Ricardo Jorge Freire Dias

Mestre em Engenharia Informática

Maintaining the Correctness of Transactional Memory Programs

Dissertação para obtenção do Grau de Doutor em
Engenharia Informática

Orientador : João Manuel dos Santos Lourenço,
Prof. Auxiliar, Universidade Nova de Lisboa

Júri:

Presidente:	José Legatheaux Martins
Arguentes:	Timothy L. Harris
	João M. Cachopo
Vogais:	José Cardoso e Cunha
	Luís E. Rodrigues
	Rui C. Oliveira
	João Costa Seco
	João M. Lourenço

FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

Novembro, 2013

Maintaining the Correctness of Transactional Memory Programs

To my wife and children
Ana, Diogo and Vasco

Acknowledgements

The work presented in this dissertation would not have been possible without the collaboration of a considerable number of people to whom I would like to express my gratitude.

First and foremost I would like to deeply thank my thesis advisor João Lourenço, who always helped me to overcome all the challenges that I had to face during my PhD studies. I've learned a lot from him, and I truly hope to be able to advise my students as well as he advised me.

To João Seco for his guidance in an area of research which was completely new to me, and for our discussions about the theoretical parts of my work. To Nuno Preguiça for always being available to discuss my ideas and give excellent tips and suggestions which allowed me to improve the quality of my work. To Dino Distefano for hosting me at the computer science department of the Queen Mary, University of London, for teaching me everything I know about separation logic, and for his insightful suggestions to solve some hard problems that emerged during the development of this thesis.

To Tiago Vale for all the collaboration in the implementation and testing in parts of my experimental work, and for all the fruitful discussions we had. To Vasco Pessanha for his help in the designing and implementation of the MoTH tool and its functionalities. To João Leitão for reviewing the last draft of this thesis.

To all my colleagues with whom I shared the ASC open space, including: Valter Balegas, Lamia Benmouffok, Jorge Custódio, Bernardo Ferreira, João Luis, David Navalho, Daniel Porto, Paulo Quaresma, João Soares, and Tiago Vale.

Finally, my very heartful thanks to my family. To my wife Ana, whom I owe everything of good in my life, including my beautiful children Diogo and Vasco. I stole too much of the time that was rightfully theirs in these past years. To my parents that although being far away always give the support that I need. To my parents-in-law for all the tasteful lunches and dinners, and for the comfort they provided while I was living at their home.

I also would like to acknowledge the following institutions for their hosting and financial support: Departamento de Informática and Faculdade de Ciências e Tecnologia of the Universidade Nova de Lisboa; Centro de Informática e Tecnologias da Informação of the FCT/UNL; Fundação para a Ciência e Tecnologia in the PhD research grant SFRH/BD/41765/2007, and in the research projects Synergy-VM (PTDC/EIA-EIA/113613/2009), and RepComp (PTDC/EIA-EIA/108963/2008).

Abstract

This dissertation addresses the challenge of maintaining the correctness of transactional memory programs, while improving its parallelism with small transactions and relaxed isolation levels.

The efficiency of the transactional memory systems depends directly on the level of parallelism, which in turn depends on the conflict rate. A high conflict rate between memory transactions can be addressed by reducing the scope of transactions, but this approach may turn the application prone to the occurrence of atomicity violations. Another way to address this issue is to ignore some of the conflicts by using a relaxed isolation level, such as snapshot isolation, at the cost of introducing write-skews serialization anomalies that break the consistency guarantees provided by a stronger consistency property, such as opacity.

In order to tackle the correctness issues raised by the atomicity violations and the write-skew anomalies, we propose two static analysis techniques: one based in a novel static analysis algorithm that works on a dependency graph of program variables and detects atomicity violations; and a second one based in a shape analysis technique supported by separation logic augmented with heap path expressions, a novel representation based on sequences of heap dereferences that certifies if a transactional memory program executing under snapshot isolation is free from write-skew anomalies.

The evaluation of the runtime execution of a transactional memory algorithm using snapshot isolation requires a framework that allows an efficient implementation of a multi-version algorithm and, at the same time, enables its comparison with other existing transactional memory algorithms. In the Java programming language there was no framework satisfying both these requirements. Hence, we extended an existing software transactional memory framework that already supported efficient implementations of some transactional memory algorithms, to also support the efficient implementation of multi-version algorithms. The key insight for this extension is the support for storing the transactional metadata adjacent to memory locations. We illustrate the benefits of our approach by analyzing its impact with both single- and multi-version transactional memory algorithms using several transactional workloads.

Keywords: Concurrent Programming, Transactional Memory, Snapshot Isolation, Static Analysis, Separation Logic, Abstract Interpretation

Resumo

Esta dissertação aborda o desafio de manter a correcção dos programas de memória transacional, quando são usadas pequenas transacções e níveis de isolamento relaxado para melhorar o paralelismo dos programas.

A eficiência dos sistemas de memória transacional depende directamente do nível de paralelismo, o que por sua vez depende da taxa de conflitos. Uma elevada taxa de conflitos entre as transações em memória pode ser diminuida através da redução do tamanho das transações, mas esta abordagem poderá tornar a aplicação propensa à ocorrência de violações de atomicidade. Outra forma de abordar esta questão é ignorar alguns dos conflitos usando um nível de isolamento relaxado, tal como o nível de isolamento *snapshot isolation*, com o custo da introdução de anomalias de serialização, denominados como *write-skews*, que quebram a consistência garantida por uma propriedade de consistência forte, como a opacidade.

Com o intuito de abordar as questões levantadas pela correção das violações de atomicidade e das anomalias de *write-skews*, propomos duas técnicas de análise estática: uma baseada num novo algoritmo de análise estática que utiliza um grafo de dependências entre variáveis de programa para detectar violações de atomicidade; e uma segunda com base numa técnica de análise baseada em lógica de separação extendida com expressões de *heap paths*, uma nova representação baseada em sequências de desreferênciações da memória, que certifica se um programa que usa memória transacional, baseada em *snapshot isolation*, está livre de anomalias write-skew durante a execução.

A avaliação da execução de um algoritmo de memória transacional usando o nível de isolamento relaxado *snapshot isolation* requer uma estrutura que permita a implementação eficiente de algoritmos baseados em multi-versão e, ao mesmo tempo, permitir a sua comparação com outros algoritmos de memória transacional existentes. Na linguagem de programação Java não existe uma ferramenta que satisfaça estes dois requisitos. Como tal, estendemos uma ferramenta de memória transacional por software já existente, que já permite a implementação eficiente de alguns algoritmos de memória transacional, a também permitir a implementação eficiente de algoritmos multi-versão. A principal ideia desta extensão é o suporte para armazenar os metadados transacionais junto às localizações de memória. Nós ilustramos os benefícios da nossa abordagem, analisando o seu impacto tanto com algoritmos de memória transacional uni-versão como multi-versão utilizando vários tipos de testes transacionais.

Palavras-chave: Programação Concorrente, Memória Transacional, *Snapshot Isolation*, Análise Estática, Lógica de Separação, Interpretação Abstrata

Contents

List of Figures

List of Tables

1

Introduction

Gordon Moore, back in 1965, observed that the number of transistors per square inch doubles every 18 months, and the rate of growth has been relatively steady since then. As the number of transistors was growing, the processor clock frequency grew along. However, since the appearance of CPUs (central processor units) with clock frequencies in the order of the gigahertzs, the growth of the clock frequency slowed down, even though the number of transistors is still rising at a steady pace. The CPU manufacturers opted to use the additional transistors by designing processors with more than one operational core, leading to the current widespread of multi-core architectures.

In the past, performance improvements had a strong dependency on the increasing of processor speed, unfortunately, processor speed has stabilized. The multi-core architectures are currently ubiquitously available, from industrial to home computers and embedded devices, and the need to exploit their full computational power raised considerably the interest in the discipline of parallel programming.

Leveraging parallelism in multi-threaded programs requires synchronization constructs to control accesses to shared resources, such as main memory. From a programmer point of view, current synchronization constructs (locks, monitors, and condition variables) require a great effort to be used correctly and achieve high scalability at the same time [LPSZ08]. The use of coarse grained locks in large data structures hinders parallelism and does not scale, while fine grained locks are prone to many difficult problems in large systems, such as priority inversion, convoying, and most specially, deadlocks.

Transactional Memory (TM) [ST95; HLMWNS03] is a synchronization technique that aims at solving the inherent pitfalls associated with the use of locks. It promises to ease the development of scalable parallel applications with performance close to finer grain locking but with the simplicity of coarse grain locking. A memory transaction, borrows the concept of transaction from the database world, but instead manages concurrent accesses to main memory. A database transaction is a unit of work that executes several operations while providing the four ACID properties:

atomicity, consistency, isolation, and durability. A memory transaction only provides three of these properties: atomicity, consistency, and isolation. Durability is dropped due to the nature of the storage medium.

Transactional memory runtime systems usually adopt an optimistic execution model, where transactions execute concurrently and conflicts are solved by a contention management algorithm, which can be as simple as aborting one of the conflicting transactions. The assertion of when two TM transactions conflict is algorithm dependent, but usually the conflict detection depends on the TM system keeping track of the memory locations accessed during the transactions lifetime, and on the validation of all those accesses during the execution of the transaction and/or at commit time.

The level of parallelism allowed by a transactional memory system depends directly on the conflict occurrence rate. A high rate of conflicts forces more transactions to abort and reduces the overall transactional throughput. Furthermore, the conflict rate depends on both the size of the transaction and on the level of permissiveness of the transactional system. Depending on the kind of workload, coarse-grain transactions may increase the probability of conflicts, which in turn reduces the system performance. The permissiveness level controls the kind of conflicts that are allowed, thereby ignored, by the transactional system without losing the desired consistency property. In the case of transactional memory the desired consistency property is *opacity* [GK08].

In some situations it is possible to reduce the number of conflicts by reducing the size of the transactions, splitting the large transactions into a sequence of smaller ones. Opposed to the use of finer-grain locks, which can easily lead to deadlocks, memory transactions never cause deadlocks. However, the use of finer-grain transactions may still lead to other concurrency bugs know as atomicity violations [LPSZ08], and thus we may expect that reducing the size of the transactions in an application may compromise its correctness. Additionally, we can also increase the permissiveness level of the transactional runtime by allowing the occurrence of some conflicts. This can be achieved by relaxing the isolation level, but at the cost of losing opacity.

In this dissertation we will address these two main problems: how to ensure the correct usage of finer-grain transactions by avoiding atomicity violations, and how to increase transactional memory performance by relaxing the isolation level without losing correctness.

1.1 Problem Statement

We argue that it is possible to develop a set of solutions that enable an increased parallelism of transactional memory without losing correctness. More specifically, we propose to address this problem by allowing the safe usage of finer-grain transactions with the avoidance of atomicity violations, and by the use of a relaxed isolation level without losing a stricter consistency property, such as opacity.

In summary the work presented in this thesis aims at demonstrate the truth of the following thesis statement:

Thesis Statement

It is possible to maintain the correctness of transactional memory programs, while improving its parallelism with small transactions and relaxed isolation levels.

```
1  atomic void getA() {
2    return pair.a;
3  }
4  atomic void getB() {
5    return pair.b;
6  }
7  atomic void setPair(int a, int b){
8    pair.a = a;
9    pair.b = b;
10 }
11 boolean areEqual(){
12   int a = getA();
13   int b = getB();
14   return a == b;
15 }
```

(a) A high-level data race.

```
1  atomic int getX() {
2    return x;
3  }
4  atomic void setX(int p0) {
5    x = p0;
6  }
7  void incX(int val) {
8    int tmp = getX();
9    tmp = tmp + val;
10   setX(tmp);
11 }
```

(b) A stale value error.

Figure 1.1: Example of atomicity violations.

In the following we present a brief overview of the main techniques used to corroborate this thesis statement.

Detection of atomicity violations Although using transactional memory is much simpler than using fine-grain locking, the use of finer-grain transactions may still introduce concurrency bugs known as atomicity violations.

High-level data races are a form of atomicity violations and result from the misspecification of the scope of an atomic block, which is split into two or more atomic blocks with other (possibly empty) non-atomic block between them. This anomaly is illustrated in Figure 1.1(a). In this example a thread uses the method `areEqual()` to check if the fields `a` and `b` are equal. This method reads both fields in separate atomic blocks, storing their values in local variables, which are then compared. The atomicity violation results from the interleaving of this thread with another thread running the method `setPair()`. If the method `setPair()` is executed between lines 12 and 13 of the method `areEqual()`, when `areEqual()` is resumed at line 13 the value of the pair may have changed. In this scenario the thread executing `areEqual()` observes an inconsistent pair, composed by the old value of `a` and the new value of `b`.

Figure 1.1(b) illustrates a stale value error, another source of atomicity violations in concurrent programs. The non-atomic method `incX()` is implemented by resorting to two atomic methods, `getX()` (at line 1) and `setX()` (at line 4). During the execution of line 9, if the current thread is suspended and another thread is scheduled to execute `setX()`, the value of `x` changes, and when the execution of the initial thread is resumed it overwrites the value in `x` at line 10, causing a lost update. This program fails due to a stale-value error, as at line 8 the value of `x` escapes the scope of the atomic method `getX()` and is reused indirectly (by way of its private copy `tmp`) at line 10, when updating the value of `x` in `setX()`.

The early detection of these kind of anomalous interleavings in the development phase of the application is crucial to avoid runtime bugs that are very hard to find and to debug. To address this challenge, the dissertation will focus on the following question:

> *Is it possible to develop a tool capable of detecting atomicity violations in transactional memory programs, at compile-time, with high precision and scalability?*

```
void withdrawX(int amount) {
  atomic {
    if (accountX + accountY > amount)
      accountX -= amount;
  }
}
```

```
BEGIN withdrawX
  READ(accountX)
  READ(accountY)
  WRITE(accountX)
COMMIT
```

```
void withdrawY(int amount) {
  atomic {
    if (accountX + accountY > amount)
      accountY -= amount;
  }
}
```

```
BEGIN withdrawY
  READ(accountX)
  READ(accountY)
  WRITE(accountY)
COMMIT
```

Figure 1.2: Write-skew example

We propose a novel approach for the detection of high-level data races and stale-value errors in transactional memory programs. The approach is based on a novel notion of variable dependencies, which we designate as *causal* dependencies. There is a *causal* dependency between two variables if the value of one of them influences the writing of the other. We also extended previous work from Artho et al. [AHB03] by reflecting the read/write nature of accesses to shared variables inside atomic regions, which we combine with the dependencies information to detect both high-level data races and stale-value errors. We formally describe the static analysis algorithms to compute the set of *causal* dependencies of a program and define safety conditions for both high-level data races and stale-value errors. The matter of detecting atomicity violations in TM programs is addressed in Chapter 3.

Consistent relaxed isolation level To solve the problem of relaxing the isolation level to increase transactional parallelism, we took inspiration from the database setting. Database systems frequently rely on weaker isolation models to improve performance. In particular, Snapshot Isolation (SI) is widely used in industry. An interesting aspect of SI is that only write-write conflicts are checked at commit time and considered for detecting conflicting transactions. As a main result, a TM system using this isolation model does not need to keep track of read accesses, thus considerably reducing the book-keeping overhead.

By only detecting write-write conflicts, and ignoring read-write conflicts, the SI model allows a much higher commit rate, which comes at the expense of allowing some real conflicting transactions to commit. Thus, relaxing the isolation level of a transactional program to SI may lead previously correct programs to misbehave due to the anomalies resulting from malign data-races that are now allowed by the relaxed transactional runtime. These anomalies can be precisely characterized, and are often called in the literature as write-skew anomalies [BBGMOO95].

In Figure 1.2 we show an example of two concurrent transactions that trigger a write-skew anomaly. These two transactions are originated from the execution of the two methods `withdrawX` and `withdrawY`. The presence of the write-skew is due to the fact that both committed transactions read a data item written by the other (`withdrawX` reads `accountY` written by `withdrawY`, and `withdrawY` reads `accountX` written by `withdrawX`), and both write in different data items. If we invoke methods `withdrawX` and `withdrawY` with the arguments `30` and `40` respectively, where the shared state is defined by `accountX = 40` and `accountY = 20`, the result of these two concurrent transactions, under snapshot isolation, would cause an inconsistent state where `accountX = 10` and `accountY = -20`, which is impossible to obtain in a serializable execution of those transactions.

A possible approach to solve the problem of identifying the write-skew anomalies in a program running under SI could be to give the programmer the burden of this task. However, this task could be overwhelming for the average programmer, the development would be very costly and error prone, and hardly worth the performance benefits. In the database setting, a different approach was followed where several algorithms were proposed to dynamically avoid write-skew anomalies, and hence provide a serializable model, while maintaining similar performance of snapshot isolation. Although this solution was well succeeded in databases, the application of such dynamic algorithms in a TM setting is not a viable option due to the significant overhead introduced at runtime, which is exactly the opposite of our objective of reducing the TM runtime overhead.

Another possible way to address the matter of correctness of TM programs executing in TM runtimes using SI, is to assert at compile-time, using static analysis techniques, that a TM program will execute without generating write-skew anomalies. This approach avoids the runtime overhead imposed by dynamic algorithms, and provide opacity guarantees to the programmer by asserting that the computations are free from write-skew anomalies.

To address this problem, the dissertation will focus on the following question:

> *Is it possible to develop a verification procedure to identify write-skew anomalies in programs written using an imperative language with support for dynamically allocated (heap) memory?*

To address this specific question, we propose a technique that performs deep-heap analysis (also called shape analysis) based on separation logic [Rey02] to approximate the memory locations in the read- and write- sets for each distinguished transaction in a program. The analysis only requires the specification of the state of the heap for each transaction and is able to automatically compute loop invariants during the analysis. Our analysis approximates read and write-sets of transactions using heap paths: a regular expression based representation that captures dereferences through field labels, choice, and repetition.

For those conflicting transactions that are prone to trigger anomalies, there are different strategies [FLOOS05] that can be applied to correct the runtime behavior of such transactions making them correct under SI. For instance, it is possible to modify the transaction code, or to execute the transaction in a more strict isolation level.

Using this approach, we achieve improved performance by relying on a less expensive snapshot isolation-based TM runtime, while guaranteeing correctness of program execution by avoiding write-skews and keeping opacity. Because it is based on static-analysis techniques, our approach introduces no runtime overhead. Our approach to detect write-skew anomalies for a general-purpose language is described in Chapter 4.

SI performance evaluation In terms of performance, the evaluation of our approach requires a fair comparison with other *opaque* TM algorithms. The implementation or adaptation of several TM algorithms to use the same transactional interface, and to work with the same benchmark code, is an unfeasible task. To solve this problem, generic and extensible frameworks were developed, allowing the implementation of different TM algorithms by following a well defined interface that captures the essential steps performed by a memory transaction (e.g. transaction start, memory read, memory write, transaction commit, and transaction abort).

The problem with the generic frameworks is that they are usually biased to some specific implementation techniques which only fit well to some TM algorithms. This implies that the

comparison of two different TM algorithms, where one of them is *unfit* for the specific framework, cannot be made in a straightforward process and the obtained results will be biased towards the algorithm that better fits the framework.

The *unfitness* problem between the TM algorithm and the generic framework may be caused by the management of transactional runtime information required by the TM algorithm. TM algorithms manage information per transaction (frequently referred to as a *transaction descriptor*), and per memory location (or object reference) accessed within that transaction. The transaction descriptor is typically stored in a thread-local memory space and maintains the information required to validate and commit the transaction. The per memory location information depends on the nature of the TM algorithm, which we will henceforth refer to as *metadata*, and may be composed by e.g. locks, timestamps or version lists. Metadata is stored either "near" each memory location (*in-place* strategy), or in an external table that associates the metadata with the corresponding memory location (*out-place* or *external* strategy).

TM libraries targeting imperative languages, such as C, frequently use an out-place strategy, while those targeting object-oriented languages bias towards the in-place strategy. The out-place strategy is implemented by using a table-like data-structure that efficiently maps memory references to its metadata. Storing the metadata in a pre-allocated table avoids the overhead of dynamic memory allocation, but incurs in overhead for evaluating the location-metadata mapping function, and has limitations imposed by the size of the table. The in-place strategy is usually implemented by using the *decorator* design pattern [GHJV94] that is used to extend the functionality of an original class by wrapping it in a *decorator* class, which also contains the required metadata. This technique allows the direct access to the object metadata without significant overhead, but is very intrusive to the application code, which must be deeply rewritten to use the decorator classes. This *decorator* pattern based technique also incurs in two additional problems: some additional overhead for non-transactional code, and multiple difficulties to cope with primitive and array types. A brief discussion of the tradeoffs of using in-place *versus* out-place strategies is presented in [RB08].

