used are part of the programs provided by the authors of *AJ*[1], with the required modifications already made. It is possible to observe that our system requires a lower number of annotations on all the programs. The use of `atomic(s)` on *AJ* is a subset of our use of the `@Atomic` annotation. Most other annotations in *AJ* simply do not require an equivalent in our system. The `atomicset(s)` construct is not used because we do not require the explicit association of an atomic variable to an atomic set. The `unitfor` construct, used to express that an atomic scope must obtain exclusive access to more than one resource set, shares some affinity with our use of the `@Atomic` annotation on function parameters.

In retrospective, the five constructs are severally different and require careful thought about their application. It is not trivial for a programmer to identify the use-case for each construct immediately. As such, we argue that the complexity of the mental process of the programmer is much higher when using *AJ* compared to our system. Moreover, the non-trivial reasoning from *AJ* is more prone to mistakes in their use. These mistakes might lead to the violation of desirable atomicity constraints. In more extreme scenarios, these mistakes might even lead to deadlock situations.

In our system, the annotation of which variables must be evaluated atomically is enough to express concurrency restrictions in a safe manner. This safety in enforced by the type system, assuring that all atomic resources are always referenced by atomic variables who, in turn, are guaranteed to be evaluated in a unit of work that guarantees its atomicity. Furthermore, due to the nature of the type system, a misuse of our annotations will quickly trigger a type check error, effectively preventing the compilation and subsequent execution of a program that does not uphold the desired atomicity constraints.

Additionally, *AJ* does not guarantee the absence of deadlocks in all dynamic scenarios where transitive circular dependencies between atomic sets might occur [VTD06]. This limitation was addressed in [MHD$^+$13], delegating into the programmer the responsibility of ordering the resources by using an extra < annotation. On the other hand, our system requires only the `@Atomic` annotation to guarantee the absence of deadlocks in all scenarios, guaranteeing progress of the execution.

---

[1] Available on the authors website. `http://sss.cs.purdue.edu/projects/aj/`

However, *AJ* is more generic than our system by allowing the acquisition of two distinct atomic sets in the same unit of work and by having annotations that can be used to deal with high-level data-races, given that the programmer identifies them beforehand. Nonetheless, none of our case-studies nor the ones supplied[2] by the authors of *AJ* benefited from this feature.

	Static	Dynamic	Livelock-free	Atomicity	Isolation	Composable
Atomic Sets	Yes	Mostly yes	Yes	Weak-	Yes	Yes
Our model	Yes	Yes	Yes	Strong	Yes	Yes

Table 5.2: Overview of properties offered by our system and by *AJ*.

Illustrated in Table 5.2 is a comparison between the properties offered by our system and *AJ*. According to what was possible to determine for *AJ*, nothing guarantees that the resources referenced by an atomic set are always associated to an atomic set. Thus, we believe *AJ* only guarantees weak atomicity, unlike our system that provides strong atomicity. The other noteworthy property is deadlock-freedom in dynamic scenarios, that is not fully guaranteed in *AJ* while our system completely guarantees it.

The issue of high-level data-races might arise in both systems, but due to distinct factors. In *AJ*, high-level data-races are accounted for and avoided by the system given that the programmer applies the correct primitives. But the reasoning behind the application of those primitives and the detection of high-level data-races prone situations still resides on the programmer. Both *aliasing* or `unitfor` annotations are required and should be applied in situations where a high-level data-race could occur. One of the works related to *AJ*, presented in Section 2.2.1, proposes the dynamic detection of high-level data-races that should be annotated. However, since this approach relies on executing the program and detecting the data-races when they occur, their absence is not guaranteed by the system at compile time.

In that perspective, our system also does not guarantee the absence of data-races. Nevertheless, if detected by the programmer, they can be fixed. Firstly, a data-race can occur if at least two units of work nested in a method access the same resources. In each unit of work, the resources are evaluated atomically. However, during the transition between units of work in the caller method, the resource is not acquired by the current thread. In this scenario, between both units of work, a data-race can occur in which an outside thread might access the resource.

Consider the example in Listings 5.1 and 5.2, respectively featuring our system and *AJ* in the same scenario. In this example, a data-race might occur during the `transfer` method, between invocations of the `withdraw` and the `deposit` methods. Both implementations incur the same data-race possibility. More specifically in our system, while both `withdraw` and `deposit` methods are units of work and evaluate the atomic account

---

[2] See footnote 1.

```
public class Account {
    int balance = 0 ;
    (...)
}

public class Bank {
    @Atomic Account[] accounts ;

    (...)

    public void transfer(int accNum_A, int accNum_B, int amount) {
        withdraw(accNum_A, amount) ;
        //during this transition, both accounts can be accessed
        deposit(accNum_B, amount) ;
    }

    public void withdraw(int accNum, int amount) {
        @Atomic Account acc = accounts[accNum] ;
        acc.balance -= amount ;
    }

    public void deposit(int accNum, int amount) {
        @Atomic Account acc = accounts[accNum] ;
        acc.balance += amount ;
    }
}
```

Listing 5.1: An example of a bank transfer between two accounts that incurs a high-level data-race in our system.

referenced, the `transfer` method is not a unit of work because it does not reference any atomic variable.

After the programmer detects this scenario, it can effectively avoid the data-race by forcing the acquisition of this resource in the method where both units of work are nested, as shown in Figures 5.3 and 5.4. Since now two accounts are acquired by the caller method, the transition between the inner units of work is not seen by an outside thread, effectively avoiding the data-race between the two inner units of work. We argue that this process of fixing perceived data-races is simpler when employing our system compared to *AJ*.

5.1.2 *versus* atomic sections

Firstly, an obvious advantage of our system is the use of a data-centric concurrency control system, with all the benefits that it entails. In this regard, the most positive aspect is the centralization of concurrency errors. In a code-centric approach, such as in atomic sections, a concurrency error might arise in the incorrect use of a single construct, or from the lack of it in a single critical section.

However, by employing a data-centric approach, the use of lower level concurrency mechanisms is delegated on the underlying system, in which the programmer only has to annotate the variables that should be atomically evaluated throughout the program.

```
public class Account {
    atomicset(a)
    atomic(a) int balance = 0 ;
    (...)
}

public class Bank {
    atomicset(s)
    atomic(s) Account[] accounts ;

    (...)

    public void transfer(int accNum_A, int accNum_B, int amount) {
        withdraw(accNum_A, amount) ;
        //during this transition, both accounts can be accessed
        deposit(accNum_B, amount) ;
    }

    public void withdraw(int accNum, int amount) {
        Account acc = accounts[accNum] ;
        acc.balance -= amount ;
    }

    public void deposit(int accNum, int amount) {
        Account acc = accounts[accNum] ;
        acc.balance += amount ;
    }
}
```

Listing 5.2: An example of a bank transfer between two accounts that incurs a high-level data-race in *AJ*.

```
public class Bank {
    @Atomic Account[] accounts ;

    (...)

    public void transfer(int accNum_A, int accNum_B, int amount) {
        @Atomic Account acc_A = accounts[accNum_A] ;
        @Atomic Account acc_B = accounts[accNum_B] ;
        withdraw(acc_A, amount) ;
        //during this transition,
        //  both accounts are still acquired by this thread
        deposit(acc_B, amount) ;
    }

    public void withdraw(@Atomic Account acc, int amount) {
        acc.balance -= amount ;
    }

    public void deposit(@Atomic Account acc, int amount) {
        acc.balance += amount ;
    }
}
```

Listing 5.3: Fixing the high-level data-race presented in Listing 5.1 using our system.

```
public class Account {
    atomicset(a)
    atomic(a) int balance = 0 ;
    (...)
}

public class Bank {
    atomicset(s)
    atomic(s) Account[] accounts ;

    (...)

    public void transfer(int accNum_A, int accNum_B, int amount) {
        withdraw(acc_A, amount) ;
        deposit(acc_B, amount) ;
    }

    public void withdraw(unitfor(a) Account acc, int amount) {
        Account|a=this.s| acc_A = accounts[accNum_A] ;
        acc.balance -= amount ;
    }

    public void deposit(unitfor(a) Account acc, int amount) {
        Account|a=this.s| acc_B = accounts[accNum_A] ;
        acc.balance += amount ;
    }
}
```

Listing 5.4: Fixing the high-level data-race presented in Listing 5.2 using *AJ*.

If any concurrency error is found in this approach, it is surely at the declaration of an atomic variables.

Additionally, in our system, it is enough to annotate a variable with `@Atomic` to disseminate its concurrency requirements throughout the program. The compiler will refuse to compile until all these are fulfilled. In other words, until all its aliases (both local variables and function parameters) are also annotated with `@Atomic`.

	Atomic sections	Our system		
	Total	Primary	Derived	Total
TSP	**15**	9	0	**9**
Elevator	**9**	4	0	**4**
Bank	**3**	1	5	**6**
IntHashSet	**3**	5	0	**5**
LinkedList	**3**	1	13	**14**
RBTree	**3**	3	15	**18**

Table 5.3: Comparison between the number of annotations required in *DeuceSTM* and our system.

A comparison on the number of annotations required was done against *DeuceSTM*, a optimistic software transactional memory system that uses atomic sections. The results attained are illustrated in Table 5.3. The *primary* column symbolizes obligatory annotations, from whom all other can be derived, represented in the *derived* column. Four of the six programs, belonging to the *DeuceSTM* benchmark suite, resulted in a higher number of annotations when using our system. The numbers of annotations in the benchmarks that use recursive data-structures (*RBTree* and *LinkedList*) is inflated due to the great use of atomic local variables that have to be annotated. However, in these two benchmarks, a significant portion of the annotations are not primary, but derived from the primary annotations. This inference of derived annotations could be completely delegated into an IDE, further emphasizing the centrality of the system and reducing the number of annotations required.

Moreover, it is important to understand that the results are in part regulated by the relation between the number of methods and the number of data that requires atomic execution. Even though each benchmark shares some degree of complexity, with multiple variables (more than three in all programs) that store the benchmark's state, only three operations need to be effectively synchronized. This means that the atomic sections only need to be applied to three functions. As such, these benchmarks are, in a way, not a good representative of the data-centric paradigm.

The justification is inherently correlated with the scaling of data-centric annotations. A program with a single structure that holds its state and uses hundreds of methods will only require, in minimum, one primary single data-centric annotation in the said structure. On the other hand, a code-centric system would have to identify the methods that make use of the said structure and then apply the atomic section construct, certainly, a greater number of times compared to the data-centric approach. Furthermore, forgetting a critical atomic section construct would issue no compile time error whatsoever, but the atomicity constraints would be broken.

The reverse is also true, as verified in the aforementioned four benchmarks. A program with a great number of distinct structures that only has three operations that have to be performed atomically will probably require a lower number of atomic section constructs compared to a data-centric approach. Nevertheless, we argue that the latter scenario is less common and more artificial.

Additionally, most code-centric systems either do not guarantee deadlock-freedom in all scenarios or do not guarantee livelock-freedom. We have implemented the proposed model in such a way that progress is ensured in all scenarios. A formal proof of such claim is in the works.

5.2 Performance

The main focus of this thesis is not the performance. Nonetheless, we still want to assess the performance of our prototype implementation. Ergo, in this section, we compare its performance against *AJ* and leading techs in code-centric concurrency for Java.

Regarding the implementations available for *AJ*, we only had access to a limited set of programs, both in the original and transformed versions. The translation mechanism used by *AJ* was not supplied by the authors, thus we were limited to the source code and subsequent Java translation of a set of six examples. Only one of those programs qualified as a performance benchmark, the travelling salesman problem (TSP), thus representing the only benchmark used in this comparison.

Regarding the code-centric comparison, we selected both Java synchronized blocks and a reference STM, *DeuceSTM*. The benchmarks used for this comparison were taken from the *DeuceSTM* benchmark suite. Those were then adapted into a naïve implementation using Java synchronized blocks, where all methods have the *synchronized* keyword added.

For each benchmark configuration (thread number, contention), a minimum of 100 executions were made. Exceptionally, due to its short execution time, the *TSP* benchmark was executed a minimum of 1000 times for each configuration. The execution of each benchmark lasted thirty seconds, except the *TSP* benchmark where the execution time is the actual benchmark result. The bottom 10% and the upper 10% of results from each configuration were excluded to eliminate potential outliers. The remaining values were used to calculate the average, that in turn was used to plot the graphs presented in this section.

Test infrastructure All tests were ran on a machine with 4 AMD Opteron™ processors with 16 cores each, totalling 64 cores, and with 64 GigaBytes of RAM. The Java version used to compile and run the benchmarks was `OpenJDK 8 Build b94 x86_64`.

Initial Remarks The operations which rely only on reading values from arrays, such as calculating the sum of money in all accounts in the bank benchmark, achieved a great increase in scalability when using read-write locks compared to the naïve implementation using exclusive locks. In the other operations where that pattern was not present, no decrease in performance was verified. This is explained due to the fact that acquiring a write lock in a read-writer lock has equal overhead to the acquisition of a writer-only lock. As such, the results presented are relevant only to the read-write lock implementation. Additionally, both the original and final codes generated[3] by our plugin used in these benchmarks are available, respectively, in Appendix A and B.

5.2.1 Comparison with *AJ*

The travelling salesmen benchmark consists of a group of threads that will test all possible combinations of paths possible, update the best path found so far if applicable to the current path and continue testing other paths. While not embarrassingly parallel, it is possible to achieve a great degree of concurrency, since only accesses to the structures

---

[3] While our plugin does not generate code *per se*, it is possible to print sources right before the bytecode is generated using the hidden *javac* compilation flag '`-printsource -d <generated_dir>`'.

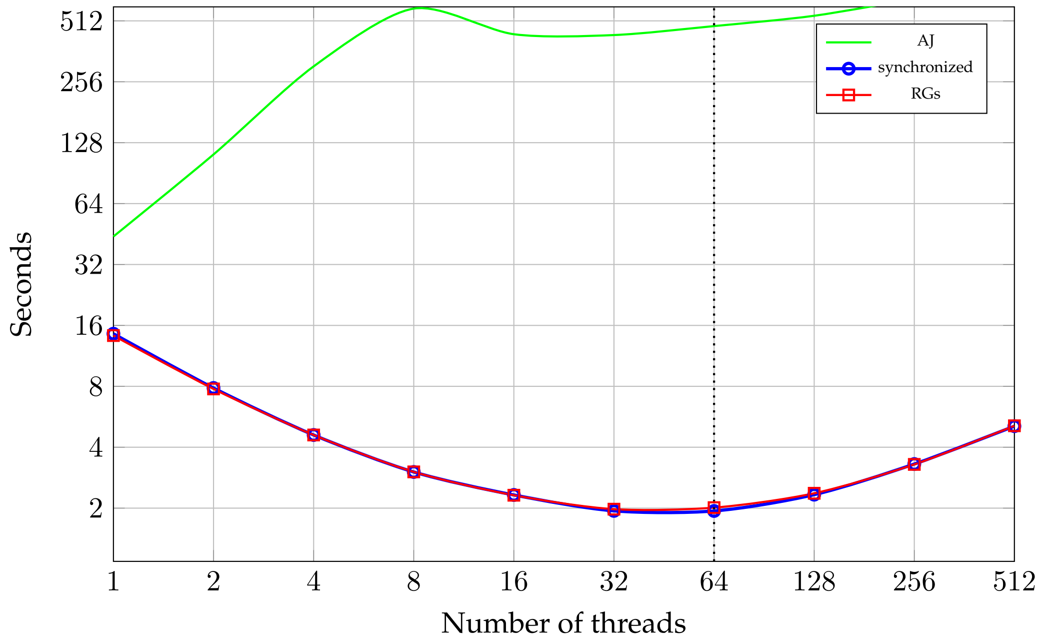


Figure 5.1: TSP benchmark performance comparison.

that hold the best path found so far have to be protected. The iterative procedure of testing the path is completely private to each thread.

As illustrated in Figure 5.1, our system manages to provide the exact same gain in performance as the fine-grained synchronized blocks implementation, arguably, the best possible performance for this benchmark. On the other hand, *AJ* does not scale at all in this benchmark. In it, the application of concurrency control is very coarse-grained (one lock to protect all structures), that leads to the serialization of the benchmark at best. Furthermore, as the number of threads increases, the contention between threads in the coarse-grain synchronized blocks used by *AJ* actually degenerates the performance.

5.2.2 Comparison against synchronized blocks and *DeuceSTM*

In this section, both static and dynamic scenarios will be evaluated. In specific, the bank and *hash set* benchmarks make use of static scenarios, while the other two benchmarks, red-black tree and linked list, rely on dynamic scenarios. Note that the later three benchmarks, taken from *DeuceSTM*, are various implementations of an **IntSet**, a set of integers. This structure supports the addition, removal, and lookup of an element. Each implementation uses a distinct underlying data structure, namely a single linked list, a hash table and a red-black tree.