An efficient technique to implement a snapshot isolation based TM algorithm is to use multi-versioning. In a multi-version algorithm, several versions of a data item exist. In the particular case of transactional memory, several versions of the same memory block exist. The implementation of a multi-version algorithm is not tolerant to the false-sharing introduced by the mapping-table approach, and is more adequate to an in-place strategy, which associate the list of versions to its respective memory block in a one-to-one relation, i.e., without false-sharing. As there was no single generic TM framework supporting efficiently both the in-place and the out-place strategies, it was inviable to compare a snapshot isolation TM algorithm (which requires the in-place strategy) with other kinds of opaque TM algorithms that use the out-place strategy. To tackle this problem, this dissertation will also address the following question:

> *Is it possible to build a generic and extensible runtime infrastructure for software transactional memory that fits equally well for both the in-place and the out-place algorithm implementation strategies?*

We address this particular question by extending a well known and very efficient Java STM framework, the Deuce [KSF10], which is biased towards the out-place strategy, to additionally support the in-place strategy as well. Our extension allows the efficient implementation of multi-version TM algorithms, and in particular allows the implementation of a snapshot isolation based

TM algorithm. We implemented a simple SI algorithm and compared it against different state-of-the-art TM algorithms already available in the Deuce framework. Some of the benchmarks were successfully certified as write-skew free by our static analysis technique. Others were not possible to certify due to limitations of the analysis in terms of data structures support or because of time scalability problems. The matter of supporting the in-place strategy in Deuce is addressed in Chapter 5.

1.2 Contributions and Results

This dissertation presents contributions to the state of the art on three major challenges:

a) Verification of atomicity violations in transactional memory programs.

- Definition of a novel notion of *causal dependencies* between program variables, which unifies the data-flow and control-flow relation between variables;
- Refinement of existing high-level data-races and stale-value errors definition to incorporate causal dependencies information;
- An implementation of our technique in a tool, called MoTH, the application of the tool to a set of well known faulty examples from the literature, and its comparison with previous works.

b) Verification of write-skew anomalies in transactional memory programs.

- The first program verification technique to statically detect the write-skew anomaly in transactional memory programs;
- The first technique able to verify transactional memory programs in the presence of deep-heap manipulation, thanks to the use of shape analysis techniques;
- A model that captures fine-grained manipulation of memory locations based on heap paths;
- An implementation of our technique and the application of the tool to a set of intricate examples.

c) Development of a generic and extensible runtime infrastructure for Java software transactional memory with support for efficient implementations of multi-version algorithms.

- Extension of Deuce to support in-place transactional metadata;
- Comparative analysis of multi-version algorithms using in-place metadata support;
- Proposal of a new multi-version algorithm with bound sized version lists and per write-set entry locking;
- Definition of a new algorithmic adaptation for multi-version algorithms to support weak-atomicity.

1.3 Outline of the Dissertation

The remainder of this dissertation is organized in five chapters, whose contents are summarized below:

Chapter 2. This chapter introduces the fundamental concepts to clearly understand the following chapters. It also presents the state-of-the-art of the techniques and tools related to the matters addressed by this dissertation.

Chapter 3. This chapter describes a new static analysis technique to detect atomicity violations. This novel approach to detect high-level data races and stale-value errors relies on the notion of *causal dependencies* to improve the precision of previous detection techniques. We formalize the analysis technique to compute the causal dependencies of a program, and formalize the refinement of existing safety conditions, for high-level data races and stale-value errors, using the causal dependencies. Finally, we describe the implementation of these techniques in a tool to verify Java bytecode, and evaluate its precision with well known examples from the literature.

Chapter 4. This chapter presents the design and development of a static analysis technique to verify if a concurrent program, which uses snapshot isolation based memory transactions, is free from the occurrences of write-skew anomalies. We define the notion of heap path expressions, present their semantics, and define their construction from separation logic formulas. We formalize the analysis abstract domain, composed by symbolic heaps and sets of heap path expressions, and abstract semantics. We finish with the experimental evaluation of the proposed technique.

Chapter 5. This chapter presents the design and implementation of an extension to the Deuce framework to support in-place transactional metadata, i.e., the co-location of transactional metadata near the memory locations instead of in a shared external mapping table. We describe in detail the technique used to implement the extension and thoroughly evaluate its performance and memory overhead. We also present the implementation of two state-of-the-art multi-version algorithms in the extended framework, and preform the first evaluation comparing multi-version algorithms within the same framework.

Chapter 6. This chapter summarizes the main results and contributions of the research work described in this dissertation, and lists some directions for future research activities.

2

Fundamental Concepts and State of the Art

In this chapter we present the research context for the theme of this dissertation. We start by describe the fundamental concepts of (software) transactional memory and present two extensible STM frameworks that allow to experiment new STM algorithms for the Java programming language. Furthermore, we describe the state of the art in detection of atomicity violations, and describe throughly the concept of snapshot isolation and its known anomalies based on the seminal work of Fekete et al. [FLOOS05]. Finally, we present the current techniques of static analysis based on the abstract interpretation framework with special emphasis on shape analysis using separation logic, i.e., static analysis of heap structures.

2.1 Transactional Memory

In 1977 Lomet [Lom77] explored the idea of including an *atomic* action as a method for program structuring, based on the idea of database atomic actions. Many years later in 1993 Herlihy et al. [HM93] introduced, for the first time, the terminology *Transactional Memory* for describing a hardware architecture to optimize the efficiency and usability of lock-free synchronization. In 1995, Nir Shavit et al. [ST95] proposed a software approach to transactional memory, calling it *Software Transactional Memory*.

Software transactional memory (STM) is a promising concurrency control approach to multithreaded programming. More than a concurrency control mechanism, it is a new programming model that brings the concept of transactions into the programming languages, by way of new language constructs or as a simple API with a supporting library. Transactions are widely known as a technique that ensure the four ACID properties [GR92]: Atomicity (A), Consistency (C), Isolation (I) and Durability (D).

Memory transactions, with roots in the database transactions, must only ensure three of the ACID properties: Atomicity, Consistency and Isolation. The Durability property is dropped, as memory transactions operate in volatile memory (RAM).

2.1.1 Semantics

The first step to study software transactional memory is to understand its execution behavior. Informally, a memory transaction is a group of read and write operations that will execute in a single step, or atomically. Thus, conceptually, no two memory transactions will ever execute at the same time. Two memory transactions are depicted in Figure 2.1. Transaction **T1** increments two variables `x` and `y`. Transaction **T2** compares the values of the two variables, `x` and `y`, and if the two variables have different values the transaction will enter in an infinite loop. Before any of these transactions execute, the variables, `x` and `y`, have the same value. As a transaction executes in a single step there are only two possible outcomes of the execution of these two transactions: either **T1** executes before **T2**, or **T1** executes after **T2**. Therefore, transaction **T2** will never enter in an infinite loop, because it will never be interleaved with **T1**.

Transaction 1 (**T1**)

```
1 atomic {
2   x = x+1;
3   y = y+1;
4 }
```

Transaction 2 (**T2**)

```
1 atomic {
2   while(x != y) {
3     // infinite loop
4   }
5 }
```

Figure 2.1: Example of two memory transactions.

A simple implementation of this behavior would be to use a global lock, and before a transaction starts its execution it has to acquire the global lock, releasing it at end of the transaction. This guarantees that only one transaction will execute at a time. The semantics just described is called *Single Global Lock Semantics*[MBSATHSW08].

More formal definitions are used to describe the semantics of memory transactions. Inherited from the databases literature, the *serialization* criteria [EGLT76] defines formally the semantics of database transactions, and can be used also to define the semantics of memory transactions. The serialization theory states that the result of a parallel execution of a program with transactions must be equivalent to a sequential execution of all transactions.

Another criteria commonly used to describe concurrent shared objects and sometimes used as a correction criteria for transactional memory is *linearizability* [HW90]. This criteria states that every transaction should appear as if it took place at some single, unique point in time during its lifespan.

Although serializability suites very well for database transactions, and linearizability for concurrent shared objects, none of them is sufficient to clearly define the semantics of a memory transaction [GK08]. Guerraoui et al., in [GK08], defines a new criteria called *opacity*. This criteria fits well for the transactional memory model, as it takes into account the property of memory consistency during the execution of a transaction. The informal definition of this criteria is that all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime, no operation performed by any aborted transaction is ever visible to other transactions (including live ones), and every transaction always

observes a consistent state of the system.

While the semantic definitions presented above define the behavior of the execution of transactions, they do not define the behaviour between code executed within a transaction and code executed outside of transactions. Although we would expect all references to shared data to be contained within transactions, legal programs may contain unprotected references to shared variables (i.e., outside transactions) without creating malignant data races, so both transactional and non-transactional code can refer to the same data. Figure 2.2 depicts an example of a transaction that is executing concurrently with a thread executing code outside a transaction.

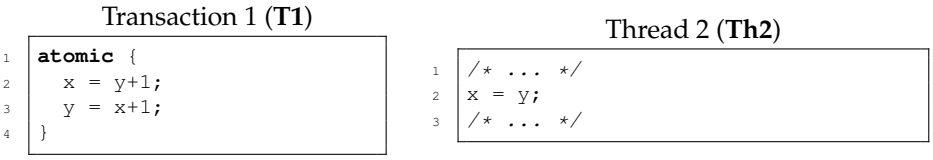


Figure 2.2: Example of a data-race between transactional and non-transactional code.

There are two approaches to define the behavior of the example depicted in Figure 2.2. These two approaches proposed in [BLM05] are called *weak atomicity* and *strong atomicity*.

Under weak atomicity model, the behavior of the example depicted in Figure 2.2 is undefined. The code executed by thread **Th2**, can execute right between line 2 and line 3 of the transaction **T1**. Thus, the isolation guaranteed by the memory transaction is lost with respect to non-transactional code. Blundell et al. [BLM05] define weak atomicity to be a semantics in which transactions are atomic only with respect to other transactions (i.e., their execution may be interleaved with non-transactional code).

This model permits very efficient implementations as it passes the burden of race errors to the programmer, and many STM frameworks implement this model such as DSTM [HLMWNS03] and TL2 [DSS06].

Under strong atomicity, the behavior of the example depicted in Figure 2.2 is either the transaction **T1** executes before the code section of thread **Th2**, or the transaction **T1** executes after the code section of thread **Th2**. Thus, in respect to transaction **T1**, the code in thread **Th2** is executed as a memory transaction. Blundell et al. [BLM05] defines strong atomicity to be a transaction semantics in which transactions execute atomically with respect to both other transactions and non-transactional code.

The implementation of this model implies a great performance penalty in order to enclose non-transactional shared accesses in transactions. Efficient implementations require specialized hardware support not available on existing commodity systems, a sophisticated type system that may not be easily integrated with languages such in Java or C++, or runtime barriers on non-transactional reads or writes that can incur substantial cost in programs that do not use transactions [MBSATHSW08]. Some STM frameworks implement this model, such as JVSTM [CRS06].

2.1.2 Algorithms Implementation

Two main techniques are used to implement STM algorithms: *blocking* and *non-blocking*. Blocking techniques rely mainly on locks to implement the transactional engine, but they may suffer from problems of deadlocking and priority inversion. Non-blocking techniques rely on well study lock-free data structures to implement the transactional engine. This latter technique, although being

free from deadlock problems, may have some performance issues due to the implementation complexity. Independently of the technique used to implement memory transactions, the same semantics must be always preserved.

Michael Scott in [Sco06] proposed a classification of the STM algorithms independent of the implementation techniques. This classification is based on the conflict detection strategy. The conflict detection strategy can be classified as: *lazy invalidation*, *eager W-R*, or *eager invalidation*.

A *lazy invalidation* algorithm detects conflict only at commit time. This means that the transaction will execute until the end, and in the end, it will execute the validation phase where it will check if all shared variables read are still consistent (i.e., if no previous committed transaction wrote to those same shared variable). This algorithm allows efficient implementations of read accesses, because it does not have to validate the consistency of the variable for every read access, but doomed transactions may waste processor time, and it may allow transactions to execute in inconsistent states, hence not satisfying the *opacity* semantics. The OSTM [FH07] framework implements this conflict detection algorithm.

An *eager W-R* algorithm detects the same conflicts as the *lazy invalidation* and also detects a conflict if a transaction performs a read access on a shared variable already written by an incomplete concurrent transaction. Although this algorithm prevents doomed transactions from continue its execution, it may also abort transactions that would not abort under *lazy invalidation*. Also, the implementation of read accesses suffers from a performance penalty. The DSTM [HLMWNS03] framework implements this conflict detection algorithm.

An *eager invalidation* algorithm detects the same conflicts as the *eager W-R* and also detects a conflict if a transaction tries to write a shared variable already read by an incomplete concurrent transaction. This conflict detection strategy is only possible to implement if the STM algorithm implements *visible readers*. In *visible readers* implementations, transactions have access to a list of incomplete transactions that have read a shared variable. On the opposite, in *invisible readers* implementations, the set of incomplete transactions that read a shared variable is not known. The implementation of a *invisible readers* algorithm is quite straightforward, but the implementation of a *visible readers* algorithm requires that each shared variable keep record of all incomplete transactions that accessed such shared variable for reading.

Another classification of the STM algorithms, which is orthogonal to the one just described, is based on the recovery strategy. STM algorithms can be classified as *lazy update* or *eager update*. *Lazy update* algorithms keep write accesses to shared variables in a private log. This technique also called *deferred update*, allows an aborted transaction to just discard this private log of tentative values. If the transaction commits, then the tentative values in the private log must substitute the respective values in the shared variables. *Eager update* algorithms perform write accesses directly in the shared variables, keeping track of their previous values in a private log. This technique, also called *direct update*, allows transactions to commit very efficiently because they just have to discard the private log, however for long transactions, conflicts are more likely to happen.

2.1.3 STM Extensible Frameworks

Software Transactional Memory (STM) algorithms differ in the properties and guarantees they provide. Among others distinctions, one can list distinct strategies used to read (visible or invisible) and update memory (direct or deferred), the consistency (opacity or snapshot isolation) and

```
@atomic
interface INode {
  int getValue();
  void setValue(int v);
  INode getNext();
  void setNext(INode n);
}
```

(a) `INode` interface.

```
class List {
  static Factory<INode> fact =
    dstm2.Thread.makeFactory(INode.class);
  INode root = fact.create();

  void insert(int v) {
    INode newNode = fact.create();
    newNode.setValue(v);
    newNode.setNext(root.getNext());
    root.setNext(newNode);
  }
}
```

(b) `insert` transaction.

```
List list = ...;
int v = ...;
dstm2.Thread.doIt(new Callable<Void>() {
  public Void call() {
    list.insert(v);
    return null;
  }
});
```

(c) Invoking `insert`.

Figure 2.3: DSTM2 programming model.

progress guarantees (blocking or non-blocking), the policies applied to conflict resolution (contention management), and the sensitiveness to interactions with non-transactional code (weak or strong atomicity). The existence of extensible frameworks allows the experimentation with new STM algorithms and their comparison, by providing a unique transactional interface and different implementations for each STM algorithm.

For the Java programming language only two extensible frameworks were proposed: DSTM2 [HLM06] and Deuce [KSF10]. These frameworks allow to experiment new STM algorithms but each one is biased towards some design choices and neither by itself is optimal for implementing all kind of STM algorithms. In Chapter 5 we present an extension of Deuce to support efficient implementation of multi-version algorithms.

In the following sections we will describe in detail the two frameworks: DSTM2 and Deuce.

2.1.3.1 DSTM2

From the work of Herlihy *et al.* comes DSTM2 [HLM06]. This framework is built on the assumption that multiple concurrent threads share data objects. DSTM2 manages synchronization for these objects, which are called *Atomic Objects*. A new kind of thread is supplied that can execute transactions, which access shared Atomic Objects, and provides methods for creating new Atomic Classes and executing transactions.

Perhaps the most notorious difference from the standard programming methodology lies on the implementation of the Atomic Classes. Instead of just implementing a class, this process is separated in two distinct phases:

Declaring the interface First we must define an interface annotated as `@atomic` for the Atomic Class. This interface defines one or more properties by declaring their corresponding *getter* and

setter. These must follow the convention signatures `T getField()` and **`void`** `setField(T t)`, which can be thought as if defining a class field named `field` of type `T`. Additionally, this type `T` can only be *scalar* or atomic. This restriction means that Atomic Objects cannot have array fields, so an `AtomicArray<T>` class is supplied that can be used wherever an array of type `T` would be needed, in order to overcome this. In Figure 2.3a we show an example of the `INode` interface to be used in a transactional linked list.

Implementing the interface The interface is then passed to a transactional factory constructor that returns a transactional factory capable of creating `INode` instances, which is charged with ensuring that the restrictions presented in the previous phase are met. This factory is able to create classes at runtime using a combination of reflection, class loaders, and the *Byte Code Engineering Library* (BCEL)[1], a collection of packages for dynamic creation or transformation of Java class files. This means that atomic objects are no longer instantiated with the **`new`** keyword, but by calling the transactional factory's `create` method.

In the example in Figure 2.3b, the transactional factory is obtained in line 2, and an atomic object is created in line 7.

Lastly, in Figure 2.3c, we inspect how does invoking a method differ from invoking a transaction. DSTM2 supplies a new `Thread` class that is capable of executing methods as transactions. Specifically, its `doIt` method receives a `Callable<T>` object whose `call` method body will be executed as a transaction, wrapped in a start-commit loop.

All things considered, DSTM2's programming model is very intrusive when compared to the sequential model. Atomic classes cannot be implemented directly, instead an `@atomic` interface must be declared (Figure 2.3a). The instantiation of atomic objects is not done through the **`new`** keyword, but by calling the `create` method of the transactional factory (Figure 2.3b). And, finally, starting a transaction is a rather verbose process. We wrap the transaction's body in the `call` method of a `Callable` object that is passed as argument to the `dstm2.Thread.doIt` method (Figure 2.3c). Moreover, the usage of arrays in atomic objects must be replaced by instances of the `AtomicArray<T>` class.

2.1.3.2 Deuce

A more recent proposal of a framework is Deuce [KSF10]. Korland *et al.* aimed for an efficient Java STM framework that could be added to existing applications *without* changing its compilation process or libraries.

In order to achieve such non-intrusive behavior, it relies heavily on Java Bytecode manipulation using ASM[2], an all-purpose Java Bytecode manipulation and analysis framework that can be used to modify existing classes or dynamically generate classes, directly in binary form. This instrumentation is performed dynamically as classes are loaded by the JVM using a Java Agent[3]. Therefore, implementing classes to be modified through transactions is no different from regular Java programming (Figure 2.4a), as Deuce will perform all the necessary instrumentation of the loaded classes.

To tackle performance-related issues, Deuce uses `sun.misc.Unsafe`[4], a collection of methods for performing low-level, unsafe operations. Using `sun.misc.Unsafe` allows Deuce to

[1] `http://commons.apache.org/bcel`
[2] `http://asm.ow2.org`
[3] `http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/package-summary.html`
[4] `http://www.docjar.com/docs/api/sun/misc/Unsafe.html`

```
1  class Node {
2    int value;
3    Node next;
4
5    int getValue() {
6      return value;
7    }
8    void setValue(int v) {
9      value = v;
10   }
11   Node getNext() {
12     return next;
13   }
14   void setNext(Node n) {
15     next = n;
16   }
17 }
```

(a) `Node` class.

```
1  class List {
2    Node root = new Node();
3
4    @Atomic
5    boolean insert(int v) {
6      Node newNode = new Node();
7      newNode.setValue(v);
8      newNode.setNext(root.getNext());
9      root.setNext(newNode);
10   }
11 }
```

(b) `insert` transaction.

```
1  List list = ...;
2  int v = ...;
3  list.insert(v);
```

(c) Invoking `insert`.

Figure 2.4: Deuce's programming model.

directly read and write the memory location of a field f given the $\langle O, f_o \rangle$ pair, where O is an instance object of class $\mathcal{C}$ and f_o the relative position of f in $\mathcal{C}$. This pair uniquely identifies its respective field, thus it is also used by the STM implementation to log field accesses.

As a framework, Deuce allows to plug in custom STM implementations, by implementing a `Context` interface which provides the typical operations used by an STM algorithm, namely: start, read, write, commit, and abort.

We now briefly present the manipulations performed by Deuce. For each field f in any loaded class $\mathcal{C}$, a synthetic constant field is added, holding the value of f_o. In addition to the synthetic field, Deuce will also generate a pair of synthetic accessors, a $\mathcal{G}_f$ *getter* and $\mathcal{S}_f$ *setter*. These accessors encapsulate the read and write operations, by delegating the access to the Deuce runtime (the `Context` implementation). The accessors are invoked with the respective $\langle$**this**$, f_o \rangle$ pair, so the runtime effectively knows which field is being accessed and can read and write its value using `sun.misc.Unsafe`.

Besides instrumenting class fields, Deuce also duplicates *all* methods. For each method m Deuce will create a synthetic method m_t, a copy of method m, to be used when in the context of a transaction. In m_t, read and write accesses to any field f are replaced by calls to the synthetic accessors $\mathcal{G}_f$ and $\mathcal{S}_f$, respectively. Besides the rewriting of field accesses, method calls within m_t are also instrumented. Each call to any method m' is replaced by a call to its transactional synthetic duplicate m'_t. The original method m remains unchanged, to avoid any performance

penalty on non-transactional code as Deuce provides the weak atomicity model.

This duplication has one exception. Each method m^a annotated with `@Atomic` is to be executed as a transaction (Figure 2.4b). Therefore, after the creation of its m_t^a synthetic counterpart, m^a is itself instrumented so that its code becomes the invocation of m_t^a wrapped in the start-commit transactional loop. The practical effect of this is that invoking a transaction is simply calling a method, as seen in Figure 2.4c, provided that the programmer annotates the method with `@Atomic`, of course.

In retrospective, Deuce is optimal regarding programming model intrusion, only requiring the `@Atomic` annotation when compared to the sequential model. Leveraging STM on an existing application using Deuce requires only the annotation of the desired methods, as all these transformations are performed behind the scenes dynamically at class loading.

2.2 Atomicity Violations

Atomicity violations are race conditions where the lack of atomicity of the operations cause incorrect behavior of the program. The atomicity violation depends on the intended atomic properties that the program relies on to ensure the correctness of program executions. A program can contain atomicity violations while being free from data races. This can happen when a program executes two operations in two atomic phases, and the intended behavior is only assured if those two operations execute as a single atomic operation.

Atomicity violations are one of the most causes of errors in concurrent programs [LPSZ08]. In the following sections we present some of the more relevant work in this area.

2.2.1 High-Level Data Races and Stale-Value Errors

Artho et al. defined the notion of view consistency in [AHB03]. View consistency violations are defined as High-Level Data Races and represent sequences of atomic operations in the code that should be atomic in a whole, but are not. A *view* of an atomic operation is the set of variables that are accessed in that atomic operation. The set of views of a thread t is defined as $V(t)$ and a thread t is said to be compatible with a view v if and only if $\{v \cap v' \mid v' \in V(t)\}$ forms a chain, i.e., is totally ordered under $\subseteq$. The program is view consistent if every view from every thread is compatible with every other thread.

The notion of High-Level Data Races (HLDR) does not capture every anomaly regarding the execution of atomic operations, and a HLDR does not imply a real atomicity violation. However this concept is precise enough to capture real world anomalies.

This definition was subsequently extended by Praun and Gross [VPG04] to introduce *methods view consistency*. Method consistency is an extension of view consistency. Based on the intuition that the set of variables that should be accessed atomically in a given method contains all the variables accessed inside a synchronized block. The authors define the concept of method views, which relates to Artho et al's maximal views, and aggregates all the shared variables accessed in a method, and also differentiates between read and write memory accesses. This approach is more precise than Artho et al's because it also detects stale-value errors.

Stale value errors are other type of anomalies that are also related to atomic operations that should be treated as an entire atomic operation. These anomalies are characterized by the re-usage of a value read in an atomic operation in other atomic operations. This may represent an atomicity

violation because the value may be stale, since it could have been updated by a concurrent thread. The freshness of the values may or may not be a problem depending on the application. Analysis to detect stale value errors are formalized in [AHB04; BL04].

The technique that we present in this dissertation, is comparable to the works just described as it detects both high-level data races and stale-value errors with a high precision.

2.2.2 Access Patterns Based Approaches

Vaziri et al. [VTD06] defines eleven access patterns that potentially represent an atomicity violation. The access patterns are sequences of read and write accesses denoted by $R_t(L)$ and $W_t(L)$ and represent, respectively, read and write accesses to memory locations L, performed by thread t. The sequence order represents the execution order of the atomic operations. An example of an access pattern defined in this paper is $R_t(x)\ W_{t'}(x)\ W_t(x)$, that represents a stale value error, since thread t is updating variable x based on an old value. The patterns make explicit use of the atomic set of variables, i.e. sets of variables that must be accessed atomically, and these correlated variables are assumed to be known. These eleven patterns are proved to be complete with respect to serializability.

A related approach by Teixeira et al. [LSTD11] identifies three access patterns that capture a large number of anomalies. These anomalies are referred as RwR, where two related reads are interleaved by a write in those variables; WrW where two related writes are interleaved by a read in those variables; and RwW that represents a stale value error.

2.2.3 Invariant Based Approaches

Another approach to detect atomicity violations is by directly knowing the intended semantics of the program. This was the approach followed in [DV12] by Demeyer and Vanhoof. The authors defined a pure functional concurrent language that is a subset of *Haskell*, and includes the *IO Monad*, hence modeling sequential execution and providing shared variables that can be accesses inside atomic transactions. A specification of the invariants of the program's functions are provided by the programmer in logic. A shared variable is said to be consistent if all invariants upon it hold before and after every atomic transaction. The static analysis acquires the facts about the program and feeds them to a theorem prover to test if every shared variable will be consistent.

This approach is very accurate provided that the programmer can express the notion of program correctness by using invariants on the global state, but is also expensive because of the theorem proving involved.

2.2.4 Dynamic Analysis Based Approaches

Flanagan et al also proposed several methods for detecting atomicity violations [FF04; FFY08; FF10]. In [FF04] is presented a dynamic analysis for serializability violations. The central notion of this work are Lipton's reductions [Lip75]. If a reduction exists from one trace to another then the execution of both traces yield the same state (although different states may be obtained intermediately). Reductions can be found by commuting right- and left-mover operations. Their analysis specifies which operations are movers and uses a result from Lipton's Theory of Reductions to show that there exists a reduction from the atomic operations in the concurrent trace

```
1  void Withdraw(boolean b, int value) {
2    if (x + y > value) {
3      if (b) {
4        x = x - value;
5      }
6      else {
7        y = y - value;
8      }
9    }
10 }
```

Figure 2.5: `Withdraw` program.

obtained dynamically to a serialization trace. If these condition are not met, then an anomaly is reported, this can, however, lead to false positives.

Another work from Flanagan et al provides a sound and complete dynamic analysis for atomicity violations [FFY08]. This work uses a well-known result from database theory that states that a trace is serializable if and only if no cycle exists in the happen-before graph of instructions of atomic operations [BHG87]. The dynamic analysis maintains a happens-before graph and reports anomalies if a cycle is found.

A different approach is presented by Shacham et al. [SBASVY11]. In this work the atomic operations are extracted from the program to be analyzed to create an adversary that will run them concurrently to the original program, if two different runs yield different results then an anomaly is reported. Some heuristics are used to explore the search space of possible interleavings from the adversary.

2.3 Snapshot Isolation

Snapshot Isolation (SI) [BBGMOO95] is a well known relaxed isolation level widely used in databases, where each transaction executes with relation to a private copy of the system state—a snapshot— taken at the beginning of the transaction and stored in a local buffer. All write operations are kept pending in the local buffer until they are committed in the global state. Reading modified items always refer to the pending values in the local buffer.

Tracking memory operations introduces some overhead, and TM systems running under opacity must track both memory read and write accesses, incurring in considerable performance penalties. Validating transactions in SI only requires to check if any two concurrent transaction wrote at a common data item. Hence the runtime system only needs to track the memory write accesses per transaction, ignoring the read accesses, possibly boosting the overall performance of the transactional runtime.

Although appealing for performance reasons, the use of SI may lead to non-serializable executions, resulting in two kinds of consistency anomalies: *write-skew* and *SI read-only anomaly* [FLOOS05]. Consider the following example that suffers from the *write-skew* anomaly. A bank client can withdraw money from two possible accounts represented by two shared variables, `x` and `y`. The program listed in Figure 2.5 can be used in several transactions to perform bank operations customized by its input values. The behavior is based in a parameter `b` and in the sum of the two accounts. Let the initial value of `x` be `20` and the initial value of `y` be `80`. If two transactions T_1 and T_2 execute concurrently, calling `Withdraw(true, 30)` and `Withdraw(false, 90)`

respectively, then one possible execution history of these two transactions under SI is:

$$H = R_1(x, 20)\ R_2(x, 20)\ R_1(y, 80)\ R_2(y, 80)\ R_1(x, 20)\ W_1(x, -10)\ C_1\ R_2(y, 80)\ W_2(y, -10)\ C_2$$

After the execution of these two transactions the final sum of the two accounts will be -20, which is unacceptable. Such execution would never be possible under opacity, as the last transaction to commit would abort because it read a value that was written by the first (committed) transaction.

In the following sections we present the notions of transaction dependency, which is based on execution histories and static dependency between transactional programs. These notions are used to precisely define the possible anomalies that may occur under snapshot isolation.

2.3.1 Transaction Histories

The execution of a transaction can be defined as a sequence of read and write accesses to shared data items that ends either with a commit operation or an abort operation. A transaction implicitly starts upon the execution of the first operation in the sequence. Each transaction results from the execution of a program, hence a program is the static representation of a transaction, and a set of programs constitute an application. We write $R_n(X)$ and $W_n(X)$ to denote read and write accesses to location X in a transaction T_n of application $\mathcal{A}$. We define a history of an application $\mathcal{A}$, written $H(\mathcal{A})$, to an interleaving of the executions of all its transactions. For example consider the application history H_1 for some application $\mathcal{A}$ with transactions T_1 and T_2 and shared variables X and Y:

$$H_1(\mathcal{A}) = R_1(X)\ R_2(X)\ W_2(X)\ C_2\ W_1(X)\ A_1$$

Notice that both transactions T_1 and T_2 read variable X and that T_2 writes variable X and commits. Transaction T_1 then writes variable X but finishes with an abort operation which reverts all the changes made by T_1, and in the end the value of X is the one written by T_2.

An application history H is said to be serializable if its effect on the state of the application is equivalent to the one of a serializable history $\mathcal{S}$ where all transactions are executed sequentially in some given order. For example, consider the following history H_2 for application $\mathcal{A}$ where variable X is initialized with value 50, and where transaction T_1 tries to increment X by 10 and transaction T_2 tries to increment X by 20:

$$H_2(\mathcal{A}) = R_1(X, 50)\ R_2(X, 50)\ W_2(X, 70)\ C_2\ W_1(X, 60)\ C_1$$

Consider that the only two possible histories that sequentially execute transactions T_1 and T_2 to a committed state are $\mathcal{S}_1$ and $\mathcal{S}_2$ below:

$$\mathcal{S}_1 = R_1(X, 50)\ W_1(X, 60)\ C_1\ R_2(X, 60)\ W_2(X, 80)\ C_2$$

$$\mathcal{S}_2 = R_2(X, 50)\ W_2(X, 70)\ C_2\ R_1(X, 70)\ W_1(X, 80)\ C_1$$

If history H_2 is serializable then the final value of variable X should be the same as the one resulting from either $\mathcal{S}_1$ or $\mathcal{S}_2$. Also the two read operations performed by T_1 and T_2 in history H_2 could never yield the same result. Hence, we conclude that history H_2 is not serializable.

However, if we consider history H_3 below, where transaction T_1 is aborted, we would have a serializable application history equivalent to a sequential history where only transaction T_1 is executed.

$$H_3 = R_1(X, 50)\ R_2(X, 50)\ W_2(X, 70)\ C_2\ W_1(X, 60)\ A_1$$

2.3.2 Transaction Dependencies

Fekete et al. [FLOOS05] defines a dependency relation between two transactions based on their execution history. Dependencies between transactions are classified into the following three categories:

- There is a *write-read* dependency, $T_n \xrightarrow{x-wr} T_m$, if a committed transaction T_n wrote a variable x and a committed transaction T_m read the value of variable x written by T_n.
- There is a *write-write* dependency, $T_n \xrightarrow{x-ww} T_m$, if a committed transaction T_n has written variable x and a committed transaction T_m has also written variable x after T_n.
- There is a *read-write* dependency, $T_n \xrightarrow{x-rw} T_m$, if a committed transaction T_n reads variable x which will be later written by a committed transaction T_m, and no other committed transaction T_p writes to the same variable between the read of T_n and the write of T_m.

From this definition we can observe that if there is a write-read dependency $T_1 \xrightarrow{x-wr} T_2$ we know that T_1 has committed before the start of T_2, thus T_1 and T_2 are not concurrent. Otherwise, T_2 would not be able to read the value written by T_1. A write-write dependency $T_1 \xrightarrow{x-ww} T_2$ means that transaction T_1 has committed before the start of T_2 due to the *First-Committer-Wins* rule, and hence they are not concurrent. A read-write dependency $T_1 \xrightarrow{x-rw} T_2$ indicates that T_1 read a value from variable x which will be later written by a committed transaction T_2, and in this case T_1 and T_2 executed concurrently.

The above dependencies can be generalized as:

- There is a $T_n \xrightarrow{wr} T_m$ dependency if $T_n \xrightarrow{i-wr} T_m$ for any data item i.
- There is a $T_n \xrightarrow{ww} T_m$ dependency if $T_n \xrightarrow{i-ww} T_m$ for any data item i.
- There is a $T_n \xrightarrow{rw} T_m$ dependency (also called an anti-dependency) if $T_n \xrightarrow{i-rw} T_m$ for any data item i.
- There is a $T_n \rightarrow T_m$ dependency if any of the following dependencies hold: $T_n \xrightarrow{wr} T_m$, $T_n \xrightarrow{ww} T_m$ or $T_n \xrightarrow{rw} T_m$.

Using these dependency definitions we can construct a dependency graph called dependency serialization graph (DSG) for a history H.

2.3.3 Dependency Serialization Graph

A dependency serialization graph is defined over a history H with vertices representing committed transactions and each distinctly labeled edge from T_m to T_n corresponding to a $T_m \xrightarrow{wr} T_n$, $T_m \xrightarrow{ww} T_n$ or $T_m \xrightarrow{rw} T_n$ dependency.

Consider the follow history:

$$H_3 : W_1(X)\ W_1(Y)\ W_1(Z)\ C_1\ W_3(X)\ R_2(X)\ W_2(Y)\ C_2\ R_3(Y)\ C_3$$

The corresponding dependency serialization graph of history H_3, $DSG(H_3)$, is shown in Figure 2.6.

The edges corresponding to read-write dependencies are drawn as dashed edges to differentiate from other types of edges. Read-write edges have a special role when analyzing the graph in search for serialization anomalies.

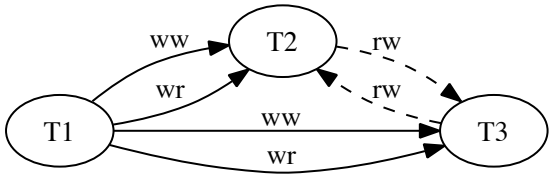


Figure 2.6: DSG of history H_3

2.3.4 Snapshot Isolation Anomalies

Snapshot Isolation anomalies can be defined in terms of a DSG of a history H. Fekete et al. in [FLOOS05] define a theorem that states the following:

Theorem 2.1. *Suppose H is a history produced under Snapshot Isolation that is not serializable. Then there is at least one cycle in the serialization graph $DSG(H)$, and we claim that in every cycle there are three consecutive transactions $T_{i.1}$, $T_{i.2}$, $T_{i.3}$ (where it is possible that $T_{i.1}$ and $T_{i.3}$ are the same transaction) such that $T_{i.1}$ and $T_{i.2}$ are concurrent, with an edge $T_{i.1} \rightarrow T_{i.2}$, and $T_{i.2}$ and $T_{i.3}$ are concurrent with an edge $T_{i.2} \rightarrow T_{i.3}$.*

The type of dependencies of $T_{i.1} \rightarrow T_{i.2}$ and $T_{i.2} \rightarrow T_{i.3}$ must be read-write because $T_{i.1}$ and $T_{i.2}$ are concurrent and $T_{i.2}$ and $T_{i.3}$ are also concurrent. Using this theorem we can easily analyse the dependency graph for anomalies by searching for cycles of the form $T_{i.1} \xrightarrow{rw} T_{i.2} \xrightarrow{rw} T_{i.3} \xrightarrow{wr|ww} T_{i.1}$ where $T_{i.1}$ and $T_{i.3}$ may be the same transaction.

Write Skew The write skew anomaly happens when two transactions running concurrently have read-write conflicts with each other.

Example: there are two accounts `x` and `y` and two methods each one to withdraw from the respective account. The condition to withdraw money from one of the accounts is that the sum of the accounts be higher than the value to be withdrawn.

```
1 int x=20, y=80;
2
3 void withdrawX(int value) {
4   if (x + y > value) {
5     x = x - value;
6   }
7 }
```

```
1
2
3 void withdrawY(int value) {
4   if (x + y > value) {
5     y = y - value;
6   }
7 }
```

If two transactions execute concurrently, one calling the `withdrawX(30)` (T_1) and the other calling the `withdrawY(90)` (T_2), then one possible execution history of the two transactions under SI is:

$$H_{ws} : R_1(x, 20)\ R_2(x, 20)\ R_1(y, 80)\ R_2(y, 80)\ R_1(x, 20)\ W_1(x, -10)\ C_1\ R_2(y, 80)\ W_2(y, -10)\ C_2$$

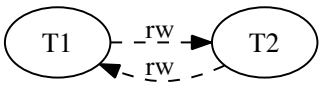


Figure 2.7: $DSG(H_{ws})$: Example of write skew.

After the execution of this two transactions the final sum of the two accounts will be -20 which is negative. Such execution would never be possible under Serializable isolation level as the last transaction to commit would abort because it had read a value that was written by the already committed concurrent transaction. According to [FLOOS05] this example has two dependency relations between transaction T_1 and transaction T_2: a read-write dependency $T_1 \xrightarrow{rw} T_2$ resulting from the operations $R_1(y = 80)$ and $W_2(y = -10)$, and a read-write dependency $T_2 \xrightarrow{rw} T_1$ resulting from the operations $R_2(x = 20)$ and $W_1(x = -10)$. Therefore, there exists a cycle between the dependency relations: $T_1 \xrightarrow{rw} T_2 \xrightarrow{rw} T_1$. Figure 2.7 depicts the dependency graph for history H_{ws}. This example fits in the case of Theorem 2.1 where $T_{i.1}$ and $T_{i.3}$ are the same transaction.

SI Read-Only Anomaly A SI read-only anomaly occurs when a read-only transaction reads a state which cannot occur under Serializable isolation.

Example: There are two accounts `x` and `y` and three methods: the `deposit` method deposits an amount `v` in account `y`, the `withdraw` method withdraws an amount `v` from account `x` if the sum of the accounts `x` and `y` is positive otherwise it withdraws an amount of `v+1`, and the method `readonly` reads the amount available on each account `x` and `y`.

```
1  int x=0, y=0;
2
3  void deposit(int v) {
4    y = y + v;
5  }
```

```
1  void withdraw(int v) {
2    if (x+y > 0) {
3      x = x - v;
4    }
5    else {
6      x = x - v - 1;
7    }
8  }
```

```
1  void readonly() {
2    int tx = x;
3    int ty = y;
4    // print tx and ty
5  }
```

Assume that accounts `x` and `y` start with value zero. If a client decides to first make a deposit of 20 (`deposit(20)`) in account `y` and then it issues the operation to print the amounts of each account (`readonly()`) to make sure that the effects of the deposit operation are persistent, and finally withdraws the amount of 10 from account `x` (`withdraw(10)`). A possible execution of this scenario under SI is given by history H_{ro}:

$$H_{ro} : R_2(x, 0)\ R_2(y, 0)\ R_1(y, 0)\ W_1(y, 20)\ C_1\ R_3(x, 0)\ R_3(y, 20)\ C_3\ W_2(x, -11)\ C_2$$

As we can see from history H_{ro} when the `readonly` operation finishes it read the state after the `deposit` operation, but the `withdraw` operation read the state previous to the `deposit` operation and commits after the `readonly` operation. This would never be possible under Serializable isolation because the `withdraw` operation would have aborted and restarted again in the new state.

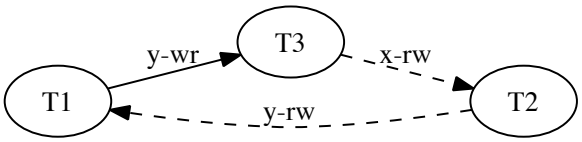


Figure 2.8: $DSG(H_{ro})$: Example of SO read-only anomaly.

In Figure 2.8 is depicted the dependency graph of history H_{ro}. By looking at the $DSG(H_{ro})$ is easy to prove using Theorem 2.1 that this execution is not serializable due to the cycle $T_3 \xrightarrow{x-rw} T_2 \xrightarrow{y-rw} T_1 \xrightarrow{y-wr} T_3$.

2.3.5 Static Dependency Graph

We now define a similar dependency relation on programs and build a static dependency graph (SDG) where the nodes are programs rather than transactions. Remember that transactions are the runtime instances of programs. A program P may define the behavior of many executing transactions. The edges of the SDG correspond to dependencies between programs defined as follows.

For each possible dependency between two transactions $T_n \xrightarrow{x-\rho} T_m$, where x is a state variable and $\rho \in \{wr, ww, rw\}$, there should exist a corresponding static dependency in the SDG. We say that there is a static dependency, written $P_n \rightarrow P_m$, between program P_n and program P_m if, for any transactions T_n and T_m, instantiating P_n and P_m, there is $T_n \xrightarrow{x-\rho} T_m$ on any variable x.

A static dependency is said to be vulnerable [FLOOS05] if there exists a history which has the properties above and in which T_n and T_m are concurrent. The vulnerable static dependency between P_n and P_m is represented as $P_n \Rightarrow P_m$.

It is important to note that in a SDG, a program P may have dependencies to itself because it may generate two different transactions.