Bank The *bank* benchmark from DeuceSTM simulates, as the name implies, a bank. The bank supports the addition and removal of accounts, transferring money between two accounts, adding a fixed interest rate to all accounts and calculating the sum of money in all accounts. First and foremost, the most distinguishing aspect of this benchmark is

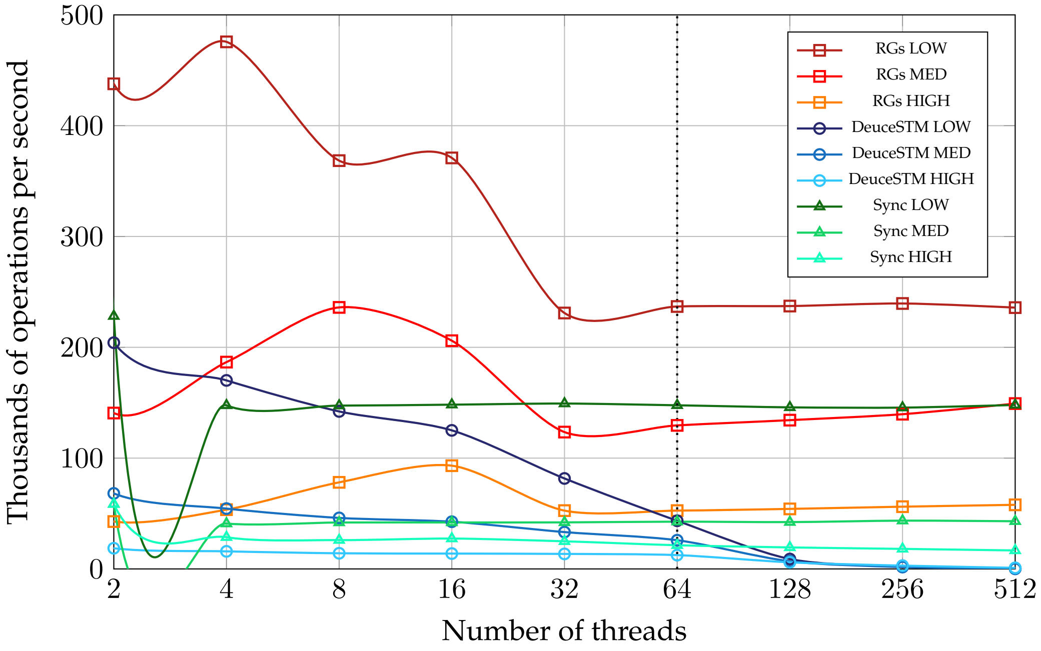


Figure 5.2: Bank benchmark performance comparison.

the fact that it consists almost exclusively in static scenarios. As such, this benchmark provides the best representation of the potential for concurrency in a pessimistic system, such as ours.

Observation of Figure 5.2 reveals that our system outperforms software transactional memory at all contention levels. Due to the static nature of the transactions between two accounts where an ordered locking of both accounts is sufficient to assure atomicity, our pessimistic system yields great performance compared to software transactional memory that has retry this operation due to the emergence of conflicts. This fact is especially revealing at higher thread counts, where DeuceSTM converges into a very low throughput compared to our system.

In this benchmark, the naïve Java synchronized block implementation achieved worse performance compared to our system. Our data-centric approach permits the concurrent execution of transfers upon non-overlapping pairs of accounts, while synchronized blocks implementation does not. The superior results at lower thread counts can be justified by the locality to a single CPU (16 cores) of the concurent operations and lock operations used. As such, this gain in concurrency allows our system to offset the overhead from the use of fine-grained synchronization primitives, ultimately allowing our system to outperform the synchronized blocks implementation.

Hash set The *hash set* benchmark builds from a *hash table* implementation. As illustrated on Figure 5.3, our system manages to outperform transactional memory at medium and high contention. The hash table relies on a single array, and most operations require access to a single position of the array. Due to disjoint data accesses patterns, transactional

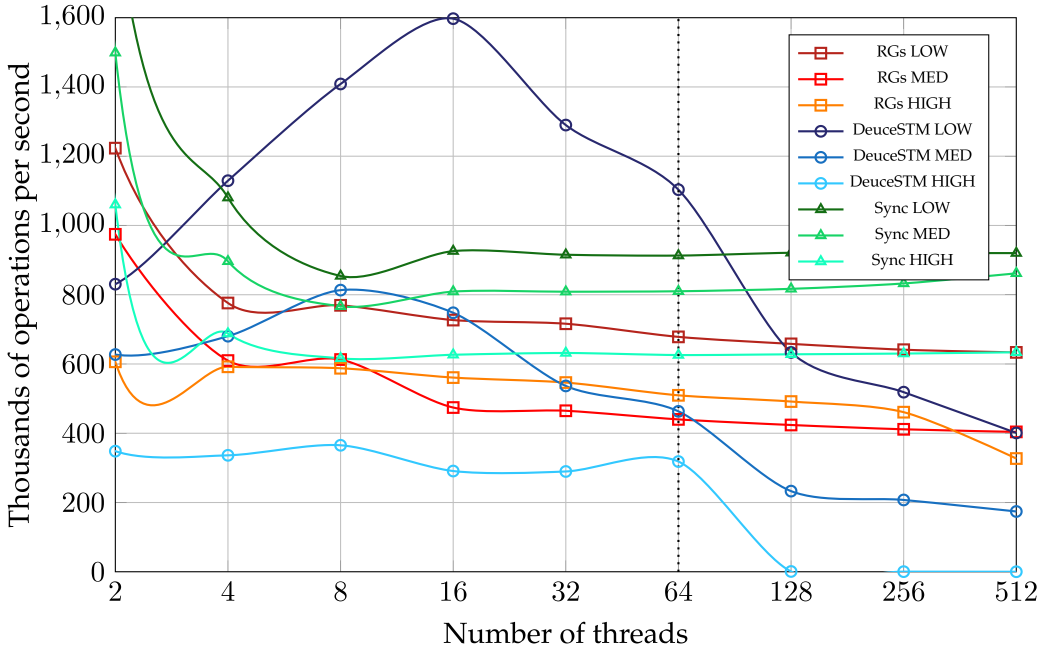


Figure 5.3: HashSet benchmark performance comparison.

memory thrives under low contention, as expected. The increase in the possibility of conflicts are exacerbated as contention increases. This is specially relevant because the `add` method might require a rehash of the table, a costly operation that requires access to all positions of the array, disrupting accesses to all of them.

The synchronized blocks implementation outperformed, surprisingly, both systems. The only combination that managed to outperform this implementation was DeuceSTM under low contention. The hash table operations used (`add`, `remove` and `contains`) share a constant temporal complexity ($O(1)$) on average. As such, the operations are too short-lived to compensate the very significant overhead inherent to the fine-grained use of synchronization primitives in our system. As such, the synchronized blocks implementation outperforms our system in this scenario.

During the high contention test, as the number of threads increased beyond the number of physical cores, DeuceSTM entered regularly in livelock on the rehash operation. The same scenario did not occur on our system due to the acquisition of a write-lock on the `add` method. However, such lock is not required for a regular add operation. Only a read-lock to the array and a write-lock to the specific position accessed would be required. This limitation is imposed due to the inability to promote read-locks to write-locks. As such, the compiler settles for the highest denominator access found, which in this case represents the possibility to invoke the rehash operation.

Due to the programming model of our system, it is possible to avoid this write-lock and convert it into a read-lock. This would require a slight re-architecture of the `add` method. Instead of finding the correct position and then rehashing if required after insertion, all in the same method, the programmer would develop an outer method, as

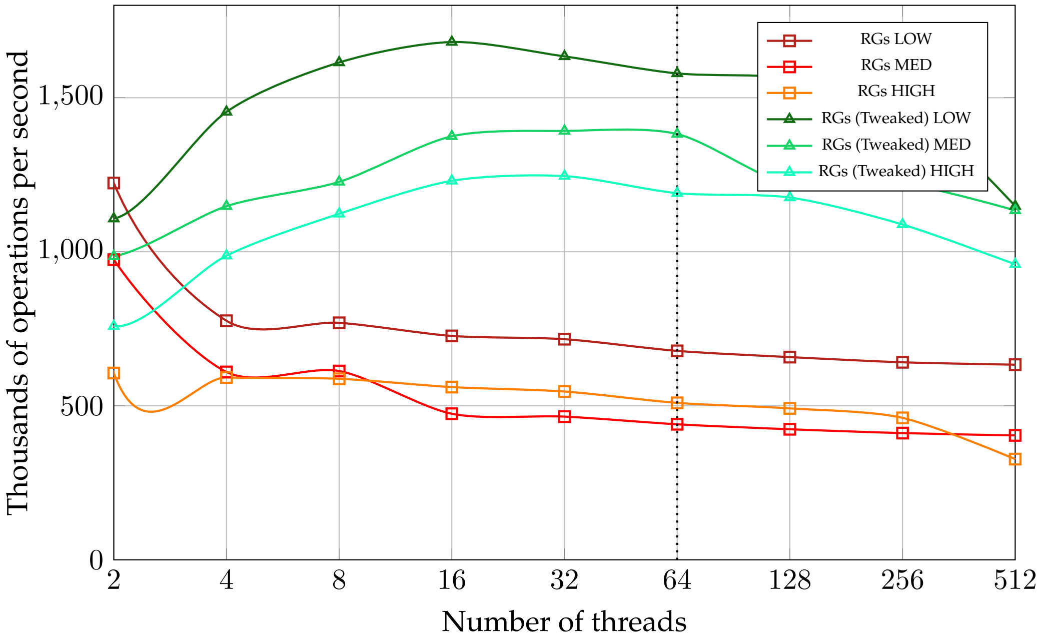


Figure 5.4: HashSet benchmark performance comparison, after tweaking the `add` method to reduce the usage of the write-lock.

illustrated in Listing B.1. This new `add` outer method would invoke a new `new_add` (the old `add`) method that adds the integer to the table if it does not require a rehash, failing otherwise. If the `new_add` method succeeded, rehash would not be required. Since the new `new_add` method does not require a rehash, the table array only requires the acquisition of a read-lock. As such, the write-lock is then exclusively delegated into the `rehash` method. Since the rehash is only required if the `new_add` fails, the write-lock acquisition in the `rehash` method is only acquired when strictly necessary. Additionally, due to the extremely limited amount of rehashes required during the execution of the program, concurrency is improved, as shown in Figure 5.4.