In summary, an $SDG(\mathcal{A})$ of an application $\mathcal{A}$ is a graph with programs $P_1, ..., P_k$ of $\mathcal{A}$ as nodes and labeled edges of the form $P_n \xrightarrow{\rho} P_m$ (non-vulnerable) or $P_n \Rightarrow P_m$ (vulnerable) representing static dependencies.

Recall the write-skew anomaly given in Section 2.3.4 where a bank client can withdraw some money from two possible accounts represented by two shared variables, `x` and `y`. The program (P_1) listed in Figure 2.9 can be used in several transactions to perform bank operations customized by its input values. The behavior is based on a parameter `b` and on the sum of the two accounts. Let the initial value of `x` be `20` and the initial value of `y` be `80`.

```
1  void withdraw(bool b, int value) {
2    if (x + y > value) {
3      if (b) {
4        x = x - value;
5      }
6      else {
7        y = y - value;
8      }
9    }
10 }
```

Figure 2.9: Bank account program (P_1).

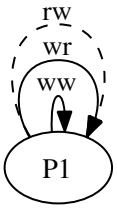


Figure 2.10: Static dependency graph of the `withdraw` function.

Consider the following execution history with two transactions of program P_1:

$$H_4 = R_1(x, 20)\ R_2(x, 20)\ R_1(y, 80)\ R_2(y, 80)\ R_1(x, 20)\ W_1(x, -10)\ C_1\ R_2(y, 80)\ W_2(y, -10)\ C_2$$

Given the above history we can extract the static dependencies of program P_1 and construct a graph with a single node representing P_1 and with edges representing the dependencies. The static dependency graph of program `withdraw` is depicted in Figure 2.10. There is one node in the graph representing program `withdraw` (P_1) and there are three edges: $P_1 \xrightarrow{ww} P_1$, $P_1 \xrightarrow{wr} P_1$ and $P_1 \overset{rw}{\Rightarrow} P_1$ (vulnerable). The vulnerable edge is represented by a dashed arrow in the diagram. Intuitively the three edges represent the following situations: the dependency $P_1 \xrightarrow{ww} P_1$ results from the case where program P is called twice with the same value for parameter `b`; dependency $P_1 \xrightarrow{wr} P_1$ results from a situation where the program is initiated twice with different values for parameter `b` and one of the transactions starts after the commit of the other (non concurrent transactions); dependency $P_1 \overset{rw}{\Rightarrow} P_1$ exists when program P is called twice with different values for parameter `b` and the two transactions are concurrent.

2.3.6 Detection of Anomalies in a SDG

We can not apply the Theorem 2.1 to a SDG because a cycle in a SDG may not correspond to a serialization problem. Fekete et al. [FLOOS05] defines the concept of *dangerous structure* in a static dependency graph. He shows that if some $SDG(\mathcal{A})$ has a dangerous structure then there are executions of application $\mathcal{A}$ which may not be serializable and that if a $SDG(\mathcal{A})$ does not have any dangerous structure then all executions of application $\mathcal{A}$ are serializable.

Definition 2.1 (Dangerous structures [FLOOS05])**.** *We say that a $SDG(\mathcal{A})$ has a dangerous structure if it contains nodes P, Q and R (not necessarily distinct) such that:*

Algorithm 1: Dangerous Structure detection algorithm.

```
Data: nodes[], edges[]
Result: true or false
initialisation;
foreach Node n : nodes do
    foreach Edge in : incoming(n, edges) do
        if vulnerable(in) then
            foreach Edge out : outgoing(n, edges) do
                if vulnerable(out) then
                    if existsPath(target(out), source(in)) then
                        return true;
return false;
```

- *There is a vulnerable edge from R to P.*
- *There is a vulnerable anti-dependency edge from P to Q.*
- *Either $Q = R$ or there is a path in the graph from Q to R; that is, (Q, R) is in the reflexive transitive closure of the edge relationship.*

The detection of dangerous structures in a SDG can be performed mechanically by Algorithm 1.

According to this definition, the SDG in Figure 2.10 has a dangerous structure. Once again the existence of a dangerous structure does not imply that the application will have a SI anomaly, only that it may have one.

2.3.7 Static Analysis of Snapshot Isolation

Fekete et al. [FLOOS05] presents a SQL-based syntactic analysis to detect SI anomalies for the database setting. This analysis computes the set of dependencies between programs, where each program represents a single transaction. Their work assumes that the applications are described in some form of pseudo-code using SQL statements to read or write to the database. Programs with if-then-else structures that have branches with different static dependencies to other program vertices must be split into two or more programs with preconditions.

The analysis may be divided in two phases. The first phase covers the extraction of the lookup and update accesses for each transaction, building their corresponding read and write sets that are composed by sets of table column names. These sets are then used in the second phase to construct a static dependency graph, where the dangerous structures detection algorithm is applied. The analysis presented was applied manually to the TPC-C benchmark, and proved that the benchmark was free of snapshot isolation anomalies.

A sequel of this work, presented in [JFRS07], describes a prototype that automatically analyses database applications. Their syntactic analysis is based on the names of the columns accessed in the SQL statements that occur within the transaction. They also discuss some solutions to reduce the number of false positives produced by their analysis.

2.3.8 Snapshot Isolation in Transactional Memory

Transactional memory systems commonly implement opacity to ensure the correct execution of transactional memory programs. To the best of our knowledge, SI-STM [RFF06] is the only implementation of a STM using snapshot isolation. Their work focuses on the improvement of transactional processing throughput by using a snapshot isolation algorithm on top of a multi-version concurrency control mechanism. They ground on the previous DSTM algorithm and extend it by adding a list of versions to each transactional object. Then each transaction has a *validity range* that is used to retrieve the correct version of each transactional object accessed for reading. The commit operation only checks for write/write conflicts and uses a contention manager to ensure the first-commiter-wins rule. They also propose a SI-safe variant of the algorithm where anomalies are automatic and dynamically avoided by enforcing validation of read/write conflicts. They report performance benefits on using snapshot isolation, although the benchmarks had to be adapted to avoid write-skew anomalies.

In our work, we aim at providing the opacity semantics under snapshot isolation STM systems. This is achieved by performing a static analysis to assert that no SI anomalies will occur when executing a transactional application.

2.4 Static Analysis

In the work presented in this dissertation, we describe two techniques to detect atomicity violations and write-skew anomalies in transactional memory programs. In the particular case of write-skew detection, although targeting similar results, our work deals with significantly different problems than the work of Fekete et al. [FLOOS05]. The most significant one is related to the full power of general purpose languages and the use of dynamically allocated heap data structures.

Both our proposed static analysis techniques are based on abstract interpretation [CC77; Cou01]. Abstract interpretation is a theory of semantics approximation. The objective is to define a new semantics of a programming language that satisfies two conditions: the semantics always terminates and the state of every program statement contains a superset of the values that are possible in the concrete semantic, for every possible input.

2.4.1 Abstract Interpretation

Abstract interpretation techniques use a partial order set to define the state, or abstract state, of a program. The partial order set must be a lattice to guarantee termination of fix point computations. A partial order of a set S is a mathematical structure $L = (S, \sqsubseteq)$ that satisfies the following conditions:

- Reflexivity: $\forall x \in S : x \sqsubseteq x$
- Transitivity: $\forall x, y, z \in S : x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- Anti-symmetry: $\forall x, y \in S : x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$

A set on which there is a defined partial order may also be called a *poset*. Let X be a subset of S. An element $s \in S$ is an upper bound of X if $x \sqsubseteq s$ for all $x \in X$. If the set of the upper bounds of X has a least element z, then z is called the *least upper bound* of X and is denoted as $z = \sqcup X$.

Dually, an element $s \in S$ is a lower bound of X if $s \sqsubseteq x$ for all $x \in X$. If the set of lower bounds of X has a maximum element z, then z is called the *greatest lower bound* of X and is denoted as $z = \sqcap X$.

A lattice is a partial order $(S, \sqsubseteq)$ with a least upper bound $\sqcup X$ where $X \subseteq S$, a greatest lower bound $\sqcap X$, a least element $\bot \in S$, and a greatest element $\top \in S$. The least upper bound between two elements $x, y \in S$ is denoted as $x \sqcup y$ and often called the *join* operator; the greatest lower bound between $x, y \in S$ is denoted as $x \sqcap y$ and often called the *meet* operator.

The abstract interpretation framework is not complete without an abstract semantics function $\mathcal{AF}_{stm} : \mathcal{AS} \to \mathcal{AS}$ that is applied to each program statement stm and where $\mathcal{AS}$ is the abstract state. The correctness of the abstract semantics can be assessed by defining an approximation relation with the concrete state. Let $\mathcal{CS}$ be the concrete state and $\mathcal{AS}$ the abstract state where both sets are lattices. The concrete semantics function $\mathcal{CF}_{stm} : \mathcal{CS} \to \mathcal{CS}$ defines the concrete semantics of a program. Moreover, two additional functions must be defined: function $\gamma : \mathcal{AS} \to \mathcal{CS}$, also called concretization function, which transforms an abstract value onto a concrete one, and function $\alpha : \mathcal{CS} \to \mathcal{AS}$, also called abstraction function, which transforms a concrete value onto an abstract one. The abstract semantics soundness is given by the following theorem:

Theorem 2.2 (Abstract semantics soundness)**.** *The abstract semantics is a sound over-approximation of the concrete semantics:* $\forall s \in \mathcal{AS} : \alpha(\mathcal{CF}_{stm}(\gamma(s))) \sqsubseteq \mathcal{AF}_{stm}(s)$

Sometimes the abstract state lattice may have an infinite height, i.e., there is not a least upper bound or a greatest lower bound of the whole set, although exist for the subsets. In these cases, fix-point computations may take a large amount of time to converge. To solve this problem, a widening operator ∇ [Cor08] is used to accelerate the convergence of the analysis. The widening operator may be defined as $\nabla : \mathcal{AS} \times \mathcal{AS} \to \mathcal{AS}$ where $\forall x, y \in \mathcal{AS} : x \sqsubseteq x \nabla y$ and $y \sqsubseteq x \nabla y$.

Optimizing compilers use simple static analyses to optimize code execution. These analyses use very simple abstract states such as a set of variables, or a map between variables and abstract values. Examples of these analyses include: the live variable analysis and the reaching definitions analysis, among others. But more complex abstract state definitions exist to analyze complex program behaviors. In the context of this work we are concerned about the behavior of dynamically allocated memory, or heap, which is a strong dynamic property of programs. Static analysis techniques that analyze the state of the heap are called shape analysis.

2.4.2 Shape Analysis

The heap is a structure that can have an infinite size and therefore a compact and bounded size representation is needed to analyze a program. From the several representations proposed in the literature, we describe the three most influent: shape graphs [SRW96; SRW98], 3-valued logic representation [SRW99; SRW02], and separation logic [Rey02]. The latter will be extensively described as is the base of our static analysis to detect snapshot isolation anomalies.

Shape Graphs A shape graph is a finite, labeled, directed graph that approximates the concrete stores that can arise during program execution. It abstracts the stack and heap of a program by using the notion of abstract location, which is the representative for one (or more) heap cells in the program heap.

A shape graph is composed by an abstract state $\mathcal{S}$, which is a mapping from variable names to abstract locations, and an abstract heap $\mathcal{H}$, which is a mapping from abstract locations to abstract

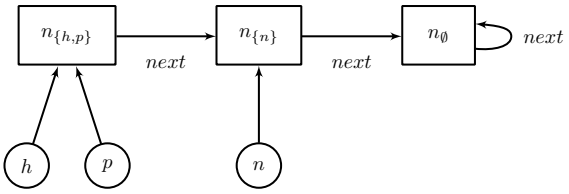


Figure 2.11: Singly linked list represented as a shape graph.

locations by means of selectors. We write n_x to denote an abstract location, where $X \subseteq Vars$ is the set of stack variables pointing to that location. In the general case, abstract locations are associated to one (and only one) heap cell. When $X = \emptyset$ we call $n_\emptyset$ the summary location. In this particular case, $n_\emptyset$ is the representative of more than one heap cell, more precisely, of all the heap cells that are not directly pointed by a stack variable. For instance, $n_{\{x,y\}}$ is the abstract location that represents a heap cell pointed by the stack variables x and y. This also means that variables x and y are aliases of the same heap cell.

Shape graphs also maintain information about sharing. More specifically they keep track of abstract locations that may be the target of more than one pointer from other distinct abstract locations. This information is represented as a map between abstract locations and a boolean value indicating weather the location is shared or not. This is particularly important for summary locations that may represent several concrete heap locations. For instance, if a summary location $n_\emptyset$ is pointed by two abstract locations, and $is\text{-}shared(n_\emptyset) = false$, then we know that the two abstract locations point to distinct locations in the summary location. Sharing information can be used to distinguish between acyclic- and cyclic-lists.

In Figure 2.11 is shown an example of a shape graph representing a singly linked list with some additional variables. Variables h and p are alias of each other for the list head, and variable n is pointing to the second node of the list. In this case the, the sharing information of the summary location $n_\emptyset$ would be $is\text{-}shared(n_\emptyset) = false$, denoting an acyclic linked list.

3-Valued Logic Sagiv et al. developed a parametric framework for specifying shape analysis in which the concrete and abstract states were represented as 2-valued and 3-valued logic formulas respectively. Memory locations are represented as logical constants ranging in $u_1, \ldots, u_n$. Stack variables pointing to some memory location are represented as unary predicates where the variable name is used as the predicate name. Links between memory locations are represented as binary predicates where the field name is used as the predicate name. In 3-valued logic, predicates may evaluate to three different values: 0, 1, or $1/2$ (i.e., false, true, and unknown, respectively).

In the representation of abstract heaps, summary locations are not represented with a special logical constant, but rather using the $1/2$ (*unknown*) value when evaluating a predicate that denotes some variable or location pointing to the summary location. Moreover, an additional unary predicate, called sm, is used to denote if a constant u represents more than one location. This predicate evaluates to 0 whenever the respective constant represents only one location, and $1/2$ when it represents more than one location. For instance, consider the linked list example of Figure 2.11. The 3-valued logic representation of such shape graph would be:

$$h(u_1) = 1 \quad \wedge \quad p(u_1) = 1 \quad \wedge \quad next(u_1, u_2) = 1 \quad \wedge \quad n(u_2) = 1$$
$$\wedge \quad next(u_2, u_3) = 1/2 \quad \wedge \quad next(u_3, u_3) = 1/2 \quad \wedge \quad sm(u_3) = 1/2$$

where h, p and n are stack variables, $next$ is a memory field, and u_1, u_2, u_3 are logical constants.

Separation Logic Separation logic [Rey02], is a first order logic extended with a *separation* conjunction operator ($*$) and a *points-to* predicate ($\mapsto$).

The abstract heap is modeled as a symbolic heap [BCO05] composed by a pure and a spatial part ($\Pi|\Sigma$). The pure part is composed by a conjunction of equalities between stack variables, capturing their aliasing. The spatial part captures the structure of the heap by representing it as a separation logic formulae.

The separation conjunction $P * Q$ denotes that the heap region represented by formula P is disjoint from the heap region represented by formula Q. The points-to predicate $x \mapsto [next : y]$ denotes that variable x is pointing to a location where the $next$ field holds a pointer to the same location as variable y.

The possible infinite heap structure of recursive data structures is represented in separation logic using recursively defined predicates. For instance, a non-empty list segment between variable x and variable y can be defined as:

$$lseg(x, y) \Leftrightarrow x \mapsto [next : y] \vee \exists z'.\ x \mapsto [next : z'] * lseg(z', y)$$

Consider the linked list example of Figure 2.11. The equivalent separation logic representation would be:

$$h = p \mid h \mapsto [next : n] * lseg(n, \mathsf{nil})$$

In the following section, we will present in detail the shape analysis technique based on separation logic.

2.4.3 Shape Analysis based on Separation Logic

The pioneer work of Distefano et al. [DOY06] formalized the first shape analysis algorithm based on separation logic capable of automatically inferring loop invariants and proving shape properties of list data structures for a simple imperative language with dynamically allocated memory.

This shape analysis algorithm is an intra-procedural analysis (i.e., analysis of single program without procedure calls.) over a program annotated with pre- and post-conditions. The algorithm verifies that the pos-condition is implied by the analysis result. The abstract domain is composed by a set of symbolic heaps [BCO05], briefly described in the previous section. In the following sections we will describe in detail the semantic of symbolic heaps and the abstract semantics of this shape analysis algorithm.

2.4.3.1 Symbolic Heaps

Separation logic is an extension of Hoare's logic [Hoa69] which as been used to reason about dynamically allocated memory. The success key of this approach is the separation conjunction ($*$), which allows to reason about only a portion of the heap with the guarantee of non-interference of other portions. The assertion $P * Q$ is satisfied by two disjoint portions of the heap h_1 and h_2 where h_1 satisfies formula P and h_2 satisfies formula Q.

This separateness property is fundamental to reason about programs that manipulate the heap in a local way. This local reasoning concept is materialized by the frame rule:

$$
\begin{array}{lcll}
e & ::= & & \textit{(expressions)} \\
 & & x, y, \ldots \in \mathsf{Vars} & \text{(program variables)} \\
 & | & x', y', \ldots \in \mathsf{Vars}' & \text{(existential variables)} \\
 & | & \mathsf{nil} & \text{(null value)} \\
\rho & ::= & f_1 : e,\ \ldots,\ f_n : e & \textit{(record)} \\
 & & & \\
S & ::= & e \mapsto [\rho] \ \mid\ p(\vec{e}) & \textit{(spatial predicates)} \\
P & ::= & e = e & \textit{(pure predicates)} \\
\Pi & ::= & \mathsf{true} \ \mid\ P \wedge \Pi & \textit{(pure part)} \\
\Sigma & ::= & \mathsf{emp} \ \mid\ S * \Sigma & \textit{(spatial part)} \\
 & & & \\
\mathcal{H} & ::= & \Pi | \Sigma & \textit{(symbolic heap)}
\end{array}
$$

Figure 2.12: Symbolic heaps syntax

$$\frac{\{P\}\ c\ \{Q\}}{\{P * R\}\ c\ \{Q * R\}} (\text{FRAME RULE})$$

The frame rule allows to extend a local specification with other *independent* resources. The local specification of a program statement c, also called the footprint of c, is the portion of the heap that is used by c. For program verification purposes this property allows to build compositional verification procedures [CDOY09].

In program verification only a fragment of separation logic is used. The classical conjunction ($\wedge$) and disjunction ($\vee$), or the separation implication ($-\!\!*$) are dropped because of the complexity rapidly becomes unmanageable. In fact, it has been shown that unrestricted separation logic is undecidable even in the purely propositional setting [BK10].

Symbolic heaps [BCO05] are commonly used as the abstract domain of shape analysis based on separation logic. The store model used to define the semantics of symbolic heaps is composed by a stack (a mapping from variables to values, which include memory locations) and a heap (a mapping from locations to values through field labels). Moreover, we assume a countable set of program variables Vars (ranged over by $x, y, \ldots$), a countable disjoint set of primed variables Vars' (ranged over by $x', y', \ldots$), a countable set of locations $\mathsf{Locations}$, and a finite set of field names Fields.

$$
\begin{aligned}
\mathsf{Values} &= \mathsf{Locations} \cup \{\mathsf{nil}\} \\
\mathsf{Stacks} &= (\mathsf{Vars} \cup \mathsf{Vars}') \to \mathsf{Values} \\
\mathsf{Heaps} &= \mathsf{Locations} \rightharpoonup_{fin} (\mathsf{Fields} \to \mathsf{Values})
\end{aligned}
$$

The fragment of separation logic formulae that we use to describe symbolic heaps is defined by the grammar in Figure 2.12. Satisfaction of a formula $\mathcal{H}$ by a stack $s \in \mathsf{Stacks}$ and a heap $h \in \mathsf{Heaps}$ is denoted $s, h \models \mathcal{H}$ and defined by structural induction on $\mathcal{H}$ in Figure 2.13. There, $[\![p]\!]$ is as usual a component of the least fixed point of a monotone operator constructed from a inductive definition set; a full description can be found in [BBC08]. In this heap model a location maps to a record of values. The formula $e \mapsto [\rho]$ can mention any number of fields in ρ, and the values of the remaining fields are implicitly existentially quantified.

$$
\begin{array}{rll}
s, h \models \mathsf{emp} & \text{iff} & \mathsf{dom}(h) = \emptyset \\
s, h \models x \mapsto [f_1 : e_1, \ , f_n : e_n] & \text{iff} & h = [s(x) \mapsto r] \text{ where } r(f_i) = s(e_i) \text{ for } i \in [1, n] \\
s, h \models p(\vec{e}) & \text{iff} & (s(\vec{e}), h) \in [\![p]\!] \\
s, h \models \Sigma_0 * \Sigma_1 & \text{iff} & \exists h_0, h_1.\ h = h_0 * h_1 \text{ and } s, h_0 \models \Sigma_0 \text{ and } s, h_1 \models \Sigma_1 \\
s, h \models e_1 = e_2 & \text{iff} & s(e_1) = s(e_2) \\
s, h \models \Pi_1 \wedge \Pi_2 & \text{iff} & s, h \models \Pi_1 \text{ and } s, h \models \Pi_2 \\
\\
s, h \models \Pi | \Sigma & \text{iff} & \exists \vec{v'}.\ \big(s(\vec{x'} \mapsto \vec{v'}), h \models \Pi\big) \text{ and } \big(s(\vec{x'} \mapsto \vec{v'}), h \models \Sigma\big) \\
 & & \text{where } \vec{x'} \text{ is the collection of existential variables} \\
 & & \text{in } \Pi | \Sigma
\end{array}
$$

Figure 2.13: Symbolic heaps semantics

e	::=		(*expression*)	b	::=		(*boolean exp*)
		x	(variables)			$e \oplus_b e$	(boolean op)
	|	null	(null value)		|	true | false	(bool values)
A	::=		(*assignments*)	S	::=		(*statements*)
		$x := e$	(local)			$S\ ;\ S$	(sequence)
	|	$x := y.f$	(heap read)		|	A	(assignment)
	|	$x.f := e$	(heap write)		|	if b then S else S	(conditional)
	|	new(x)	(allocation)		|	while b do S	(loop)

Figure 2.14: Simple imperative language.

Symbolic heaps are abstract models of the heap of the form $\mathcal{H} = \Pi|\Sigma$ where Π is called the *pure part* and Σ is called the *spatial part*. Primed variables $(x'_1, \ldots, x'_n)$ are used to implicitly denote existentially quantified variables that occur in $\Pi|\Sigma$. The pure part Π is a conjunction of pure predicates which states facts about the stack variables and existential variables (e.g., $x = \mathsf{nil}$). The spatial part Σ is the $*$ conjunction of spatial predicates, i.e., related to heap facts. In separation logic, the formula $S_1 * S_2$ holds in a heap that can be split into two disjoint parts, one of them described exclusively by S_1 and the other described exclusively by S_2.

2.4.3.2 Abstract Semantics

The abstract semantics is defined over a simple imperative language defined in Figure 2.14. This language captures essential features of imperative languages with dynamically allocated memory such as object creation (new), field dereferencing ($x.f$), and assignment ($x := e$).

For the sake of simplicity, and without losing generality, we define the abstract semantics for a single symbolic heap. We present the abstract semantics rules in Figure 2.15. The simplified abstract semantics is a function $\mathcal{AF} : \mathsf{SHeaps} \longrightarrow \mathcal{P}(\mathsf{SHeaps})$ where SHeaps is the set of all symbolic heaps. In the ASSIGN rule, variable x is renamed to an existential variable x' in symbolic heap $\mathcal{H}$ and a new equality $x = e$ is added the pure part denoting the new assignment of x. In the READ rule, variable x is assigned with the value of $y.f$ therefore we need to rename x as in the ASSIGN rule, and add the equality $x = z$ where z is the value of $y.f$ in the symbolic heap $\mathcal{H}$. In

$$\langle \mathcal{H}, S \rangle \Longrightarrow \langle \mathcal{H}' \rangle$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, x := e \rangle \Longrightarrow \langle x = e[x'/x] \wedge \mathcal{H}[x'/x] \rangle} (\text{ASSIGN})$$

$$\frac{\mathcal{H} \vdash \mathcal{H}' * y \mapsto [f : z] \qquad x' \text{ is fresh}}{\langle \mathcal{H}, x := y.f \rangle \Longrightarrow \langle x = z[x'/x] \wedge \mathcal{H}[x'/x] \rangle} (\text{READ})$$

$$\frac{\mathcal{H} \vdash \mathcal{H}' * x \mapsto [f : y]}{\langle \mathcal{H}, x.f := e \rangle \Longrightarrow \langle \mathcal{H}' * x \mapsto [f : e] \rangle} (\text{WRITE})$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, \mathsf{new}(x) \rangle \Longrightarrow \langle \mathcal{H}[x'/x] * x \mapsto [] \rangle} (\text{ALLOCATION})$$

Figure 2.15: Operational Symbolic Execution Rules

the WRITE rule, the value e is associated with field f of the memory location pointed by variable x. In the ALLOCATION rule, a new points-to predicate for variable x is added to the spatial part of the symbolic heap after renaming the occurrences of x.

To define the final abstract semantics definition, we can lift the domain of $\mathcal{AF}$ to $\mathcal{P}(\mathsf{SHeaps})$ by defining the following function $\mathcal{AF}^\dagger : \mathcal{P}(\mathsf{SHeaps}) \longrightarrow \mathcal{P}(\mathsf{SHeaps})$ where:

$$\mathcal{AF}^\dagger(\mathcal{H}^\dagger) = \bigcup_{\mathcal{H} \in \mathcal{H}^\dagger} \mathcal{AF}(\mathcal{H})$$

The abstract semantics of conditional statements is the union of the resulting symbolic heaps of each branch. The resulting symbolic heap of a loop statement, which corresponds to the loop invariant, is computed using a fix-point computation over the abstract semantics rules defined in Figure 2.15.

The READ and WRITE rules require, in the pre-condition, a symbolic heap with an explicit points-to predicate for variable y (in READ), or variable x (in WRITE), in order to read, or write, the value associated with field f. To satisfy this requirement, the symbolic heap must be *transformed* before applying the rule. This transformation is called *rearrangement*. A rearrangement rule $\mathsf{rearr} : \mathsf{SHeaps} \times \mathsf{Vars} \longrightarrow \mathcal{P}(\mathsf{SHeaps})$ is a function defined as:

$$\mathsf{rearr}(\mathcal{H}, x) = \{\mathcal{H}' * x \mapsto [\ldots] \mid \mathcal{H} \vdash \mathcal{H}' * x \mapsto [\ldots]\}$$

The application of the rearrangement rule may generate more than one symbolic heap due to the unfolding of inductive predicates. For example, the rearrangement of the symbolic heap $x = y \mid lseg(x, \mathsf{nil})$ for variable x would generate the following set of symbolic heaps:

$$\{x = y \mid x \mapsto [next : nil], \quad x = y \mid x \mapsto [next : z'] * lseg(z', nil)\}$$

Where predicate *lseg* is defined as:

$$lseg(x, y) \Leftrightarrow x \mapsto [next : y] \vee \exists z'.\ x \mapsto [next : z'] * lseg(z', y)$$

The unfolding of inductive predicates may lead the analysis to diverge because infinite applications of the rearrangement rule may occur while analyzing a loop statement, generating an infinite sequence of points-to predicates. To solve this problem a new set of rules must be applied to the symbolic heaps in the end of each loop iteration. These set of rules are called *abstraction* rules. Abstraction rules are rewrite rules that transform symbolic heap in a more abstract one, usually by folding a sequence of points-to predicates in an inductive predicate. The rewriting rules have the following structure:

$$\frac{premises}{\mathcal{H} \vdash \mathsf{emp} \rightsquigarrow \mathcal{H}' \vdash \mathsf{emp}} (\text{ABSTRACTION RULE})$$

This rewrite is sound if the symbolic heap $\mathcal{H}$ implies the symbolic heap $\mathcal{H}'$. The application of these rules ensure termination of the analysis, and allow to automatically compute loop invariants. Below we show an example of an abstraction rule for the $lseg$ predicate:

$$\frac{x' \notin \mathit{Vars}(\mathcal{H})}{x \mapsto [next : x'] * lseg(x', \mathsf{nil}) * \mathcal{H} \rightsquigarrow lseg(x, \mathsf{nil}) * \mathcal{H}}$$

The existential variable x' must not occur in remaining symbolic heap $\mathcal{H}$, because otherwise we may be losing essential information for the analysis, such as some other program variable pointing to the middle of the list.

2.4.3.3 Evolution of Separation Logic Based Shape Analysis

Since the pioneer work of Distefano et al. [DOY06], several extensions and significant improvements have been proposed. There exists an extensive literature on the subject. We will point out only a few works that led to the development of scalable shape analysis algorithms capable of analyzing large and complex systems.

In [GBC06] is presented an inter-procedural version of the shape analysis described in the previous section. By relying in the spatial locality of each procedure, this new analysis is able to automatically compute procedure summaries represented by symbolic heaps.

Shape analysis based on separation logic traditionally require user-annotated pre- and post-conditions. In [CDOY07], Calcagno et al. present the first work to automatically infer pre-conditions without user assistance. They call the analysis footprint analysis, as the analysis tries to discover an over-approximation of the memory footprint, specified in separation logic. With the footprint analysis one might analyze several procedures independently, and then use the results as partial summaries to avoid analyzing the whole program, which sometimes might not be even available.

In [BCCDOWY07] is presented an extension of the shape analysis abstract domain to support composite data-structures, e.g., "singly-linked lists of cyclic doubly linked lists with back-pointers to head nodes". This extension relies in a generic higher-order inductive predicates describing spatial relationships. This new predicate definition allows to describe complex data-structures present in system code such as device drivers.

In [YLBCCDO08] is presented a sound join operator for the separation logic domain which increases the analysis scalability without incurring in false negatives. This new analysis is the first working application of shape analysis to verification of whole industrial programs, such as

windows and linux device drivers.

In [DPJ08] is presented the jStar framework. jStar is an automatic verification tool for Java programs, based on separation logic, that enables the automatic verification of entire implementations of several design patterns. The framework is highly customizable allowing the developer to define the properties to be verified.

In [CDOY09] is presented a compositional shape analysis based on separation logic. The analysis follows a bottom-up approach where pre- and post-conditions are automatically inferred using a technique called bi-abduction. Bi-abduction is the technique to infer the anti-frame (missing part of the state) and the frame (portion of state not touched by an operation) of a separation logic assertion. The described analysis is able to analyze the entire code-base of several open-source projects including the linux kernel.

In [CD11] is presented a new automatic program verification tool aimed at proving memory safety of C programs. This is an industrial tool developed by Monoidics Ltd[5].

2.4.4 Shape Analysis to Detect Memory Accesses

One of the main contributions of this thesis concerns the detection of snapshot isolation anomalies at compile time. To perform such detection we need to compute the set of read and write memory accesses made by programs. Achieving this objective requires the use of a shape analysis technique capable of computing an approximation of the read and write accesses.

There are some works in the literature describing analysis algorithms with similar objectives, especially in the area of purity analysis (e.g., checking that a method does not make updates to memory). Salcianu and Rinard [SR05] present a purity analysis method for Java programs capable of asserting if a method makes an update to an *external* abstract location (i.e., an abstract location that already existed upon the start of the method). Methods that do not make updates to external abstract locations are considered pure. This analysis computes a points-to graph for each method that distinguishes between locations that are allocated inside the method and locations that are received by parameter or are loaded from parameters. Along with the points-to graph, the analysis also computes a set of abstract field accesses, which stores the updates made to parameter locations or loaded locations.

Prabhu et al. [PRV10] informally describes a static analysis to infer that speculative executions do not need to rollback. To successfully infer this safety property, the analysis computes an over-approximation of the read and write accesses made by each method. The analyses uses a combined pointer and escape analysis similar to previous described analysis [SR05]. Moreover, the analysis also computes an under-approximation of write accesses.

The work described in [PMPM11] presents a novel shape analysis to optimize programs with set and graph data structures, which infers properties for optimizing speculative parallel graph programs. The shape analysis was implemented using the TLVA system, which implements the 3-valued logic shape analysis presented in [SRW02]. The analysis computes an under-approximation of the set of objects that are always locked at a program point. For each root variable is generated a set of path expressions (i.e., sequence of fields starting from a program variable) denoting the set of locked objects. These path expressions are limited to bounded size data structures, which limits the applicability of this method, for example, to recursive data structures.

[5] `http://www.monoidics.com`

The approach described in [RCG09] defines an analysis to detect memory independences between statements in a program, which can be used for parallelization. They extended separation logic formulae with labels, which are used to keep track of memory regions through an execution. They can prove that two distinct program fragments use disjoint memory regions on all executions, and hence, these program fragments can be safely parallelized.

3

Detection of Atomicity Violations

Concurrent programming is not a trivial task even when using high-level abstractions such as memory transactions. Memory transactions provide the ACI (atomicity, consistency, and isolation) model semantics, which allows the programmer to reason sequentially about the transaction code. Although a single transaction code executes without interference from others, the execution of two consecutive transactions by the same thread may be interleaved by transactions ran by other thread. The programmer must be aware of this fact when writing the code for each transaction, otherwise application invariants may be broken and other semantic errors may arise. These errors are called atomicity violations and are mostly due to the wrong definition of the scope of transaction in the program.

In this chapter we present a technique to detect two kinds of atomicity violations: high-level data-races and stale-value errors. These atomicity violations are detected using static analysis algorithms, which we will describe in detail throughout the chapter.

3.1 Introduction

The absence or misspecification of the scope of atomic blocks[1] in a concurrent program may trigger atomicity violations and lead to runtime misbehaviors.

Low-level data races occur when the program includes unsynchronized accesses to a shared variable, and at least one of those accesses is a write, i.e., one of those accesses changes the value of the variable. Although low-level data races are still a common source of errors and malfunctions in concurrent programs, they have been addressed by others in the past [SBNSA97; CLLOSS02; MHFA13] and are out of the scope of this work. We will consider herein that the concurrent programs under analysis are free from low-level data races.

High-level data races results from the misspecification of the scope of an atomic block, by splitting it in two or more atomic blocks with other (possibly empty) non-atomic block between

[1] Memory transactions can be specified using atomic blocks.

```
1  atomic void getA() {
2    return pair.a;
3  }
4  atomic void getB() {
5    return pair.b;
6  }
7  atomic void setPair(int a, int b){
8    pair.a = a;
9    pair.b = b;
10 }
11 boolean areEqual(){
12   int a = getA();
13   int b = getB();
14   return a == b;
15 }
```

(a) A high-level data race.

```
1  atomic int getX() {
2    return x;
3  }
4  atomic void setX(int p0) {
5    x = p0;
6  }
7  void incX(int val) {
8    int tmp = getX();
9    tmp = tmp + val;
10   setX(tmp);
11 }
```

(b) A stale value error.

Figure 3.1: Example of atomicity violations.

them. This anomaly is often referred as a high-level data race, and is illustrated in Figure 3.1a. A thread uses the method `areEqual()` to check if the fields '`a`' and '`b`' are equal. This method reads both fields in separate atomic blocks, storing their values in local variables, which are then compared. However, due to an interleaving with another thread running the method `setPair()` between lines 12 and 13, the value of the pair may have changed and the first thread observes an inconsistent pair, composed by the old value of '`a`' and the new value of '`b`'.

Figure 3.1b illustrates a stale value error, another source of atomicity violations in concurrent programs. The non-atomic method `incX()` is implemented by resorting to two atomic methods, `getX()` (at line 1) and `setX()` (at line 4). If the current thread is suspended immediately before or after the execution of line 9, and another thread is scheduled to execute `setX()`, the value of '`x`' changes, and when the execution of the initial thread is resumed it overwrites the value in '`x`' at line 10, causing a lost update. This program fails due to a stale-value error, as at line 8 the value of '`x`' escapes the scope of the atomic method `getX()` and is reused indirectly (by way of its private copy '`tmp`') at line 10, when updating the value of '`x`' in `setX()`.

In this work we propose a novel approach for the detection of high-level data races and stale-value errors in concurrent programs. Our proposal only depends on the concept of atomic regions and is neutral concerning the mechanisms used for their identification. The atomic regions are delimited using the `@Atomic` method annotation. Our approach is based on a novel notion of variable dependencies, which we designate as *causal* dependencies. There is a *causal* dependency between two variables if the value of one of them influences the writing of the other. We also extended previous work from Artho et al. [AHB03] by reflecting the read/write nature of accesses to shared variables inside atomic regions and additionally use the dependencies information to detect both high-level data races and stale-value errors. We formally describe the static analysis algorithms to compute the set of *causal* dependencies of a program and define safety conditions for both high-level data races and stale-value errors.

Our approach can yield both false positives and false negatives. However, the experimental results demonstrate that it still achieves high precision when detecting atomicity violations in well know examples from the literature, suggesting its usefulness for software development tools.

In the next section we define a core language and introduce some definitions that support

e	::=		(*expression*)
		x	(variables)
	|	null	(null value)
A	::=		(*assignments*)
		$x := e$	(local)
	|	$x := y.f$	(heap read)
	|	$x := \mathit{func}(\vec{y})$	(method call)
	|	$x.f := e$	(heap write)
	|	$x :=$ new $id \in C$	(allocation)
S	::=		(*statements*)
		$S\ ;\ S$	(sequence)
	|	A	(assignment)
	|	if e then S else S	(conditional)
	|	while e do S	(loop)
	|	return e	(return)
	|	skip	(Skip)
M	::=	$\mathit{func}(\vec{x})\,\{S\}$	(methods decl)
C	::=	class id $\{\mathit{field}^*\ (M \mid \text{atomic } M)^*\}$	(*class decl*)
P	::=	C^+	(*program*)

Figure 3.2: Core language syntax

the remainder of the Chapter, namely Sections 3.3 and 3.4, where we propose algorithms for defining *causal dependencies* between variables and for detecting atomicity violations (data races). In Section 3.5 we briefly describe a tool that applies the proposed algorithms with static analysis techniques for Java Bytecode programs, and discuss the results obtained in Section 3.6. This chapter terminates with the presentation of the relevant related work in Section 3.7, and with some concluding remarks in Section 3.8.

3.2 Core Language

We start by defining a core language that captures essential features of a subset of the Java programming language, namely class declaration (class $id\{...\}$), object creation (new), field dereferencing ($x.f$), assignment ($x := e$), and method invocation ($\mathit{func}(\vec{x})$). The syntax of the language is defined by the grammar in Figure 3.2.

A program in this language is composed by a set of class declarations. Atomic blocks correspond to methods that are declared using the atomic keyword. Variables can hold integers or object references and boolean values are encoded as integers using the value '1' for true and value '0' for false. We also do not support exception handling as normally found in typical object-oriented languages.

We now define some sets that are necessary to the definition of the static analysis algorithms:

- Classes: is the set of the identifiers of all the classes declared in the program.
- Fields: is the set of all the class fields defined in the program.

- Methods: is the the set of all the methods defined in the program.
- Atomics $\subseteq$ Methods: is the subset of the methods that were declared as atomic.

We define a local (stack) variable as a pair of the form (x, m) where x is the variable identifier and $m \in$ Methods is the method where this variable is declared. For the sake of simplicity we write the pair (x, m) as only x whenever is not ambiguous to do so. The set of all local variables of a program is denoted as LocalVars.

We define a global variable as an object field and we represent it as the pair (c, f) where $c \in$ Classes represents the class where field $f \in$ Fields is declared. The set of all global variables is denoted as GlobalVars. These global variables appear in the code when dereferencing an object reference. For instance, in the statement $x.f := 4$, the expression $x.f$ represents a global variable of the form (c, f) where c is the class of the object reference pointed by local variable x.

We define a function typeof : LocalVars $\rightarrow$ Classes, which given a local variable returns the class of the object reference that it holds. So, in the example above $c =$ typeof(x). This information can be easily obtained because we assume that variables have type annotations as in the Java programming language, although we do not explicitly represent these annotations in the language syntax.

Please note that by deciding to represent an access to a field of an object as a pair with the class of the object reference and the field accessed, we are not able to differentiate between different object instances of the same class, and hence we may consider that there is always at most one object instance of each declared class in the program. This allows us to avoid pointer analysis at the cost of losing precision and becoming unsound in some cases but, as the results in Section 3.6 show, this design choice has proven to be very effective.

Finally we define the set Vars $\equiv$ LocalVars + GlobalVars, which corresponds to all variables used in the program, both local and global variables.

3.3 Causal Dependencies

There is a *causal dependency*, which we will designate herein simply as dependency, between two program variables (local or global) if the value read from one variable influences the value written into the other. For instance, the following expression

```
y := x
```

generates a dependency between variable x and y because the value that is written into variable y was read from variable x. As another example, consider the following code:

```
if (x == 0) { y := 4 }
```

In this example, the variable y is written only if the condition $x = 0$ is true, thus it depends on the current value of variable x and therefore there is also a dependency between variables x and y. We represent a dependency between two variables x and y as $x \hookrightarrow y$ where $x \in$ Vars is the variable read and $y \in$ Vars is the variable written.

For each program (in the core language introduced in Section 3.2), we can compute a directed graph of *causal dependencies*. The information provided by this graph plays an important role in finding correlations between variables, which can be used to detect atomicity violations. We can define two kinds of correlations between variables.

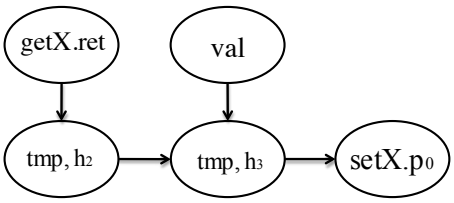


Figure 3.3: Dependency graph example

Definition 3.1 (Direct Correlation)**.** *There is a* direct *correlation between a read variable x and a written variable y if there is a path from x to y, in a dependency graph $\mathcal{D}$.*

We denote as $\mathcal{DC}(x, y)$ a direct correlation between variables x and y.