However, this approach has its pitfalls: since the new outer method does not access the `@Atomic` array directly, atomicity will not be enforced on this method, but rather on the delegating methods. As such, the programmer has to keep this in mind when applying this tweak to the program, and attempt to repeat the rehash/add operation until it succeeds. This tweak is artificially generating a data-race between the rehash and the add method that was not present before, hence the need to retry. This subject is closely related to the high-level data-races issue discussed in Section 5.1.1. While not a simple modification, it highlights the power and flexibility of the model presented in this thesis.

Red-black tree The *red-black tree* benchmark had mixed performance, as illustrated on Figure 5.5. Due to the disjoint nature of the tree and most operations occurring on a specific node instead of modifying the whole tree, transactional memory provides better

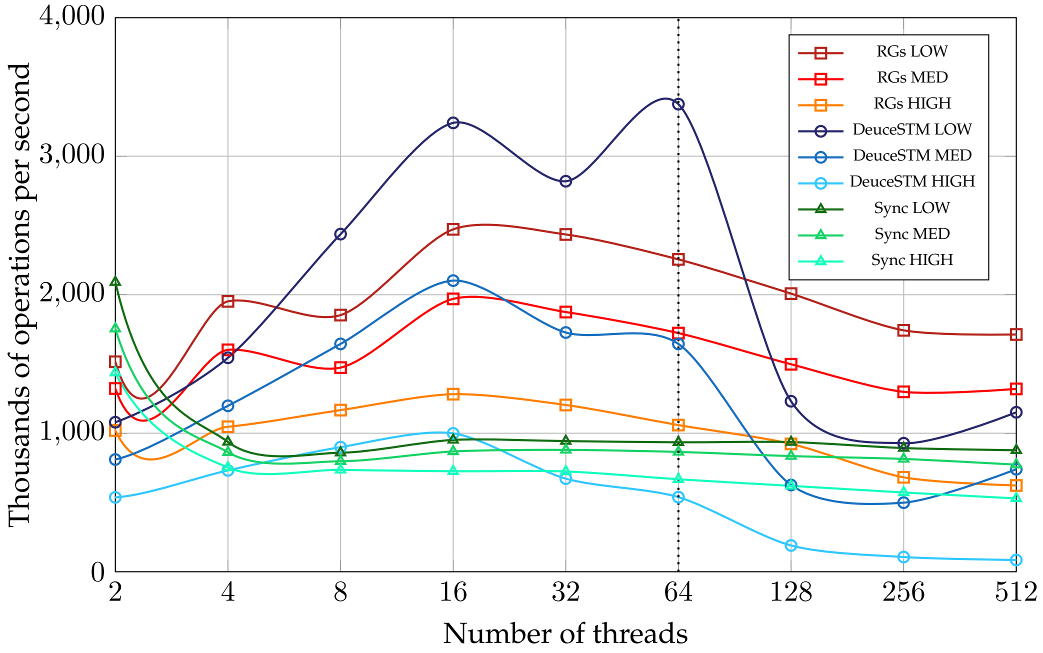


Figure 5.5: RBTree benchmark performance comparison.

performance at lower contentions. As contention increases, conflicts increase, diminishing the transactional memory throughput. Our pessimistic approach becomes more desirable at higher contentions, outperforming transactional memory at medium and high contention.

The synchronized blocks implementation achieved inferior performance to our system. However, since our system also forces serialization due to partitions injected, the serialization cannot be blamed for the inferior performance in the synchronized blocks implementation. In fact, this worse performance is attributed to the forced re-acquisition of the locks in the synchronized blocks. The three main methods in the red-black, `add`, `remove`, and `contains`, invoke several sub-methods, that are also using synchronized blocks. As such, a single invocation of the main methods might require several lock re-acquisitions in the sub-methods. But in our system, those locks in the sub-methods are removed due to the redundant lock identification and removal algorithm presented in Section 4.4.2. In conclusion, even though our system also serializes the main methods in the red-black tree, these methods have a lower overhead due to the absence of the redundant locks that are present in the synchronized blocks implementation, allowing our system to perform better.

Linked list The *linked list* implementation had the worst performance by a great margin as illustrated on Figure 5.6. Traversing a (singly) linked list represents a dynamic scenario due to the inability to order the resource acquisitions beforehand. To avoid the possibility of a deadlock in these situations, the partitioning algorithm generates a partition to protect unordered acquisition of list nodes. This partition, when applied to all methods

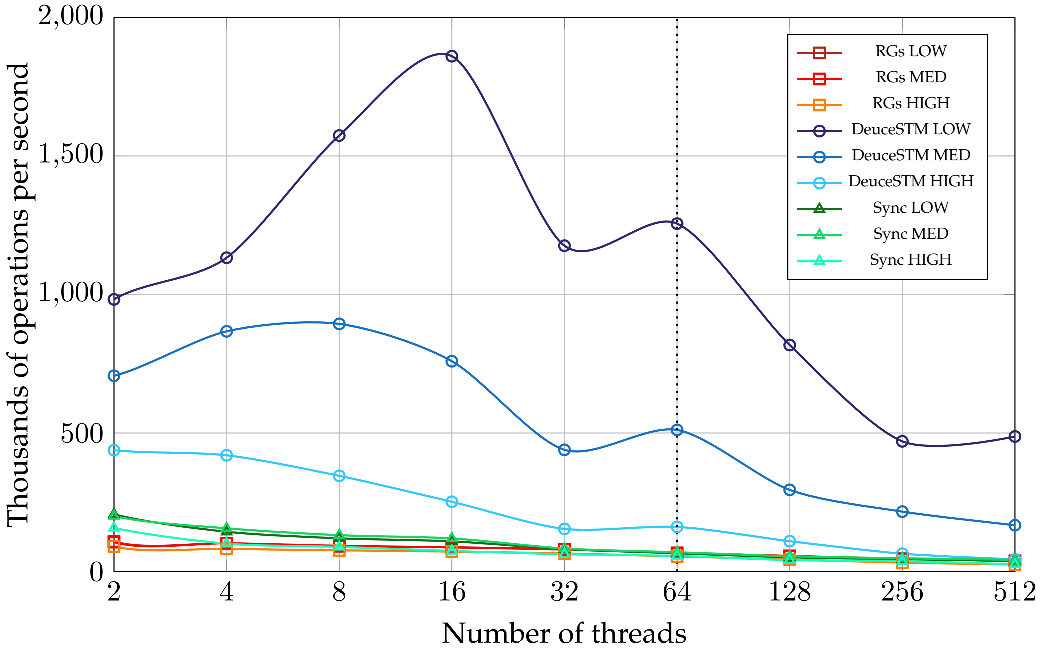


Figure 5.6: LinkedList benchmark performance comparison.

that traverse the linked list will effectively serialize the execution of those methods.

However, such protection is unnecessary due to the way that the list is traversed. The list is traversed always from the head to the tail. As such, the deadlock-prone situation here detected does not happen due to the one-way recursive nature of the structure.

Without the partition, as illustrated in Listing B.2, only the previous and next node are acquired at any time while the correct position is found, leaving the remaining nodes free for the other threads. As such, the program remains correct and deadlock-free. Removing the partition and making a fine-grained approach where each node is acquired and released as the list is traversed, our system achieves increased scalability, as illustrated in Figure 5.7. Unfortunately, due to the severe overhead incurred by having one lock operation per node, the performance is not as good as the increased parallelism would have lead us to think. These kind of situations, where partitions are not effectively required, could be inferred by the compiler and is a possibility for future work.

The linked list benchmark is an example of a data structure that is very fond of an optimistic approach to concurrency, greatly due to the low rate of conflicts. The nodes traversed are not modified, therefore, a conflict will only arise on the lesser possibility multiple threads modified the exact same previous and next nodes. As such, the high performance of DeuceSTM in this benchmark does not come as a surprise, outperforming both our system and the synchronized blocks implementation.