Definition 3.2 (Common Correlation)**.** *There is a* common *correlation between a read variable x and a read variable y if there is a written variable z, where $z \neq x$ and $z \neq y$, for which exists a direct correlation between x and z ($\mathcal{DC}(x, z)$), and a direct correlation between y and z ($\mathcal{DC}(y, z)$).*

We denote as $\mathcal{CC}(x, y)$ a common correlation between variables x and y.

In the following section we describe how to compute the graph of dependencies from the program code using a static analysis algorithm.

3.3.1 Dependency Analysis

The construction of the dependency graph is done in two steps. In the first step we only detect data dependencies between variables. In the second step we detect control dependencies between variables. In the end we merge all dependencies in a single graph.

3.3.1.1 Data Dependencies

The accurate detection of data dependencies relies on the precise localization of where the variables are defined. SSA (Single Static Assignment) [AWZ88] could be used, because each variable would only have one definition site, but this only works for local variables, and we would still need to track each definition site for global variables. Therefore we did not use SSA as internal representation and we solve the problem by defining a new variable version whenever the variable is updated.

A variable version is defined as a triple of the form (x, h, m) where $x \in \mathsf{Vars}$ is a variable (local or global), h is a unique identifier, and $m \in \mathsf{Atomics} \cup \{\bot\}$ indicates if this variable is used inside an atomic method or not ($\bot$). The set of all variable versions is denoted as $\mathsf{Versions}$.

The unique identifier h is a hash value based on the line of code of the respective definition site. If the version of the variable is not known in the current context, as in the case of method arguments, a special hash value is used. We denote this special hash value as $h_?$.

Figure 3.3 depicts the dependency graph for the method '`incX()`' from Figure 3.1b. For the sake of simplicity, we omitted the method (m) part of the version representation. We denote $getX.ret$ as the return value of method `getX()`, and $setX.p0$ as the parameter of method `setX(int p0)`. Both the parameter and the return value have no need of an associated hash value, which was thus omitted from their representation.

In method `incX(int val)`, the value returned by the method `getX()` is written into a temporary variable tmp, which is then incremented using parameter val, and is used afterwards as a parameter on the invocation of method `setX(int p0)`.

While analyzing this method, we first start by creating the dependency $getX.ret \hookrightarrow (tmp, h_2)$ between the return value of `getX()` method and variable tmp with an hash value h_2. In the next statement variable tmp is redefined with a value resulting from the sum of the previous tmp variable and the val parameter, and hence we create two dependencies $(tmp, h_2) \hookrightarrow (tmp, h_3)$ and $val \hookrightarrow (tmp, h_3)$, where the new version of tmp variable has the hash value h_3. Finally, we invoke method `setX(int p0)` with the value of tmp as parameter and therefore we create the dependency $(tmp, h_3) \hookrightarrow setX.p0$.

The symbolic execution rules are defined as a transition system ($\langle \mathcal{D}, \mathcal{H}, S \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle$) over a state composed by a dependency graph $\mathcal{D}$ and a set of versions, denoted as $\mathcal{H} \subseteq \mathsf{Versions}$, which holds the current versions of each program variable. In a single program point, we may find different versions of the same variable because our analysis over-approximates the runtime state of a program. The rules are depicted in Figure 3.4, where always omit the method (m) parameter from the representation of a variable version.

Function $\mathsf{ver}_{\mathcal{H}}$ is used to retrieve the set of current versions of a variable, and is defined as follows:

Definition 3.3 (Version Retrieval)**.** *Given a set of versions $\mathcal{H}$ and a variable $v \in \mathsf{Vars}$:*

$$\mathsf{ver} : \mathcal{P}(\mathsf{Versions}) \times \mathsf{Vars} \to \mathcal{P}(\mathsf{Versions})$$

$$\mathsf{ver}_{\mathcal{H}}(v) \triangleq \begin{cases} \{(v,h,m) \mid (v,h,m) \in \mathcal{H}\} & \text{if } \exists (v,h,m) \in \mathcal{H} \\ \{(v, h_?, m)\} & \text{otherwise} \end{cases}$$

If a variable version cannot be found in $\mathcal{H}$, a version with the special hash value $h_?$ is returned.

Every time that a variable is written, it is created a new version for such variable and all other existing current versions are replaced by the new one. We define a helper function $\mathsf{subs}_{\mathcal{H}}$ for this purpose as:

Definition 3.4 (Version Substitution)**.** *Given a set of versions $\mathcal{H}$ and a variable version $(v,h,m) \in \mathsf{Versions}$:*

$$\mathsf{subs} : \mathcal{P}(\mathsf{Versions}) \times \mathsf{Versions} \to \mathcal{P}(\mathsf{Versions})$$

$$\mathsf{subs}_{\mathcal{H}}((v,h,m)) \triangleq (\mathcal{H} \setminus \{(v,h',m') \mid (v,h',m') \in \mathcal{H}\}) \cup \{(v,h,m)\}$$

Each hash value is generated using the function nhash, which given a statement S generates a new and unique hash value based in the line number of that statement. This function is deterministic in the sense that for any statement S the same hash value is always returned.

At the beginning of the analysis, the sets $\mathcal{D}$ and $\mathcal{H}$ are empty. We represent the parameters of methods as $meth.p_i$, and the return value of a method as $meth.ret$. When evaluating the RETURN statement, the return value of the method is denoted as retVar.

All assignment operations, namely ASSIGN, HEAP READ, and HEAP WRITE, create dependencies between all versions of the variables used in the right side of the assignment and the new

$$\frac{\langle \mathcal{D}, \mathcal{H}, S_1 \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle \qquad \langle \mathcal{D}', \mathcal{H}', S_2 \rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}'' \rangle}{\langle \mathcal{D}, \mathcal{H}, S_1; S_2 \rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}'' \rangle} (\textsc{Seq})$$

$$\frac{\begin{array}{c} h = \mathsf{nhash}(x := y) \\ \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}((x,h)) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x,h) \mid v \in \mathsf{ver}_{\mathcal{H}}(y)\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := y \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle} (\textsc{Assign})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(y) \qquad h = \mathsf{nhash}(x := y.f) \qquad \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}((x,h)) \\ \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x,h) \mid v \in \mathsf{ver}_{\mathcal{H}}((c,f))\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := y.f \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle} (\textsc{Heap Read})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(x) \qquad h = \mathsf{nhash}(x.f := y) \qquad \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}(((c,f),h)) \\ \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow ((c,f),h) \mid v \in \mathsf{ver}_{\mathcal{H}}(y)\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x.f := y \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle} (\textsc{Heap Write})$$

$$\frac{h = \mathsf{nhash}(x := \mathsf{new}\, C()) \qquad \mathcal{H}' = \mathsf{subs}_{\mathcal{H}}((x,h))}{\langle \mathcal{D}, \mathcal{H}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{D}, \mathcal{H}' \rangle} (\textsc{Allocation})$$

$$\frac{\begin{array}{c} h = \mathsf{nhash}(x := \mathit{func}(\vec{y})) \qquad \mathsf{spec}(\mathit{func}) = \langle \mathcal{D}_f, \mathcal{H}_f \rangle \qquad \mathcal{D}' = \mathcal{D}_f \cup \mathcal{D} \\ \mathcal{D}'' = \mathcal{D}' \cup \{v_i \hookrightarrow \mathit{meth}.p_i \mid y_i \in \vec{y} \wedge v_i \in \mathsf{ver}_{\mathcal{H}}(y_i)\} \cup \{\mathit{meth}.\mathit{ret} \hookrightarrow (x,h)\} \\ \mathcal{H}' = \{(v,h) \mid (v,h) \in \mathcal{H} \wedge ((v,h_?) \in \mathcal{H}_f \vee (v,h) \notin \mathcal{H}_f)\} \\ \mathcal{H}'' = \{(v,h) \mid (v,h) \in \mathcal{H}_f \wedge h \neq h_?\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, x := \mathit{func}(\vec{y}) \rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}' \cup \mathcal{H}'' \rangle} (\textsc{Meth Call})$$

$$\frac{\begin{array}{c} \langle \mathcal{D}, \mathcal{H}, S_1 \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle \qquad \langle \mathcal{D}, \mathcal{H}, S_2 \rangle \Longrightarrow \langle \mathcal{D}'', \mathcal{H}'' \rangle \\ \mathcal{H}''' = \mathcal{H}' \cup \mathcal{H}'' \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H}' \wedge (v,h_2) \notin \mathcal{H}''\} \\ \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H}'' \wedge (v,h_2) \notin \mathcal{H}'\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{if}\, b\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{D}' \cup \mathcal{D}'', \mathcal{H}''' \rangle} (\textsc{Conditional})$$

$$\frac{\begin{array}{c} \langle \mathcal{D}, \mathcal{H}, S \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H}' \rangle \qquad \mathcal{H}'' = \mathcal{H} \cup \mathcal{H}' \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H} \wedge (v,h_2) \notin \mathcal{H}'\} \\ \cup \{(v,h_?) \mid (v,h_1) \in \mathcal{H}' \wedge (v,h_2) \notin \mathcal{H}\} \end{array}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{while}\, b\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{D} \cup \mathcal{D}', \mathcal{H}'' \rangle} (\textsc{Loop})$$

$$\frac{\mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow \mathsf{retVar} \mid v \in \mathsf{ver}_{\mathcal{H}}(x)\}}{\langle \mathcal{D}, \mathcal{H}, \mathsf{return}\, x \rangle \Longrightarrow \langle \mathcal{D}', \mathcal{H} \rangle} (\textsc{Return})$$

$$\frac{}{\langle \mathcal{D}, \mathcal{H}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{D}, \mathcal{H} \rangle} (\textsc{Skip})$$

Figure 3.4: Symbolic execution rules of data dependencies analysis

version of the assigned variable. The newly generated version is then used to replace all existing versions of that same variable.

In the rule METH CALL, the function spec() returns the result, denoted as $\langle \mathcal{D}_p, \mathcal{H}_p \rangle$, of the analysis of method $func$. The dependencies in $\mathcal{D}_p$ are merged with the current dependencies and we create a dependency between each value that is passed as an argument to $func$ and the respective declared parameter $meth.p_i$. We also need to update the variables' versions that are generated inside the method. If a variable was redefined ($h \neq h_?$) inside $func$ then we replace the existing versions with the new version, otherwise we keep the current versions. Finally, we add one more dependency between the return value of method $func$ and the assigned value.

In the rule CONDITIONAL, the dependencies are generated in both branches and are merged with the initial $\mathcal{D}$. We also generate the versions for each branch, and if a variable x has a version $h \neq h_?$ in one branch but there is no version for the same variable in the other branch, then we generate a special version $h_?$ for variable x and we join it to all the other versions. The intuition behind this operation is that if a variable is written only in one of the branches then we also need to add the case that the variable might not have been written. The rule LOOP is similar to the CONDITIONAL rule. The remaining rules should be self-explanatory.

After analyzing all methods of the program we get a dependency graph for the whole program, based on data-flow information. Next, we have to add the remaining dependencies based on the control flow information.

3.3.1.2 Control Dependencies

If an assignment or return statement is guarded by some condition then that assignment or return statement depends on the variables used in the condition. This situation may occur with every conditional statement such as an if then else, or a while loop.

The analysis of control dependencies traverses the control flow graph and keeps the set of variables that the assignments may depend on. When an assignment or return statement is found we create a dependency between the current variables, that it may depend on, and the respective assigned variable.

```
boolean b1,b2,b3,b4;
if(b1){
  // depends on b1
  if(b2){
    // b1 and b2
  }
  else if(b3){
    // b1, b2 and b3
  }
  else{
    // b1, b2 and b3
  }
  // b1
}
```

Figure 3.5: Example of the variables that guard each block

In Figure 3.5 is shown an example with nested conditional statements and the information of the variables that *guard* each inner branch block of the conditions. The state of our symbolic

execution will maintain the same information.

The symbolic execution rules are shown in Figure 3.6 as a transition system ($\langle \mathcal{IS}, \mathcal{D}, S\rangle \implies \langle \mathcal{IS}', \mathcal{D}'\rangle$). The state is composed by a set of conditional variables $\mathcal{IS} \subseteq$ Versions, which correspond to the variable versions that the current statement depends on, and a dependency graph $\mathcal{D}$. In the beginning of the analysis the dependency graph is empty, and the set of conditional variables has the union of all conditional variables that are present at all calling contexts of the method that is going to be analyzed. For instance, given the program methods m_1, m_2 and m_3 where method m_1 calls method m_2 with the current conditional variables set $\mathcal{IS} = \{c_1, c_2\}$, and m_3 calls method m_2 with the current conditional variables set $\mathcal{IS} = \{c_3, c_4\}$, then the initial set of conditional variables when analyzing method m_2 is $\mathcal{IS} = \{c_1, c_2, c_3, c_4\}$.

In the end of this analysis the resulting graph of dependencies is merged with the one that resulted from the data dependencies analysis, described in the previous section, thus forming the complete graph of *causal* dependencies.

For every kind of assignment we create a dependency between the current conditional variables and the assigned variable. This situation may occur in the rules ASSIGN, HEAP READ, HEAP WRITE, ALLOCATION and METH CALL. In the case of a return statement, as in rule RETURN, we create a dependency with the special variable retVar.

In the rules CONDITIONAL and LOOP, we analyze each branch with a new set of conditional variables, which include the current conditional variables plus the variable of the condition. Each variable is actually a variable version with an unique hash value. When we exit the scope of the condition we remove the condition variable and proceed with the analysis. The remaining rules are self-explanatory.

The result of these two analysis generate the graph of *causal* dependencies that is used to detect the existence of atomicity violations in a concurrent program, as we will show in the following sections.

3.4 Atomicity Violations

The purpose of our work is to detect two kinds of atomicity errors, the high-level data race and the stale-value error, that may occur during the execution of concurrent programs that use atomic blocks to guarantee mutual exclusion in the access to shared data.

The definition of both errors assume that the concurrent program has no low-level data races, meaning that all accesses to shared variables are done inside atomic blocks.

3.4.1 High Level data races

A *view*, as described by Artho et al. in [AHB03] is a dynamic property that expresses what variables are accessed inside a given atomic code block. In this work we export this definition as a static property, and additionally extend it, by also keeping the kind of access (read or write) that was made for each variable in the *view*. As in all static analysis, this static property must be an approximation of the dynamic property. In our setting, a *view* is an over-approximation of the variables accessed inside a given atomic method. Please note that a *view* only stores global variables. Local variables are not shared between threads and thus do not require synchronized accesses.

We denote as Accesses the set of memory accesses made inside an atomic block. An access

$$\frac{\langle \mathcal{IS}, \mathcal{D}, S_1 \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}' \rangle \qquad \langle \mathcal{IS}', \mathcal{D}', S_2 \rangle \Longrightarrow \langle \mathcal{IS}'', \mathcal{D}'' \rangle}{\langle \mathcal{IS}, \mathcal{D}, S_1; S_2 \rangle \Longrightarrow \langle \mathcal{IS}'', \mathcal{D}'' \rangle} (\textsc{Seq})$$

$$\frac{h = \mathsf{nhash}(x := y) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := y \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Assign})$$

$$\frac{h = \mathsf{nhash}(x := y.f) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := y.f \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Heap Read})$$

$$\frac{\begin{array}{c} c = \mathsf{typeof}(x) \\ h = \mathsf{nhash}(x.f := y) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow ((c, f), h) \mid v \in \mathcal{IS}\} \end{array}}{\langle \mathcal{IS}, \mathcal{D}, x.f := y \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Heap Write})$$

$$\frac{h = \mathsf{nhash}(x := \mathsf{new}\, C()) \qquad \mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Allocation})$$

$$\frac{\begin{array}{c} h = \mathsf{nhash}(x := \mathit{func}(\vec{y})) \qquad \mathsf{spec}(\mathit{func}) = \langle \mathcal{IS}_f, \mathcal{D}_f \rangle \\ \mathcal{D}' = \mathcal{D} \cup \mathcal{D}_f \cup \{v \hookrightarrow (x, h) \mid v \in \mathcal{IS}\} \end{array}}{\langle \mathcal{IS}, \mathcal{D}, x := \mathit{func}(\vec{y}) \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Meth Call})$$

$$\frac{\begin{array}{c} \mathcal{IS}' = \mathcal{IS} \cup \{b\} \\ \langle \mathcal{IS}', \mathcal{D}, S_1 \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}' \rangle \qquad \langle \mathcal{IS}', \mathcal{D}, S_2 \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}'' \rangle \end{array}}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{if}\, b\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \cup \mathcal{D}'' \rangle} (\textsc{Conditional})$$

$$\frac{\mathcal{IS}' = \mathcal{IS} \cup \{b\} \qquad \langle \mathcal{IS}', \mathcal{D}, S \rangle \Longrightarrow \langle \mathcal{IS}', \mathcal{D}' \rangle}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{while}\, b\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D} \cup \mathcal{D}' \rangle} (\textsc{Loop})$$

$$\frac{\mathcal{D}' = \mathcal{D} \cup \{v \hookrightarrow \mathsf{retVar} \mid v \in \mathcal{IS}\}}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{return}\, x \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D}' \rangle} (\textsc{Return})$$

$$\frac{}{\langle \mathcal{IS}, \mathcal{D}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{IS}, \mathcal{D} \rangle} (\textsc{Skip})$$

Figure 3.6: Symbolic execution rules of control dependencies analysis

$a \in$ Accesses is a pair of the form (α, v) where $\alpha \in \{r, w\}$ represents the kind of access (r-read or w-write) and $v \in$ GlobalVars is a global variable[2]. A *view* is a subset of Accesses and the set of all views in a program is denoted as Views. A *view* is always associated with one atomic method, and we define the bijective function Γ that given a *view* returns the associated atomic method as:

$$\Gamma : \text{Views} \rightarrow \text{Atomics}$$

The inverse function, denoted as Γ^{-1}, returns the *view* associated with a given atomic method. The set of *generated views* of a process p, denoted as $V(p)$, corresponds to the atomic blocks executed by one process, and is defined as:

$$v \in V(p) \Leftrightarrow m = \Gamma(v) \wedge \text{executes}(p, m)$$

The predicate executes asserts if a method m may be executed by process p, and is defined by an auxiliary static analysis that computes the set of processes and the atomic methods that are called in each process using the program call graph.

We can refine the previous definition of $V(p)$ with a parameter α, where $\alpha \in \{r, w\}$, to get only the views of a process with read (V_r) or write accesses (V_w).

Definition 3.5 (Process Views)**.**

$$V_\alpha(p) = \{v_2 \mid v_1 \in V(p) \wedge v_2 = \{(\alpha, x) \mid (\alpha, x) \in v_1\}\} \quad \textit{where } \alpha \in \{r, w\}$$

In Figure 3.7 we show the symbolic execution rules for creating a *view* from the analysis of an atomic method. The result of the analysis is a single *view* that corresponds to the analyzed atomic method. This *view* represents an over-approximation of the memory accesses that are made during the execution of the atomic method. The analysis of all atomic methods in the program results in a set of *views*. The analysis is defined as a reduction relation on an abstract state composed by a single *view*, denoted as $\mathcal{V}$, which is empty at the beginning of the analysis.

Every time a global variable is read or written, the corresponding read or write access is created and added to the *view*. This operation is demonstrated in rules HEAP READ and HEAP WRITE. The rules ASSIGN, ALLOCATION, and RETURN, only access local variables and therefore nothing is done as in the SKIP rule.

In rule METH CALL, the function spec($func$) returns the *view* $\mathcal{V}_f$ that resulted from the analysis of function *func*, and we merge the resulting *view* with the current *view* $\mathcal{V}_f \cup \mathcal{V}$.

The rules CONDITIONAL and LOOP merge the resulting *views* of all branches. This decision allows us to avoid a state explosion and have a scalable analysis with the number of lines of code.