Similarly to the red-black tree benchmark, the partition effectively serializes the accesses to the linked-list. Again, this culminates in our system having equivalent performance to the synchronized blocks implementation due to the similarity between the

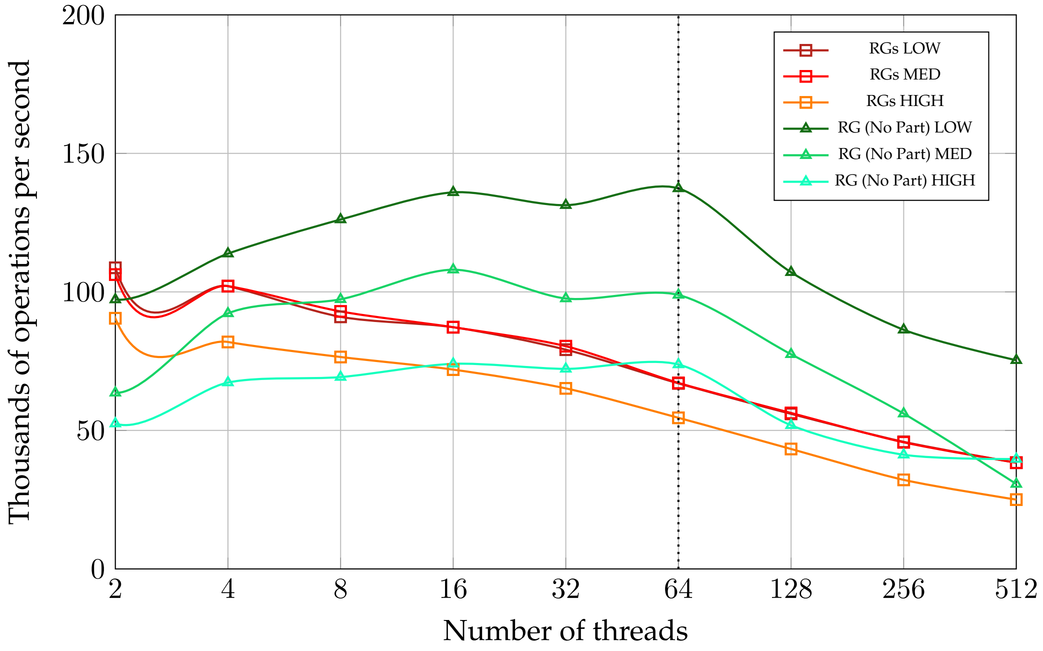


Figure 5.7: LinkedList benchmark performance comparison against a variant without the superfluous partition and using fine-grained locks.

partition's mapping to the *synchronized* keywords applied to the functions of the synchronized blocks implementation.

5.2.3 Closing Remarks

As a whole, our system manages to achieve respectable performance compared to other alternatives. The high performance inherent to the fact it adopts a pessimistic approach is evidenced in programs that make use of static scenarios, where optimistic approaches fall short. Conversely, programs that rely on dynamic scenarios might achieve better performance with an optimistic approach if they also report a low conflict rate. When high conflict rates are present, our system will most likely outperform software transactional memory.

Regarding the synchronized blocks implementation, we argue our model, besides all the properties, centrality and non-intrusivity it brings to the table, also allows the implementation of a system that can allow greater performance that a coarse synchronized blocks approach to concurrent programming, as was exposed in this section. Even though several optimizations are identified for future work, and that the performance of our system was not the main contribution of this thesis, the performance achieved is already very positive in a system that is still very recent. We believe that further work in this implementation will allow us to further close the gap between our system and the manual application of low level synchronization primitives.

6

Conclusion

In this work we proposed a data-centric concurrency management model, with a single annotation, that guarantees strong atomicity and deadlock-freedom. Our model makes uses of the type-system to enforce these properties. To validate this model, a prototype was produced based on pessimistic concurrency. This prototype was then compared against state of the art concurrency control mechanisms, in both performance and productivity.

In the field of productivity, our system managed to require less modifications of the code, on average, to achieve the desired atomicity constraints. In scenarios where our system required more annotations, a large percentage of those annotations could be inferred from a smaller subset of annotations and delegated into a tool such as an IDE. Additionally, we believe the use of our system is more intuitive compared to the other state of the art in data-centric concurrency control, *AJ*, due to only requiring one single annotation with a single meaning, unlike the latter. This simplicity, however, does not result in a loss in expressiveness. While *AJ* provides annotations to avoid high-level data-races, they require that the programmer identifies the high-level data-races correctly and apply the correct annotations. In turn, in our model it is also possible to avoid high-level data-races that the programmer identifies by using the atomic variables to the method that manifests said data-races. Regarding the properties, our system manages to guarantee strong atomicity and deadlock-freedom in all scenarios, while *AJ* only guarantees weak atomicity and deadlock-freedom in some dynamic scenarios.

On the subject of performance, our system managed to achieve very competitive results, despite the fact that the performance of the prototype was not the focus of this thesis. In specific, applications that rely heavily on static scenarios achieve a very positive performance using our system. Nevertheless, the results obtained show that the prototype is mature and represents a good starting point for future work.

Regarding future work, and as referenced throughout this thesis, there are a couple of aspects that could be further developed in the future. First, it would be interesting to explore more sophisticated techniques that might allow us to reduce the number of false-positives regarding the detection of deadlock-prone scenarios. Second, and from a technical standpoint, the calculation of types in arbitrary expressions should be redone as to use the compiler's type system if possible. Third, the creation of atomic types should be reimplemented without the use of the `extends` clause, allowing it to be completely faithful to our model. Fourth, the work units should be surrounded by a `try/catch` block to release the group acquired even in the face of an unexpected exception. Finally, performance improvements regarding the mapping performed could be attained by attempting to narrow down the computations performed while having a group acquired.

The outcome of the work developed in this thesis was published at INForum 2013 [PP13]. We believe that the work developed met our expectations, and will hopefully foster additional future developments.

Bibliography

[AAK$^+$05] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005), 12-16 February 2005, San Francisco, CA, USA*, pages 316–327, 2005.

[APT79] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Information Processing Letters*, 8(3), 1979.

[BLM06] Colin Blundell, E Christopher Lewis, and Milo M. K. Martin. Unrestricted Transactional Memory: Supporting I/O and System Calls within Transactions. Technical Report CIS-06-09, Department of Computer and Information Science, University of Pennsylvania, Apr 2006.

[Boe09] Hans-J. Boehm. Transactional memory should be an implementation technique, not a programming interface. In *Proceedings of the First USENIX conference on Hot topics in parallelism*, HotPar'09, pages 15–15, Berkeley, CA, USA, 2009. USENIX Association.

[Bou09] Gérard Boudol. A Deadlock-Free Semantics for Shared Memory Concurrency. In *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, pages 140–154, 2009.

[CCG08] Sigmund Cherem, Trishul M. Chilimbi, and Sumit Gulwani. Inferring locks for atomic sections. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*, pages 304–315, 2008.

[DHM$^+$12] Julian Dolby, Christian Hammer, Daniel Marino, Frank Tip, Mandana Vaziri, and Jan Vitek. A data-centric approach to synchronization. *ACM Transactions on Programming Languages and Systems*, 34(1):4, 2012.

[Dij65] Edsger W. Dijkstra. Cooperating Sequential Processes, Technical Report. Technical Report EWD 123, 1965.

[DP12] Nuno Delgado and Hervé Paulino. Sobre um mecanismo de controlo de concorrência baseado em grupos de recursos. In Antónia Lopes and José Orlando Pereira, editors, *INForum 2012 - Atas do 4º Simpósio de Informática*, pages 302–305. Universidade Nova de Lisboa, 09 2012. URL=http://asc.di.fct.unl.pt/ herve/papers/RG-INForum2012.pdf.

[EBA$^+$11] Hadi Esmaeilzadeh, Emily R. Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *38th International Symposium on Computer Architecture (ISCA 2011), June 4-8, 2011, San Jose, CA, USA*, pages 365–376, 2011.

[EFJM07] Michael Emmi, Jeffrey S. Fischer, Ranjit Jhala, and Rupak Majumdar. Lock allocation. In *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17-19, 2007*, pages 291–296, 2007.

[EK08] David Erni and Adrian Kuhn. The hacker's guide to Javac. Bachelor's thesis, supplementary documentation, University of Bern, August 2008.

[GLPT76] Jim Gray, Raymond A. Lorie, Gianfranco R. Putzolu, and Irving L. Traiger. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In *IFIP Working Conference on Modelling in Data Base Management Systems,Freudenstadt, Germany*, pages 365–394, 1976.

[HDVT08] Christian Hammer, Julian Dolby, Mandana Vaziri, and Frank Tip. Dynamic detection of atomic-set-serializability violations. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, pages 231–240, 2008.

[HFP06] Michael Hicks, Jeffrey S. Foster, and Polyvios Pratikakis. Lock Inference for Atomic Sections. *TRANSACT'06: Proceedings of the 1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing, Ottawa, Canada.*, 2006.

[HLR10] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[HR83] Theo Härder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM Computing Surveys*, 15(4):287–317, 1983.

[Jin12] Wasuwee Sodsong Jingun Hong. Design, Implementation and Evaluation of a Task-parallel JPEG Decoder for the Libjpeg-turbo Library. *International Journal of Multimedia and Ubiquitous Engineering*, 7(2), 2012.

[Jon07] Simon Peyton Jones. Beautiful Concurrency. In *Beautiful Code: Leading Programmers Explain How They Think (Theory in Practice (O'Reilly))*. O'Reilly Media, Inc., 2007.