Given the set of views of a process, the *maximal views* of a process, denoted as M_α, are all the views of the process that are not a subset of any other view in that same process. A maximal view is defined as follows:

Definition 3.6 (Maximal Views)**.** *Given a process p, a* maximal view v_m *is defined as:*

$$v_m \in M_\alpha(p) \Leftrightarrow v_m \in V_\alpha(p) \wedge (\forall v \in V_\alpha(p) : v_m \subseteq v \Rightarrow v = v_m) \quad \textit{where } \alpha \in \{r, w\}$$

Each *maximal view* represent the set of variables that should be accessed atomically, i.e., should

[2]Please remember that global variables are represented as a pair with a class identifier and the field accessed.

$$\boxed{\langle \mathcal{V}, S \rangle \Longrightarrow \langle \mathcal{V}' \rangle}$$

$$\frac{\langle \mathcal{V}, S_1 \rangle \Longrightarrow \langle \mathcal{V}' \rangle \qquad \langle \mathcal{V}', S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle}{\langle \mathcal{V}, S_1; S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle} (\textsc{Seq})$$

$$\frac{}{\langle \mathcal{V}, x := e \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Assign})$$

$$\frac{c = \mathsf{typeof}(y) \qquad \mathcal{V}' = \mathcal{V} \cup \{(r, (c, f))\}}{\langle \mathcal{V}, x := y.f \rangle \Longrightarrow \langle \mathcal{V}' \rangle} (\textsc{Heap Read})$$

$$\frac{c = \mathsf{typeof}(x) \qquad \mathcal{V}' = \mathcal{V} \cup \{(w, (c, f))\}}{\langle \mathcal{V}, x.f := e \rangle \Longrightarrow \langle \mathcal{V}' \rangle} (\textsc{Heap Write})$$

$$\frac{}{\langle \mathcal{V}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Allocation})$$

$$\frac{\mathsf{spec}(\mathit{func}) = \mathcal{V}_f \qquad \mathcal{V}' = \mathcal{V}_f \cup \mathcal{V}}{\langle \mathcal{V}, x := \mathit{func}(\vec{y}) \rangle \Longrightarrow \langle \mathcal{V}' \rangle} (\textsc{Meth Call})$$

$$\frac{\langle \mathcal{V}, S_1 \rangle \Longrightarrow \langle \mathcal{V}' \rangle \qquad \langle \mathcal{V}, S_2 \rangle \Longrightarrow \langle \mathcal{V}'' \rangle}{\langle \mathcal{V}, \mathsf{if}\, e\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{V}' \cup \mathcal{V}'' \rangle} (\textsc{Conditional})$$

$$\frac{\langle \mathcal{V}, S \rangle \Longrightarrow \langle \mathcal{V}' \rangle}{\langle \mathcal{V}, \mathsf{while}\, e\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{V} \cup \mathcal{V}' \rangle} (\textsc{Loop})$$

$$\frac{}{\langle \mathcal{V}, \mathsf{return}\, e \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Return})$$

$$\frac{}{\langle \mathcal{V}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{V} \rangle} (\textsc{Skip})$$

Figure 3.7: Symbolic execution rules for creating a *view*

```
int x, y; // global variables
```

Maximal View v_m

```
atomic {
  x = 4;
  y = 5;
}
```

Process p

```
int tx, ty;
atomic {
  tx = x;
}
...
atomic {
  ty = y;
}

print(tx+ty);
```

Figure 3.8: Example of compatibility property between a process p and a maximal view v_m. In this case, process p is incompatible with maximal view v_m.

always be accessed in the same atomic block.

Given a set of views of a process p and a *maximal view* v_m of another process, we define the read or write *overlapping views* of process p with *view* v_m as all the non empty intersection views between v_m and the views of process p.

Definition 3.7 (Overlapping Views)**.** *Given a process p and* maximal view v_m*:*

$$\mathsf{overlap}_\alpha(p, v_m) \triangleq \{v_m \cap v \mid v \in V_\alpha(p) \wedge v_m \cap v \neq \emptyset\} \quad \textit{where } \alpha \in \{r, w\}$$

The notion of compatibility between a process p and a *view* v_m, defined in [AHB03], states that a process p and a *view* v_m are *compatible* if all their *overlapping views* form a chain, i.e., a total ordered set. We explain the intuition behind this definition using the example shown in Figure 3.8. In this example process p and maximal view v_m are not compatible because process p reads the variables x and y in two different atomic blocks, although the same variables are accessed in the same atomic block associated with the maximal view v_m. Hence, the atomic block of v_m may execute between the two atomic blocks of process p and therefore, process p, might see an inconsistent state, or at least, a state that was not intended to be seen by the update done by v_m.

We extended this definition with the information given by the *causal* dependencies graph, and we additionally require that, even if the read overlapping views do not form a chain, there may not exist a *common correlation* (Definition 3.2) between the variables in the read overlapping views.

Definition 3.8 (Process Compatibility)**.** *Given a process p and* maximal view v_m*:*

$$\mathsf{comp}_w(p, v_m) \Leftrightarrow \forall v_1, v_2 \in \mathsf{overlap}_w(p, v_m) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1$$
$$\mathsf{comp}_r(p, v_m) \Leftrightarrow \forall v_1, v_2 \in \mathsf{overlap}_r(p, v_m) : v_1 \subseteq v_2 \vee v_2 \subseteq v_1 \vee \neg \mathcal{CC}(v_1, v_2)$$

The intuition behind this additional condition is that, even if two shared variables that belong to a *maximal view* were read in different atomic blocks, we will only consider that there is an incompatibility if both variables are used in a common write operation as in the example shown in Figure 3.8.

```
int x, y; // global variables

int getXorY(boolean cond) {
  int res;
  atomic {
    if (cond)
      res = x;
    else
      res = y;
  }
  return res;
}
```

Maximal View v_m

```
atomic {
  x = 4;
  y = 5;
}
```

Process p

```
int tx, ty;

tx = getXorY(true);
ty = getXorY(false);

print(tx+ty);
```

Figure 3.9: Example of an atomic block that generates a false-negative.

We can now define the *view consistency* safety property in terms of the compatibility between all pairs of processes of a program. A process may only have *views* that are compatible with all *maximal views* of another process. A program is free from high-level data races if the following condition holds:

Definition 3.9 (View Consistency)**.**

$$\forall p_1, p_2 \in \mathcal{P}_\mathcal{S}, m_r \in M_r(p_1), m_w \in M_w(p_1) : \mathsf{comp}_w(p_2, m_r) \wedge \mathsf{comp}_r(p_2, m_w)$$

where $\mathcal{P}_\mathcal{S}$ is the set of processes.

If the view consistency property is not met, then we consider that there is a high-level data race in the analyzed program.

In order to achieve scalability, our method is unsound, i.e., reports false negatives, mainly due to the way we treat conditional statements. In conditional statements, we join the variables accessed in both branches, although in a concrete execution, only one of the branches is executed. This join operation directly influences the result of the compatibility test between a maximal view and another process. For instance, consider the example shown in Figure 3.9. In this example, the atomic block in method `getXorY` generates a view with two accesses on variables x and y, although at runtime only one of the variables is actually accessed each time the `getXorY` is invoked. Therefore, in this case the two views from process p form a chain and our analysis detects no high-level data race.

3.4.2 Stale-Value Error

Stale-value errors are a class of atomicity violations that are not detected by the *view consistency* property. Our approach to detect this kind of errors uses the graph of *causal* dependencies to

detect values that escape the scope of an atomic block (e.g., by assigning a shared variable to a local variable) and are later used inside another atomic block (e.g., by assigning the previous local variable to a shared variable).

First we define the set $\mathsf{IVersions} \subseteq \mathsf{Versions}$, which stores all global variable versions that were accessed inside an atomic block. Each variable version has a parameter m that indicates in which atomic method it was defined, or has the value $\perp$ if it was not used inside an atomic method.

Definition 3.10 (Atomic Variable Version)**.** *A global variable version* (x, h, m) *is an atomic variable if:*

$$(x, h, m) \in \mathsf{IVersions} \Leftrightarrow (x, h, m) \in \mathsf{Versions} \wedge x \in \mathsf{GlobalVars} \wedge m \neq \perp$$

Now we define a new graph, denoted as $\mathcal{D}_\mathcal{V}$, which represents the dependencies between views. A labeled edge of this graph $\mathcal{D}_\mathcal{V}$ is represented as (m_1, x, m_2) where $m_1, m_2 \in \mathsf{Atomics}$ and $x \in \mathsf{GlobalVars}$, and can be interpreted as atomic method m_2 depends on atomic method m_1 through global variable x. Intuitively, this means that the value of variable x exited the scope of method m_1 and entered the scope of method m_2, and while it was out of the atomic scope it might have become outdated.

Each edge (m_1, x_1, m_2) of a *view* dependency graph $\mathcal{D}_\mathcal{V}$, is created when, given two version variables $a_1 = (x_1, h_1, m_1) \in \mathsf{IVersions}$ and $a_2 = (x_2, h_2, m_2) \in \mathsf{IVersions}$, and a *causal* dependency graph $\mathcal{D}$, the following conditions hold:

$$\big(\mathcal{DC}_\mathcal{D}(a_1, a_2) \wedge m_1 \neq m_2\big) \vee \big(m_1 = m_2 \wedge \mathcal{DC}_\mathcal{D}(a_1, m_1.ret) \\ \wedge \mathcal{DC}_\mathcal{D}(m_1.ret, m_1.p_i) \wedge \mathcal{DC}_\mathcal{D}(m_1.p_i, a_2)\big)$$

The predicate $\mathcal{DC}$ asserts if two variables are *directly correlated* according to Definition 3.1. These conditions state that there is a dependency between m_1 and m_2 through variable x_1, if the variable version a_1 is directly correlated with a_2 when m_1 and m_2 are two different atomic methods, or if the two methods m_1 and m_2 are the same, then there must exist a data-flow relation such that the value of a_1 exits method m_1, through its return variable $m_1.ret$, and enters again the same method through one of its parameters $m_1.p_i$ and is assigned to variable a_2.

A process p writes in a variable $x \in \mathsf{Vars}$ if there is a write access on variable x in one of the views of process p:

$$\mathsf{writes}(x, p) \Leftrightarrow \exists v \in V_w(p) : (w, x) \in v$$

The safety property for *stale-value* errors can be defined as the case where no process writes to a global variable that leaves, and then enters, the scope of an atomic method of another process.

Definition 3.11 (Stale-Value Safety)**.**

$$\forall p \in \mathcal{P}_\mathcal{S}, (m_1, x, m_2) \in \mathcal{D}_\mathcal{V} : \neg\, \mathsf{writes}(x, p) \textit{ where } \mathcal{P}_\mathcal{S} \textit{ is the set of processes,} \\ \textit{and } \mathcal{D}_\mathcal{V} \textit{ is the graph of view dependencies}$$

If there is a *view* dependency for variable x and there is a process p that writes on that variable then a *stale-value* error is detected.

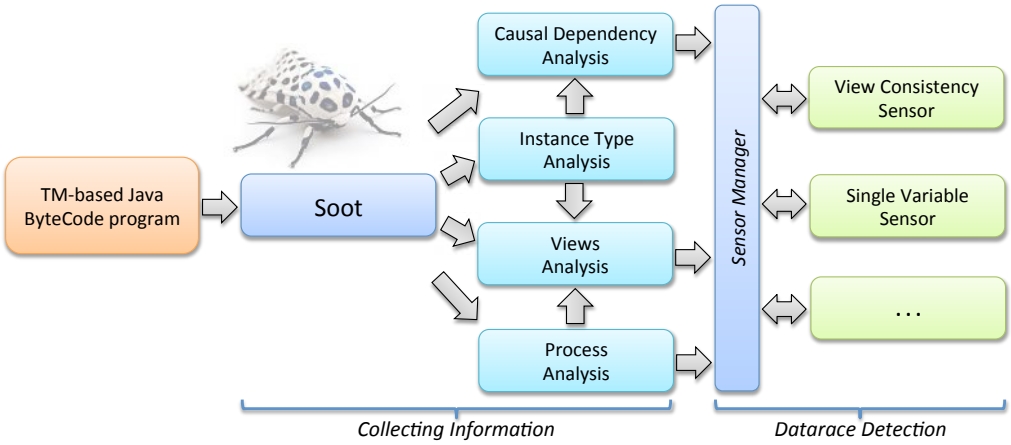


Figure 3.10: MoTH architecture.

3.5 The MoTH Prototype

To evaluate the accuracy of our algorithms and techniques, the theoretical framework described in the previous sections were adapted and implemented in the MoTH tool, in the context of the MSc of Pessanha [Pes11]. This tool targets the Java bytecode language, where the atomic blocks are represented as methods using the `@Atomic` method annotation, and uses the data-flow analysis infrastructure of the Soot framework [RHSLGC99].

To apply our atomicity violation detectors to real programs, some practical problems had first to be solved, namely: identification of the possible set of processes, or threads, generated by the application, cope with virtual method invocations (dynamic dispatch problem), and cope with native method invocations.

The prototype, which we baptized as MoTH, implements the algorithms previously described to detect high-level data races and stale-value errors. MoTH has a very modular architecture that allows to easily extend it with more analysis to detect other properties. We show the architecture schematic in Figure 3.10. The architecture is composed by two different sets of modules. The *collectors* set, contains the static analysis algorithms that collect abstract information about the program being analyzed. This abstract information serves as input to the second set of modules. The *sensors* set, contains all the algorithms that verify abstract properties about the program.

In the following sections we will first describe how to identify the possible set of processes that may be generated by the application, and then describe how we deal with the invocation of interface and native methods.

3.5.1 Process Analysis

Atomicity violations are generated through anomalous interactions between transactions running in different threads. Thus, to detect such interactions it is necessary to ascertain which thread execution flows are generated by the program and which transactions are executed in each thread.

The process analysis aims at identifying the set of different static control-flows that are possible to be executed within an application thread and which atomic methods are executed in each process. We represent the static control-flow of a process as a call-graph, i.e., a graph of method invocations where each node represents a method and the edges correspond to invocations. We

```
static void main(String[] args) {
  MyThread1 t1 = new MyThread1();
  MyThread2 t2 = new MyThread2();
  startTimer();
  t1.start();
  t2.start();
  stopTimer();
}
```

```
class MyThread1 {
  public void run(){
    m1();
    m2();
    }
}
class MyThread2 {
  public void run(){
    m3();
    m4();
   }
}
```

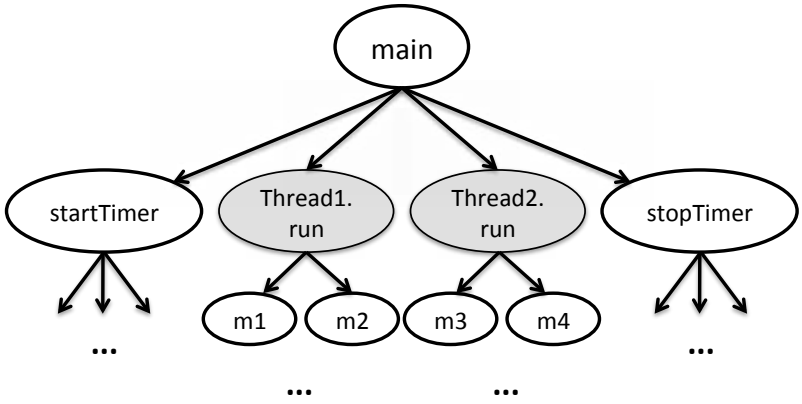


Figure 3.11: Call-graph of the above code examples.

define a process as being a sequence of atomic methods that are possible to be executed in the corresponding process's call-graph.

A Java thread may be created by implementing a class that inherits from the `Runnable` interface, or that extends the `Thread` class. In MoTH we define two kinds of processes, the process generated by the execution of the `main` method (the program's entry point), and the processes generated by the creation of new Java threads. We denote as P_{main} the former and as P_C the latter kind of process. In Figure 3.11 we show an example of a Java program that creates two threads, and present the corresponding call-graph. The grey-filled nodes represent the `run` methods that create new processes.

Since the same thread class, may be used to create an unbound number of threads at runtime, we always create two processes for each thread class in order to be detected interactions between different instances of the same thread class. Furthermore, since we do not use any *May-Happens-In-Parallel* analysis, we take a pessimistic approach and assume that all the generated threads may execute in parallel and may interleave each other, i.e., a transaction from one thread may execute between any two transactions from other thread.

The process analysis works by traversing the program call-graph using a pre-order depth-first strategy while maintaining the current process information. The analysis begins at the `main` method, with the P_{main} as the current process. Whenever an edge is found to a *run* method, from a class that extends the `Thread` class, or inherits the `Runnable` interface, is created a new process that becomes the current process of the analysis. Furthermore, whenever an atomic method is found, we associate it to the current process. Hence, for each process we collect the sequence of

Algorithm 2: Function **analyseNode**

```
Input: method, process, visited[]
Result: void
foreach Edge e : outOf(method) do
    if !visited.contains(process, target(e)) then
        visited.add(process, target(e));
        if isThreadCreationEdge(e) then
            process = createProcess(target(e));
            analyseNode(target(e), process, visited);
        else
            if isAtomicMethod(target(e)) then
                associate(process, target(e));
            else
                analyseNode(target(e), process, visited);
            end
        end
    end
end
```

```
protected static List<Integer> list;
public static void main(String[] args) {
  Random r = new Random();
  int value = r.nextInt();

  if(value % 2 == 0)
    list = new ArrayList<Integer>();
  else
    list = new LinkedList<Integer>();

  list.add(1);
}
```

Figure 3.12: Dynamic dispatch example.

atomic methods that may be executed by such process.

In Algorithm 2 we present the algorithm that computes the set of processes of a program. Function **analyseNode** is first called for the `main` method, with P_{main} process and with a visited empty set. The result of this analysis is a set of processes, where each contains a sequence of atomic methods.

3.5.2 Instance Type Analysis

The analysis to compute the causal dependencies, and to compute the views, are inter-procedural analysis, i.e., are able to analyze method invocations. To analyze a method invocation, we need to know which class has the method's implementation. This could be a problem in languages that support dynamic dispatch, such as the Java programming language.

Consider the example shown in Figure 3.12. In this example variable `list` is declared with type `List`, and randomly is either instantiated as an `ArrayList` or a `LinkedList`. When the `add` method is invoked, it is not possible at compile-time to know which class was used to instantiate variable `list`, and as a consequence, we do not know what is the implementation of the `add` method.

To overpass this difficulty we developed a simple data-flow analysis that computes an over-approximation of the set of possible classes that were used to instantiate a particular variable. We can then use this information to annotate the method invocation nodes of the control-flow graph with the set of classes that may implement the method being invoked. The analysis that use this information, such as the causal dependency analysis, and the views analysis, analyze the methods of each class in the set and join the results. For instance, in the views analysis, we join the accesses of all method's implementations.

In Figure 3.13 we show the rules of this analysis, which we call type instance analysis. The rules are define using a reduction relation $\langle \mathcal{CS}, S \rangle \Longrightarrow \langle \mathcal{CS}' \rangle$, over an abstract state composed by a map from variables to a set of classes: $\mathcal{CS} \subseteq \mathsf{VarClass} \equiv \mathsf{Vars} \times \mathsf{Classes}$.

All the assignment rules, ASSIGN, HEAP READ, HEAP WRITE, and METH CALL, use two auxiliary functions: $\mathsf{impl} : \mathcal{P}(\mathsf{VarClass}) \times \mathsf{Vars} \longrightarrow \mathcal{P}(\mathsf{Classes})$, which returns the set of classes associated with a given variable, and $\mathsf{kill} : \mathcal{P}(\mathsf{VarClass}) \times \mathsf{Vars} \longrightarrow \mathcal{P}(\mathsf{VarClass})$, which removes from the state the implementations for the given variable. We define these functions below:

Definition 3.12 (Implements Function)**.** *Given the current state $\mathcal{CS}$ and a variable $v \in \mathsf{Vars}$.*

$$\mathsf{impl}(\mathcal{CS}, v) \triangleq \{c \mid (v, c) \in \mathcal{CS}\}$$

This function is sometimes used with an expression e. If the expression corresponds to the null value then the result of this function is the empty set.

Definition 3.13 (Kill Function)**.** *Given the current state $\mathcal{CS}$ and a variable $v \in \mathsf{Vars}$.*

$$\mathsf{kill}(\mathcal{CS}, v) \triangleq \mathcal{CS} \setminus \{(v, c) \mid (v, c) \in \mathcal{CS}\}$$

In the METH CALL rule, we need to propagate the implementation classes to the assigned variable x. The retVar is a special variable that corresponds to the returning variable of function *func*, and which has all the associated implementation classes.

In the CONDITIONAL and LOOP rules, we join the states of all branches. This rule solves the problem identified in the example shown in Figure 3.12. At the invocation point of statement `list.add(1)`, our analysis identifies the set of possible instantiation classes for variable `list`, which in this case is { `ArrayList` , `LinkedList` }.

3.5.3 Native Methods

During the analysis we might encounter methods that are implemented in a different language and are accessed through Java runtime using a JNI[3] interface. Since our prototype can only analyze Java bytecode, it cannot analyze these methods. Furthermore, the same problem may happen if we do not have access to the code of some java library. Our solution to this problem was a conservative one, and is specific to each analysis that needs to analyze methods' code.

Causality Dependency Analysis When analyzing a native method, or a method belonging to some missing library, we consider that the returning value, if it exists, depends from all the method's parameters. This conservative approach allows to not lose any dependency path even if the value of some variable enters into a native method.