[Lom77] David B. Lomet. Process structuring, synchronization, and recovery using atomic actions. In *Language Design for Reliable Software 1977, Raleigh, North Carolina*, pages 128–137, 1977.

[MBM$^+$06] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: log-based transactional memory. In *12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, February 11-15, 2006*, pages 254–265, 2006.

[MHD$^+$] Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. Detecting deadlock in programs with data-centric synchronization. In *International Conference on Software Engineering (ICSE), San Francisco, CA, 2013.*

[MHD$^+$13] Daniel Marino, Christian Hammer, Julian Dolby, Mandana Vaziri, Frank Tip, and Jan Vitek. Detecting deadlock in programs with data-centric synchronization. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 322–331, Piscataway, NJ, USA, 2013. IEEE Press.

[MZGB06] Bill McCloskey, Feng Zhou, David Gay, and Eric A. Brewer. Autolocker: synchronization inference for atomic sections. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 346–358, 2006.

[PD12] Hervé Paulino and Nuno Delgado. A deadlock-free data-centric concurrency control mechanism. Technical report, Departamento de Informática, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa, 2012.

[PJ98] Jens Palsberg and C. Barry Jay. The essence of the visitor pattern. In *Proceedings of the 22nd International Computer Software and Applications Conference*, COMPSAC '98, pages 9–15, Washington, DC, USA, 1998. IEEE Computer Society.

[PP13] Daniel Luis Landeiroto Parreira and Hervé Paulino. Uma abordagem alto nível ao controlo de concorrência componível centrado nos dado. In João Cachopo and Beatriz Sousa Santos, editors, *INForum 2013 - Atas do 5º Simpósio de*

Informática, pages 298–309. Escola de Ciências e Tecnologia da Universidade de Évora, 09 2013.

[PW10] Donald E. Porter and Emmett Witchel. Understanding transactional memory performance. In *IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2010, 28-30 March 2010, White Plains, NY, USA*, pages 97–108, 2010.

[RG02] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, USA, October 5-9, 2002*, pages 5–17, 2002.

[SGG05] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating system concepts (7. ed.).* Wiley, 2005.

[ST95] Nir Shavit and Dan Touitou. Software Transactional Memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing, Ottawa, Ontario, Canada, August 20-23, 1995*, pages 204–213, 1995.

[UBES09] Takayuki Usui, Reimer Behrends, Jacob Evans, and Yannis Smaragdakis. Adaptive Locks: Combining Transactions and Locks for Efficient Concurrency. In *PACT 2009, Proceedings of the 18th International Conference on Parallel Architectures and Compilation Techniques, 12-16 September 2009, Raleigh, North Carolina, USA*, pages 3–14, 2009.

[VTD06] Mandana Vaziri, Frank Tip, and Julian Dolby. Associating synchronization constraints with data in an object-oriented language. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, pages 334–345, 2006.

[WLK+09] Yin Wang, Stéphane Lafortune, Terence Kelly, Manjunath Kudlur, and Scott A. Mahlke. The theory of deadlock avoidance via discrete control. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 252–263, 2009.

A

Appendix of the annotated code

```
package tests.microbenchmarksRW.intset;

import annotations.Atomic;
import tests.microbenchmarksRW.finalClasses.*;

public class IntSetHash implements IntSet {

  @Atomic
  private XIntVector set = new XIntVector(13);
  @Atomic
  private XByteVector states = new XByteVector(13);
  @Atomic
  private XInteger free = new XInteger(states.value.length);
  @Atomic
  private XInteger size = new XInteger(0);
  @Atomic
  private XInteger maxSize = new XInteger(states.value.length/2);

  private static final byte FREE = 0;
  private static final byte FULL = 1;
  private static final byte REMOVED = 2;

  public boolean add(int value) {
    int index = findIndex(value);
    if (index < 0) {
      return false;
    }

    byte previousState = states.value[index];
    set.value[index] = value;
    states.value[index] = FULL;
    postInsertHook(previousState == FREE);

    return true;
  }


  public boolean contains(int value) {
    return index(value) >= 0;
  }


  public boolean remove(int value) {
    int index = index(value);
    if (index >= 0) {
      set.value[index] = (int)0;
      states.value[index] = REMOVED;
      size.value--;
      return true;
    }
    return false;
  }

  private int index(int value) {

    int length = states.value.length;
    int hash = (value * 31) & 0x7fffffff;
    int index = hash % length;
    int probe;

    if (states.value[index] != FREE &&
      (states.value[index] == REMOVED || set.value[index] != value)) {
      probe = 1 + (hash % (length - 2));

      do {
        index -= probe;
        if (index < 0) {
          index += length;
        }
      } while (states.value[index] != FREE &&
      (states.value[index] == REMOVED || set.value[index] != value));
    }

    return states.value[index] == FREE ? -1 : index;
  }

  int findIndex(int value){
    int length = states.value.length;
    int hash = ((value * 31) & 0x7fffffff);
    int index = hash % length;
    int probe;
    if (states.value[index] == FREE) {
      return index;
    }
    else if (states.value[index] == FULL && set.value[index] == value) {
      return -index -1;
    } else {
      probe = 1 + (hash % (length - 2));

      if (states.value[index] != REMOVED) {
        do {
          index -= probe;
          if (index < 0) {
            index += length;
          }
        }
        while (states.value[index] == FULL && set.value[index] != value);
      }
      if (states.value[index] == REMOVED) {
        int firstRemoved = index;
        while (states.value[index] != FREE &&
          (states.value[index] == REMOVED || set.value[index] != value)) {
          index -= probe;
          if (index < 0) {
            index += length;
          }
        }
        return states.value[index] == FULL ? -index -1 : firstRemoved;
      }
      return states.value[index] == FULL ? -index -1 : index;
    }
  }

  protected final void postInsertHook(boolean usedFreeSlot) {
    if (usedFreeSlot) {
      free.value--;
    }

    if (++size.value > maxSize.value || free.value == 0) {
      int newCapacity =
        size.value > maxSize.value
        ? PrimeFinder.nextPrime(states.value.length << 1)
        : states.value.length;
      rehash(newCapacity);
      maxSize.value = states.value.length/2;
    }
  }

  protected void rehash(int newCapacity) {
    int oldCapacity = set.value.length;
    int oldSet[] = set.value;
    byte oldStates[] = states.value;

    set = new XIntVector(newCapacity);
    states = new XByteVector(newCapacity);

    for (int i = oldCapacity; i-- > 0;) {
      if(oldStates[i] == FULL) {
        int o = oldSet[i];
        int index = findIndex(o);
        set.value[index] = o;
        states.value[index] = FULL;
      }
    }
  }
}
```

Listing A.1: Original `IntSetHash` class of the *hash set* benchmark.

```java
package tests.microbenchmarksRW.intset;

import annotations.Atomic;
import java.util.ArrayList;
import java.util.List;


/**
 * @author Pascal Felber
 * @since 0.1
 */
public class IntSetLinkedList implements IntSet {

  private static final long serialVersionUID = 1L;

  @Atomic
  private Node m_first = new Node(0);

  public IntSetLinkedList() {
    @Atomic Node min = new Node(Integer.MIN_VALUE);
    @Atomic Node max = new Node(Integer.MAX_VALUE);
    min.setNext(max);
    m_first = min;
  }

  public boolean add(int value) {
    boolean result;

    @Atomic Node previous = m_first;
    @Atomic Node next = previous.getNext();
    int v;
    while ((v = next.getValue()) < value) {
      previous = next;
      next = previous.getNext();
    }

    result = v != value;
    if (result) {
      previous.setNext(new Node(value,next));
    }

    return result;
  }

  public boolean remove(int value) {
    boolean result;

    @Atomic Node previous = m_first;
    @Atomic Node next = previous.getNext();
    int v;
    while ((v = next.getValue()) < value) {
      previous = next;
      next = previous.getNext();
    }

    result = v == value;
    if (result) {
      previous.setNext(next.getNext());
    }

    return result;
  }

  public boolean contains(int value) {
    boolean result;

    @Atomic Node previous = m_first;
    @Atomic Node next = previous.getNext();
    int v;
    while ((v = next.getValue()) < value) {
      previous = next;
      next = previous.getNext();
    }

    result = (v == value);

    return result;
  }
}
```

Listing A.2: Original `IntSetLinkedList` class of the *linked list* benchmark.

```
package tests.microbenchmarksRW.intset;

import annotations.Atomic;

/**
 * Created with IntelliJ IDEA.
 * User: MetaBrain
 * Date: 18-06-2013
 * Time: 10:15
 *
 * @autor metabrain (https://github.com/metabrain) - Daniel Parreira 2013
 */
public class Node {

  private static final long serialVersionUID = 1L;
  final private int m_value;
  @Atomic
  private Node m_next ;

  public Node(int value, @Atomic Node next) {
    m_value = value;
    m_next = next;
  }

  public Node(int value) {
    this(value, null);
  }

  public int getValue() {
    return m_value;
  }

  public void setNext(@Atomic Node next) {
    m_next = next;
  }

  @Atomic
  public Node getNext() {
    return m_next;
  }
}
```

Listing A.3: Original `Node` class of the *linked list* benchmark.

B

Appendix of generated code

```
package tests.microbenchmarksRW.intset;

import rgs.RWResourceGroup;
import tests.microbenchmarksRW.finalClasses.AtomicGENXByteVector;
import tests.microbenchmarksRW.finalClasses.AtomicGENXIntVector;
import tests.microbenchmarksRW.finalClasses.AtomicGENXInteger;

public class IntSetHash implements IntSet {

    public IntSetHash() {
        super();
    }
    private AtomicGENXIntVector set = new AtomicGENXIntVector(10000);
    private AtomicGENXByteVector states = new AtomicGENXByteVector(10000);
    private AtomicGENXInteger free = new AtomicGENXInteger(states.value.length);
    private AtomicGENXInteger size = new AtomicGENXInteger(0);
```

```
public class IntSetHash implements IntSet {

    (...)

    public boolean add(int value) {
        int result ;

        while((result = new_add(value))==REHASH_REQUIRED)
            postInsertHook(false);

        return result==ADDED ;
    }
}
```

Listing B.1: Implementation of the new `add` method in the *tweaked* hash set benchmark.

```
public boolean add(int value) {
        RWResourceGroup rg = new RWResourceGroup();
        boolean result;
        AtomicGENNode previous = m_first;
        rg.add(previous);
        rg.lock();
        AtomicGENNode next = previous.getNext();
        rg.add(next);
        int v;
        while ((v = next.getValue()) < value) {
            rg.remove(previous);
            previous = next;
            next = previous.getNext();
            rg.add(next) ;
        }

        result = v != value;
        if (result) {
            previous.setNext(new AtomicGENNode(value, next));
        }

        boolean returnExpression = result;
        rg.unlock();
        return returnExpression;
    }
```

Listing B.2: Mapping of the add method in the linkedlist benchmark without the partition.

```
    private AtomicGENXInteger maxSize = new AtomicGENXInteger(states.value.length /
        2);

    private static final byte FREE = 0;
    private static final byte FULL = 1;
    private static final byte REMOVED = 2;

    public boolean add(int value) {
        RWResourceGroup rg = new RWResourceGroup();
        rg.add(set);
        rg.lock();
        int index = findIndex(value);
        if (index < 0) {
            boolean returnExpression = false;
            rg.unlock();
            return returnExpression;
        }
        byte previousState = states.value[index];
        set.value[index] = value;
        states.value[index] = FULL;
        postInsertHook(previousState == FREE);
        boolean returnExpression = true;
        rg.unlock();
        return returnExpression;
    }

    public boolean contains(int value) {
        return index(value) >= 0;
    }

    public boolean remove(int value) {
        RWResourceGroup rg = new RWResourceGroup();
        rg.add(set);
    rg.lock();
        int index = index(value);
        if (index >= 0) {
            set.value[index] = (int)0;
            states.value[index] = REMOVED;
            size.value--;
            boolean returnExpression = true;
            rg.unlock();
            return returnExpression;
        }
        boolean returnExpression = false;
        rg.unlock();
        return returnExpression;
    }

    private int index(int value) {
        RWResourceGroup rg = new RWResourceGroup();
        rg.addRead(set);
        rg.lock();
        int length = states.value.length;
        int hash = (value * 31) & 2147483647;
        int index = hash % length;
        int probe;
        if (states.value[index] != FREE && (states.value[index] == REMOVED || set.
            value[index] != value)) {
            probe = 1 + (hash % (length - 2));
            do {
                index -= probe;
                if (index < 0) {
                    index += length;
                }
            }
      while (states.value[index] != FREE && (states.value[index] == REMOVED || set.
          value[index] != value));
        }
        int returnExpression = states.value[index] == FREE ? -1 : index;
        rg.unlock();
        return returnExpression;
    }

    int findIndex(int value) {
        int length = states.value.length;
        int hash = ((value * 31) & 2147483647);
        int index = hash % length;
        int probe;
        if (states.value[index] == FREE) {
            return index;
        } else {
            if (states.value[index] == FULL && set.value[index] == value) {
                return -index - 1;
            } else {
                probe = 1 + (hash % (length - 2));
                if (states.value[index] != REMOVED) {
                    do {
                        index -= probe;
                        if (index < 0) {
                            index += length;
                        }
                    }                      while (states.value[index] == FULL && set.
                        value[index] != value);
                }
                if (states.value[index] == REMOVED) {
                    int firstRemoved = index;
                    while (states.value[index] != FREE && (states.value[index] ==
                        REMOVED || set.value[index] != value)) {
                        index -= probe;
                        if (index < 0) {
                            index += length;
                        }
                    }
                    return states.value[index] == FULL ? -index - 1 : firstRemoved;
                }
                return states.value[index] == FULL ? -index - 1 : index;
            }
        }
    }

    /**
     * After an insert, this hook is called to adjust the size.value/free
     * values of the set.value and to perform rehashing if necessary.
     */
    protected final void postInsertHook(boolean usedFreeSlot) {
        if (usedFreeSlot) {
            free.value--;
        }
        if (++size.value > maxSize.value || free.value == 0) {
            int newCapacity = size.value > maxSize.value ? PrimeFinder.nextPrime(
                states.value.length << 1) : states.value.length;
            rehash(newCapacity);
            maxSize.value = states.value.length / 2;
        }
    }

    protected void rehash(int newCapacity) {
        int oldCapacity = set.value.length;
        int[] oldSet = set.value;
        byte[] oldStates = states.value;
    RWResourceGroup rg = new RWResourceGroup() ;
        set = new AtomicGENXIntVector(newCapacity);
        states = new AtomicGENXByteVector(newCapacity);
    rg.add(set);
    rg.lock();
        for (int i = oldCapacity; i-- > 0; ) {
            if (oldStates[i] == FULL) {
                int o = oldSet[i];
                int index = findIndex(o);
                set.value[index] = o;
                states.value[index] = FULL;
            }
        }
    rg.unlock();
    }
}
```

Listing B.3: Generated `IntSetHash` class of the *hash set* benchmark.

```
package tests.microbenchmarksRW.intset;

import rgs.RWResourceGroup;
import tests.microbenchmarksRW.finalClasses.AtomicGENXByte;
import tests.microbenchmarksRW.finalClasses.AtomicGENXByteVector;
import tests.microbenchmarksRW.finalClasses.AtomicGENXIntVector;
import tests.microbenchmarksRW.finalClasses.AtomicGENXInteger;

public class IntSetHash_Tweaked implements IntSet {

    public IntSetHash_Tweaked() {}

    private AtomicGENXIntVector set = new AtomicGENXIntVector(100000);
    private AtomicGENXByteVector states = new AtomicGENXByteVector(100000);
    private AtomicGENXInteger free = new AtomicGENXInteger(states.value.length);
    private AtomicGENXInteger size = new AtomicGENXInteger(0);
    private AtomicGENXInteger maxSize = new AtomicGENXInteger(states.value.length / 2);

    private static final byte FREE = 0;
    private static final byte FULL = 1;
    private static final byte REMOVED = 2;

  private static final int USED_FREE = 0 ;
  private static final int USED_FILED = 1 ;
  private static final int FAILED = 2 ;

  public boolean add(int value) {
    int added = new_add(value) ;
    if(added!=FAILED)
      postInsertHook(added==FREE);

    return added!=FAILED ;
  }

  public int new_add(int value) {
        RWResourceGroup rg = new RWResourceGroup();
        rg.addRead(set);
        int index = findIndex(value);
        if (index < 0) {
      int returnExpression = FAILED;
            rg.unlock();
            return returnExpression;
        }
    rg.add(set.value[index]);
        rg.lock();
        byte previousState = states.value[index].value;
        set.value[index].value = value;
        states.value[index].value = FULL;
    int returnExpression = previousState == FREE ? USED_FREE : USED_FILED;
        rg.unlock();
        return returnExpression;
    }

    public boolean contains(int value) {
        return index(value) >= 0;
    }

    public boolean remove(int value) {
        RWResourceGroup rg = new RWResourceGroup();
        rg.addRead(set);
    int index = index(value);
    rg.lock();
        if (index >= 0) {
      rg.add(set.value[index]);
      rg.add(states.value[index]);
            set.value[index].value = (int)0;
            states.value[index].value = REMOVED;
            size.value--;
            boolean returnExpression = true;
            rg.unlock();
            return returnExpression;
        }
        boolean returnExpression = false;
        rg.unlock();
        return returnExpression;
    }

    private int index(int value) {
        RWResourceGroup rg = new RWResourceGroup();
        rg.addRead(set);
        rg.lock();
        int length = states.value.length;
        int hash = (value * 31) & 2147483647;
        int index = hash % length;
        int probe;
        if (states.value[index].value != FREE && (states.value[index].value ==
            REMOVED || set.value[index].value != value)) {
            probe = 1 + (hash % (length - 2));
            do {
                index -= probe;
                if (index < 0) {
                    index += length;
                }
            }
      while (states.value[index].value != FREE && (states.value[index].value ==
          REMOVED || set.value[index].value != value));
        }
        int returnExpression = states.value[index].value == FREE ? -1 : index;
        rg.unlock();
        return returnExpression;
    }

    int findIndex(int value) {
        int length = states.value.length;
        int hash = ((value * 31) & 2147483647);
        int index = hash % length;
        int probe;
        if (states.value[index].value == FREE) {
            return index;
        } else {
            if (states.value[index].value == FULL && set.value[index].value == value) {
                return -index - 1;
            } else {
                probe = 1 + (hash % (length - 2));
                if (states.value[index].value != REMOVED) {
                    do {
                        index -= probe;
                        if (index < 0) {
                            index += length;
                        }
                    }                     while (states.value[index].value == FULL && set.value[index].value != value);
                }
                if (states.value[index].value == REMOVED) {
                    int firstRemoved = index;
                    while (states.value[index].value != FREE && (states.value[index].value == REMOVED || set.value[index].value != value)) {
                        index -= probe;
                        if (index < 0) {
                            index += length;
                        }
                    }
                    return states.value[index].value == FULL ? -index - 1 : firstRemoved;
                }
                return states.value[index].value == FULL ? -index - 1 : index;
            }
        }
    }

    /**
     * After an insert, this hook is called to adjust the size.value/free
     * values of the set.value and to perform rehashing if necessary.
     */
    protected final void postInsertHook(boolean usedFreeSlot) {
    if (usedFreeSlot) {
      free.value--;
    }
        if (++size.value > maxSize.value || free.value == 0) {
            int newCapacity = size.value > maxSize.value ? PrimeFinder.nextPrime(states.value.length << 1) : states.value.length;
            rehash(newCapacity);
            maxSize.value = states.value.length / 2;
        }
    }

    protected void rehash(int newCapacity) {
    RWResourceGroup rg = new RWResourceGroup() ;
        int oldCapacity = set.value.length;
        AtomicGENXInteger[] oldSet = set.value;
    AtomicGENXByte[] oldStates = states.value;
        set = new AtomicGENXIntVector(newCapacity);
        states = new AtomicGENXByteVector(newCapacity);
    rg.add(set);
    rg.lock();
        for (int i = oldCapacity; i-- > 0; ) {
            if (oldStates[i].value == FULL) {
                int o = oldSet[i].value;
                int index = findIndex(o);
                set.value[index].value = o;
                states.value[index].value = FULL;
            }
        }
    rg.unlock();
    }
}
```

Listing B.4: Generated `IntSetHash` class of the *hash set* benchmark, *tweaked* for performance.

```
package tests.microbenchmarksRW.intset;

import rgs.RWResourceGroup;
import java.util.ArrayList;
import java.util.List;

/**
 * @author Pascal Felber
 * @since 0.1
 */
public class IntSetLinkedList implements IntSet {

  private static final long serialVersionUID = 1L;
  private AtomicGENNode m_first = new AtomicGENNode(0);

  public IntSetLinkedList() {
    RWResourceGroup rg = new RWResourceGroup();
    rg.add(m_first);
    AtomicGENNode min = new AtomicGENNode(Integer.MIN_VALUE);
    rg.add(min);
    AtomicGENNode max = new AtomicGENNode(Integer.MAX_VALUE);
    rg.add(max);
    rg.lock();
    min.setNext(max);
    m_first = min;
    rg.unlock();
  }

  public boolean add(int value) {
    RWResourceGroup rg = new RWResourceGroup();
    rg.add(m_first);
    boolean result;
    rg.lock();
    AtomicGENNode previous = m_first;
    AtomicGENNode next = previous.getNext();
    int v;
    while ((v = next.getValue()) < value) {
      previous = next;
      next = previous.getNext();
    }
    result = v != value;
    if (result) {
      previous.setNext(new AtomicGENNode(value, next));
    }
    boolean returnExpression = result;
    rg.unlock();
    return returnExpression;
  }

  public boolean remove(int value) {
    RWResourceGroup rg = new RWResourceGroup();
    rg.add(m_first);
    boolean result;
    rg.lock();
    AtomicGENNode previous = m_first;
    AtomicGENNode next = previous.getNext();
    int v;
    while ((v = next.getValue()) < value) {
      previous = next;
      next = previous.getNext();
    }
    result = v == value;
    if (result) {
      previous.setNext(next.getNext());
    }
    boolean returnExpression = result;
    rg.unlock();
    return returnExpression;
  }

  public boolean contains(int value) {
    RWResourceGroup rg = new RWResourceGroup();
    rg.add(m_first);
    boolean result;
    rg.lock();
    AtomicGENNode previous = m_first;
    AtomicGENNode next = previous.getNext();
    int v;
    while ((v = next.getValue()) < value) {
      previous = next;
      next = previous.getNext();
    }
    result = (v == value);
    boolean returnExpression = result;
    rg.unlock();
    return returnExpression;
  }
}
```

Listing B.5: Generated `IntSetLinkedList` class of the *linked list* benchmark.

```
package tests.microbenchmarksRW.intset;

import rgs.RWResourceGroup;

/**
 * @author Pascal Felber
 * @since 0.1
 */
public class IntSetLinkedList_No_Partition implements IntSet {

  private static final long serialVersionUID = 1L;
  private AtomicGENNode m_first = new AtomicGENNode(0);

  public IntSetLinkedList_No_Partition() {
    AtomicGENNode min = new AtomicGENNode(Integer.MIN_VALUE);
    AtomicGENNode max = new AtomicGENNode(Integer.MAX_VALUE);
    min.setNext(max);
    m_first = min;
  }

  public boolean add(int value) {
    RWResourceGroup rg = new RWResourceGroup();
    boolean result;
    AtomicGENNode previous = m_first;
    rg.add(previous);
    rg.lock();
    AtomicGENNode next = previous.getNext();
    rg.add(next);
    int v;
    while ((v = next.getValue()) < value) {
      rg.remove(previous);
      previous = next;
      next = previous.getNext();
      rg.add(next) ;
    }
    result = v != value;
    if (result) {
      previous.setNext(new AtomicGENNode(value, next));
    }
    boolean returnExpression = result;
    rg.unlock();
    return returnExpression;
  }

  public boolean remove(int value) {
    RWResourceGroup rg = new RWResourceGroup();
    boolean result;
    AtomicGENNode previous = m_first;
    rg.add(previous);
    rg.lock();
    AtomicGENNode next = previous.getNext();
    rg.add(next);
    int v;
    while ((v = next.getValue()) < value) {
      AtomicGENNode old_previous = previous ;
      previous = next;
      rg.remove(old_previous);
      next = previous.getNext();
      rg.add(next) ;
    }
    result = v == value;
    if (result) {
      previous.setNext(next.getNext());
    }
    boolean returnExpression = result;
    rg.unlock();
    return returnExpression;
  }

  public boolean contains(int value) {
```

```java
    RWResourceGroup rg = new RWResourceGroup();
    boolean result;
    AtomicGENNode previous = m_first;
    rg.add(previous);
    rg.lock();
    AtomicGENNode next = previous.getNext();
    rg.add(next);
    int v;
    while ((v = next.getValue()) < value) {
      AtomicGENNode old_previous = previous ;
      previous = next;
      rg.remove(old_previous);
      next = previous.getNext();
      rg.add(next) ;
    }
    result = (v == value);
    boolean returnExpression = result;
    rg.unlock();
    return returnExpression;
  }
}
```

Listing B.6: Generated `IntSetLinkedList` class of the *linked list* benchmark if the partition was ignored.

```java
package tests.microbenchmarksRW.intset;

import rgs.RWResourceGroup;

/**
 * Created with IntelliJ IDEA.
 * User: MetaBrain
 * Date: 18-06-2013
 * Time: 10:15
 *
 * @autor metabrain (https://github.com/metabrain) - Daniel Parreira 2013
 */
public class Node {

  /**
   *
   */
  private static final long serialVersionUID = 1L;
  private final int m_value;
  private AtomicGENNode m_next;

  public Node(int value, AtomicGENNode next) {
//    RWResourceGroup rg = new RWResourceGroup();
//    rg.add(m_next);
    m_value = value;
//    rg.lock();
    m_next = next;
//    rg.unlock();
  }

  public Node(int value) {
    this(value, null);
  }

  public int getValue() {
    return m_value;
  }

  public void setNext(AtomicGENNode next) {
    RWResourceGroup rg = new RWResourceGroup();
    rg.add(m_next);
    rg.lock();
    m_next = next;
    rg.unlock();
  }

  public AtomicGENNode getNext() {
    m_next.lockRead();
    tests.microbenchmarksRW.intset.AtomicGENNode returnExpression = m_next;
    m_next.unlockRead();
    return returnExpression;
  }
}
```

Listing B.7: Generated `Node` class of the *linked list* benchmark.