[3]Java Native Interface

$$\frac{\langle \mathcal{CS}, S_1 \rangle \Longrightarrow \langle \mathcal{CS}' \rangle \qquad \langle \mathcal{CS}', S_2 \rangle \Longrightarrow \langle \mathcal{CS}'' \rangle}{\langle \mathcal{CS}, S_1; S_2 \rangle \Longrightarrow \langle \mathcal{CS}'' \rangle} (\textsc{Seq})$$

$$\frac{\mathcal{CS}' = \mathsf{kill}(\mathcal{CS}, x) \cup \{(x, c) \mid c \in \mathsf{impl}(\mathcal{CS}, e)\}}{\langle \mathcal{CS}, x := e \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Assign})$$

$$\frac{c' = \mathsf{typeof}(y) \qquad \mathcal{CS}' = \mathsf{kill}(\mathcal{CS}, x) \cup \{(x, c) \mid c \in \mathsf{impl}(\mathcal{CS}, (c', f))\}}{\langle \mathcal{CS}, x := y.f \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Heap Read})$$

$$\frac{c' = \mathsf{typeof}(x) \qquad \mathcal{CS}' = \mathsf{kill}(\mathcal{CS}, (c', f)) \cup \{((c', f), c) \mid c \in \mathsf{impl}(\mathcal{CS}, e)\}}{\langle \mathcal{CS}, x.f := e \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Heap Write})$$

$$\frac{\mathcal{CS}' = \mathsf{kill}(\mathcal{CS}, x) \cup \{(x, C)\}}{\langle \mathcal{CS}, x := \mathsf{new}\, C() \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Allocation})$$

$$\frac{\mathsf{spec}(\mathit{func}) = \mathcal{CS}_f \qquad \mathcal{CS}' = \mathcal{CS}_f \cup \mathsf{kill}(\mathcal{CS}, x) \cup \{(x, c) \mid c \in \mathsf{impl}(\mathcal{CS}, \mathsf{retVar})\}}{\langle \mathcal{CS}, x := \mathit{func}(\vec{y}) \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Meth Call})$$

$$\frac{\langle \mathcal{CS}, S_1 \rangle \Longrightarrow \langle \mathcal{CS}' \rangle \qquad \langle \mathcal{CS}, S_2 \rangle \Longrightarrow \langle \mathcal{CS}'' \rangle}{\langle \mathcal{CS}, \mathsf{if}\, e\, \mathsf{then}\, S_1\, \mathsf{else}\, S_2 \rangle \Longrightarrow \langle \mathcal{CS}' \cup \mathcal{CS}'' \rangle} (\textsc{Conditional})$$

$$\frac{\langle \mathcal{CS}, S \rangle \Longrightarrow \langle \mathcal{CS}' \rangle}{\langle \mathcal{CS}, \mathsf{while}\, e\, \mathsf{do}\, S \rangle \Longrightarrow \langle \mathcal{CS} \cup \mathcal{CS}' \rangle} (\textsc{Loop})$$

$$\frac{\mathcal{CS}' = \mathcal{CS} \cup \{(\mathsf{retVar}, c) \mid c \in \mathsf{impl}(\mathcal{CS}, e)\}}{\langle \mathcal{CS}, \mathsf{return}\, e \rangle \Longrightarrow \langle \mathcal{CS}' \rangle} (\textsc{Return})$$

$$\frac{}{\langle \mathcal{CS}, \mathsf{skip} \rangle \Longrightarrow \langle \mathcal{CS} \rangle} (\textsc{Skip})$$

Figure 3.13: Type instance analysis rules.

```
class MathComputer{
  /**
   * This method returns the maximum between x1 and x2
   * @param x1 first number
   * @param x2 second number
   * @return the maximum of both numbers
   */
  public native int getMax(int x1, int x2);
}
```

```
<class id="MathComputer">
  <method id="getMax(int,int)">
    <reads>this</reads>
    <writes>this</writes>
    <reads>0</reads>
    <writes>0</writes>
    <reads>1</reads>
    <writes>1</writes>
  </method>
</class>
```

(a) Automatic generated specification.

```
<class id="MathComputer">
  <method id="getMax(int,int)">
    <reads>0</reads>
    <reads>1</reads>
  </method>
</class>
```

(b) User assisted specification.

Figure 3.14: Example of a native method XML specification.

Views Analysis The analysis of a method with an unavailable implementation is done by assuming that such method accesses, for read and write, all method's parameters, and the **`this`** variable. Since, this is a very conservative approach we generate an XML specification for each native method that includes these conservative assumptions, and we allow the programmer to modify this specification in order give more precise information to system. In Figure 3.14 is shown an example of a specification for the native method `getMax`. In the specification, method parameters are identified by integers that correspond to the order of declaration. In Figure 3.14a is shown the specification generated by our system, and in Figure 3.14b is shown the specification corrected by the programmer after reading the documentation of the native method.

3.6 Evaluation

Besides comparing our results with those reported in the literature for individual benchmarks, we did an exhaustive comparison with two other approaches: the work of Artho et al [AHB03], because our approach is an extension of Artho's work; and the work of Teixeira et al [TLFDS10], because their results are currently a reference for the field. The results presented were obtained by running our tool with the algorithms described in this Chapter; by using Artho et al's algorithm implemented with static analysis techniques (rather than the dynamic analysis reported in [AHB03]); and by running Teixeira's tool on the Java source (instead of the Bytecode).

Tables 3.1 and 3.2 summarize the results achieved by applying our tool to a set of benchmarking programs, most of them well known from related works and compares them with the two works cited above. Teixeira's tool was unable to process some of the benchmarks, so they are reported in a separate second set. Columns *AV* indicate the number of known atomicity violations, *false negatives* indicate the number of known program atomicity violations that were missed by the

Table 3.1: Results for benchmarks

Tests	AV	False Negatives			False Positives			Acc. Vars	LOC	Time (sec.)
		MoTH	Artho	Teix.	*MoTH*	Artho	Teix.			
Connection [BBA08]	2	0	1	1	0	0	1	34	112	45
Coord03 [AHB03]	1	0	0	0	0	0	3	13	170	43
Local [AHB03]	1	0	1	0	0	0	1	3	33	42
NASA [AHB03]	1	0	0	0	0	0	0	7	121	43
Coord04 [AHB04]	1	0	0	0	0	0	3	7	47	40
Buffer [AHB04]	0	0	0	0	1	0	7	8	64	41
DoubleCheck [AHB04]	0	0	0	0	1	0	2	7	51	41
StringBuffer [FF04]	1	0	1	1	0	0	0	12	52	44
Account [vG03]	1	0	1	0	0	0	0	3	65	40
Jigsaw [vG03]	1	0	0	0	0	0	1	33	145	40
OverReporting [vG03]	0	0	0	0	0	0	2	6	52	42
UnderReporting [vG03]	1	0	1	0	0	0	0	3	31	39
Allocate Vector [Ibm]	1	0	1	0	0	0	1	24	304	41
Knight [TLFDS10]	1	0	1	0	0	0	2	10	223	41
Arithmetic Database [TLFDS10]	3	0	3	1	1	0	0	24	416	54
Total	**15**	**0**	**10**	**3**	**3**	**0**	**23**	–	–	–

Table 3.2: Results for benchmarks

Tests	AV	False Negatives		False Positives		Acc. Vars	LOC	Time (sec.)
		MoTH	Artho	MoTH	Artho			
Elevator [vG03]	16	0	16	6	4	39	558	46
Philo [vG03]	0	0	0	2	0	9/594	96	45/612
Tsp [vG03]	0	0	0	2	0	635	795	869
Store	2	0	1	0	1	44/608	901	149/1763
Total	**18**	**0**	**17**	**10**	**5**	–	–	–

approach[4], *false positives* indicate the number of reported but non-existing atomicity violations, *Acc. Vars* indicate the number of variables accessed inside atomic regions and is an indication of the problem size, together with the number of *LOC*, and how long it took for our analysis to run.

In the case of Table 3.2, the benchmarks Philo and Store have two different values for *accessed variables* and *time*. The second values report on the original benchmarks, which includes some (non-essential) calls to I/O methods in the JDK library. The first values report on a tailored version of the benchmarks where those calls to the JDK library were commented.

For the benchmarks listed in Table 3.1, our approach revealed a very high accuracy by reporting no false negatives and only three false positives. The false positive in the Buffer benchmark is due to an assumption claim from its authors that is not implemented in the actual code. The information collected by the Causal Dependency Analysis is incomplete and imprecise and originates false positives in the Double Check and Arithmetic Database benchmarks while checking for stale-value errors, which are not detected by Artho et al's approach.

For the benchmarks listed in Table 3.2, our approach again revealed very high accuracy. Although it reported 10 false positives (vs. only 5 from Artho et al's), it reported zero false negatives (vs. 17 from Artho et al's). These benchmarks also indicate that our algorithms scale well with the the size of the problem, both in the number of accessed variables inside the atomic blocks and the number of lines of code.

[4]The identification of false negative is only possible because the sets of atomicity violations in the benchmarking programs are well known.

3.7 Related Work

Several past works have addressed the detection of the same class of atomicity violations in concurrent programs as addressed in this work.

The work from Artho et al. [AHB03] introduces the concept of *view consistency*, to detect high-level data races. A *view* of an atomic block is a set containing all the shared variables accessed (both for reading and writing) within that block. The *maximal views* of a process are those views that are not a subset of any other view. Intuitively, a maximal view defines a set of variables that should always be accessed atomically (inside the same atomic block). A program is free from high-level data races if all the views of one thread that are a subset of the maximal views from another thread form an inclusion chain among themselves.

Our work builds on the proposal from Artho et al. [AHB03], but we extend it by incorporating the type of memory access (read or write) into the views, and refine the rules for detecting high-level data races to combine this additional information with the information given by the *causal* dependencies: Our refinement of the rules has a considerable positive impact in the precision of the algorithm, as demonstrated in Section 3.6.

Praun and Gross [vG03] introduce *method consistency* as an extension of view consistency. Based on the intuition that the variables that should be accessed atomically in a given method are all the variables accessed inside a synchronized block, the authors define the concept of *method views* that relates to Artho's maximal views. A method view aggregates all the shared variables accessed in a method, differentiating read and write memory accesses. Similarly to ours, this approach is more precise than Artho's because it also detects stale-value errors. Our algorithm however has higher precision than Praun's and give less false positives, as we use *maximal views* rather than *method views*.

Wang and Stoller [WS03] use the concept of *thread atomicity* to detect and prevent data races, where thread atomicity guarantees that all concurrent executions of a set of threads is equivalent to a sequential execution of those threads. In an attempt to reduce the number of false positives yield by Wang and Stoller [WS03], Teixeira et al. [TLFDS10] proposed a variant of this algorithm based in the intuition that the majority of the atomicity violations come from two consecutive atomic blocks that should be merged into a single one. The authors detect data races by defining and detecting some anomalous memory access patterns for both high-level data races and stale-value errors. Our approach may be seen as a generalization of this concept of memory access patterns, but in our case supported by the notion of *causal* dependencies between variables, which allow to reduce considerably the number of both false negatives and false positives.

Other related approaches include Flanagan et al. [FQ03] work that proposes a type system that verifies the *atomicity* of code blocks. This concept is further explored by Beckman et al. [BBA08], which present an intra-procedural static analysis formalized as a type system, based in the concept of *access permissions* to detect data races. Contrarily to our approach, this work demands that the programmer explicitly declares the access permissions and invariants for the objects in the program.

Vaziri et al. [VTD06] proposes a new definition for the concept of data race, through the theoretical assemblage of all possible anomalous memory access patterns, including both low- and high-level data races. Although this work shares with ours the goals in detecting atomicity violations, it grounds on a completely different concurrent programming model where locks are

associated directly with program variables and not with code statements, which make it impractical to use with traditional programming languages such as Java.

3.8 Concluding Remarks

In this Chapter we presented a novel approach to detect high-level data races and stale-value errors in concurrent programs. The proposed approach relies on the notion of *causal* dependencies to improve the precision of previous detection techniques. The high-level data races are detected using an algorithm based on a previous work by Artho et al. refined to distinguish between read and write accesses and extended with the information given by the *causal* dependencies. The stale-value errors are detected using the information given by the *causal* dependencies, which exposes the values of variables that escaped an atomic block and entered into another atomic block.

Our detection analysis still remains unsound mainly due to the absence of pointer analysis and to the way that *views* are computed. But these design decisions allowed us to maintain the scalability of our approach without incurring in a strong precision loss, as our experimental results confirm.

We evaluated our analysis techniques with well known examples from the literature and compared them to previous works. Our results show that we are able to detect all atomicity violations present in the examples, while reporting a low number of false positives.

Publications The contents of this chapter were partially published in:

- [PDLFS11] **Practical verification of transactional memory programs**. Vasco Pessanha, Ricardo J. Dias, João M. Lourenço, Eitan Farchi, and Diogo Sousa. In proceedings of PADTAD 2011 (Workshop), July 2011.
- [DPL12] **Precise detection of atomicity violations**. Ricardo J. Dias, Vasco Pessanha, and João M. Lourenço. In proceedings of Haifa Verification Conference 2012, November 2012.

4

Verification of Snapshot Isolation Anomalies

In this chapter we describe a verification technique to certify that a multi-threaded Java program, using transactional memory with snapshot isolation, is free from write-skew anomalies. This technique resorts to a shape analysis based on separation logic to model memory updates performed by transactions.

4.1 Introduction

Full-fledged Software Transactional Memory (STM) [ST95; HLMWNS03] usually provides strict isolation between transactions and opacity semantics. Alternative relaxed semantics approaches, based on weaker isolation levels that allow transactions to interfere and to generate non-serializable execution schedules, are known to perform considerably better in some cases. The interference among non-serializable transactions are commonly known as *serializability anomalies* [BBGMOO95].

Snapshot Isolation (SI) [BBGMOO95] is a well known relaxed isolation level widely used in databases, where each transaction executes with relation to a private copy of the system state — a snapshot — taken at the beginning of the transaction. All updates to the shared state are kept pending in a local buffer (the transaction *write-set*). If the transaction succeeds, the pending updates are committed in the global state. Reading modified items always refer to the pending values in the local buffer. Committing transactions always obey the general *First-Commiter-Wins* rule. This rule states that a transaction A can only commit if no other concurrent transaction B has committed modifications to data items pending to be committed by transaction A (i.e., the write-sets of A and B were not disjunct). Hence, for any two concurrent transactions modifying a common data item, only the first one to commit will succeed.

Although appealing for performance reasons, SI may lead to non-serializable executions, resulting in a serializability anomaly called *write-skew*. The following example illustrates the occurrence of this anomaly.

$$x := x + y \quad || \quad y := y + x$$

When the above statements are executed in concurrent transactions, whose write-sets are disjoint, it is possible to find a non-serializable trace of execution in which both transactions commit and yield unexpected results. In general, the write-skew anomaly occurs when two transactions are writing on disjoint memory locations (x and y in the example above) and also reading the data that is being modified by the other.

Tracking memory operations adds some overhead to the computations. TM systems running under opacity must track both memory read and write accesses, thus incurring in considerable performance penalties. As the validation of transactions under SI only requires the checking of conflicting updates to shared data items by concurrent transactions, an SI-based runtime system may ignore memory read accesses and only track the write accesses and check for write-write conflicts. Hence, the use of SI may boost the overall performance of the transactional runtime.

In the remainder of this chapter, we use the terms serializability and opacity interchangeably, as in the context of transactional memory, opacity is the default strong consistency criteria.

4.1.1 Motivation

To validate our hypothesis of performance boosting of Transactional Memory by using SI instead of serializability [DLP11], we adapted JVSTM [CRS06], a multi-version STM, to support SI and ran some micro benchmarks (a Linked List and a Skip List) to compare the throughput of running memory transactions in serializability and SI. The list implementations in both micro benchmarks suffer from the write-skew anomaly when running under SI, triggered by the concurrent execution of transactions executing the *insert* and *remove* operations. Because the programmer is expecting the resulting computation to be serializable, we created and evaluated a second version of each micro benchmark where the write-skew anomaly was corrected with dummy-writes.

Figure 4.1 depicts the results of the execution of the Linked List and Skip List micro benchmarks, with a maximum of 10 000 keys, for two workloads that differ in the number of update—insert and remove—transactions executed, with approximately 50% and 90% of updates. The tests were performed in a Sun Fire X4600 M2 x64 server, with eight dual-core AMD Opteron Model 8220 processors @ 2.8 GHz and 1024 KB of cache in each processor.

The serializable isolation variant corresponds to the original JVSTM algorithm. Since the JVSTM is a multi-version STM, the read-only transactions are already highly optimized and have similar performance to the read-only transactions in the snapshot isolation variant. The observed performance improvement for SI depends only on the read-write (RW) transactions. The original JVSTM must keep track of the memory read accesses in RW transactions to validate the transaction at commit time, while the SI variant never tracks the memory read accesses. This performance gain is higher when the frequency of updates increases. The SI variant performs better than the serializable for both micro benchmarks, and scales much better for the Linked List than for the Skip List. This is due to the internal structure and organization of each data structure. The Skip List has a low read-write conflict rate and thus the benefits of only detecting write-write conflicts are limited.

Another important fact of these results is that the corrected version of the snapshot isolation,

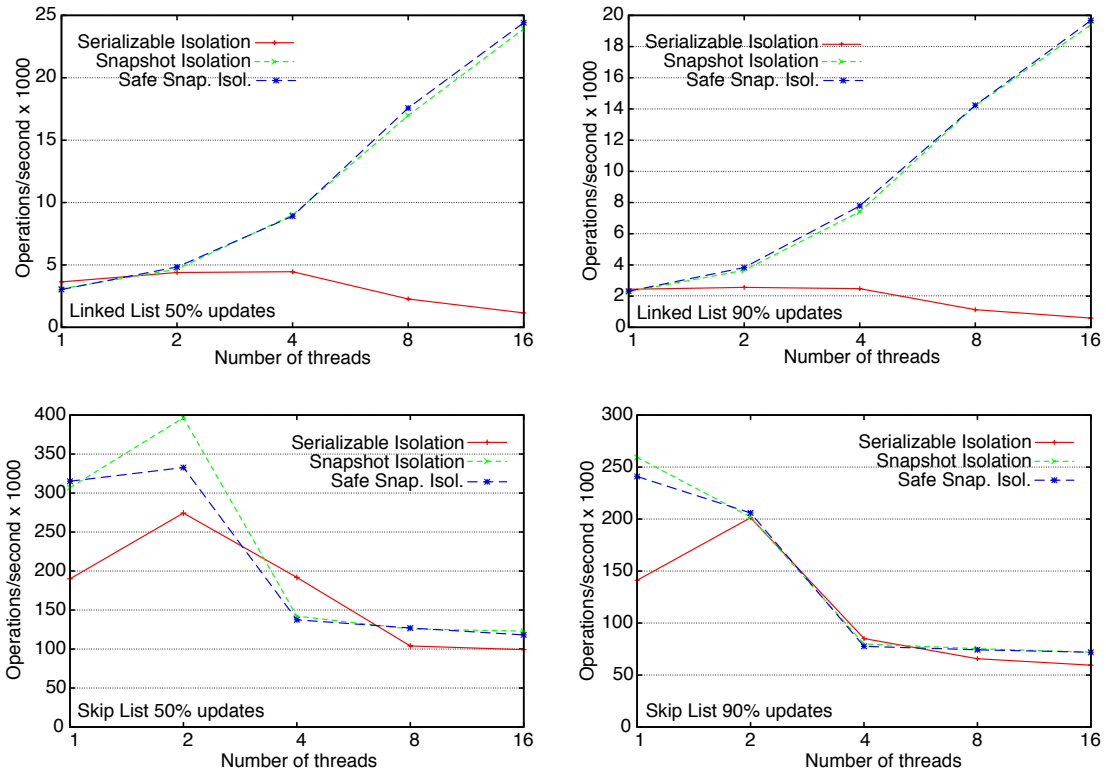


Figure 4.1: Linked List (top) and Skip List (bottom) performance throughput benchmarks with 50% and 90% of write operations.

which is anomaly-free, has almost the same performance as the non-safe version. In this case the correction was a single dummy write introduced in the remove operation in both benchmarks.

These results on performance and scalability confirmed the potential performance benefits of using snapshot isolation in STM. To acquire some additional insight about the potential performance gains of using SI in the Distributed Software Transactional Memory (DSTM) setting, we calculated the size of the read and write-sets for each variant. The size of the read- and write-sets is directly related to the network traffic generated by the DSTM runtime, hence we can extrapolate on the potential impact of using SI with DSTM.

Table 4.1 depicts the average and maximum size of the read- and write-sets for the execution of the three variants of the Linked List and Skip List micro benchmarks, with a maximum of 10 000 keys and 90% of write operations. The SI variants always have empty read-sets. For the Linked List under JVSTM, the read-set has an average size of 1992.9 entries. This result clearly depends on the nature of the benchmark application. In the case of the Linked list, to insert a node in the middle of the list, one has to traverse all nodes until the right position, implying larger read sets. The average size of the write-sets for all variants in both data structures is almost the same. The small difference between the two SI variants is due to the dummy write introduced in the safe version of both data structures.

These preliminary results were encouraging, and we pursued with our goal of using static analysis to allow the transactional runtime to safely use snapshot isolation while providing serialization semantics.

Linked List

	Read-Set		Write-Set		Traffic
	Avg	Max	Avg	Max	
Serializable	1992.9	4929	1.6	2	100%
Snap. Isol.	0.0	0	1.6	2	0.08%
Safe Snap. Isol.	0.0	0	2.0	2	0.1%

Skip List

	Read-Set		Write-Set		Traffic
	Avg	Max	Avg	Max	
Serializable	36.3	103	2.0	20	100%
Snap. Isol.	0.0	0	2.0	22	5.2%
Safe Snap. Isol.	0.0	0	2.6	22	6.8%

Table 4.1: Read- and write-set statistic per transaction for a Linked List (top) and a Skip List (bottom).

4.1.2 Verification of Snapshot Isolation Anomalies

In this chapter we propose a verification technique for STM Java programs that statically detects if any two transactions may cause a *write-skew* anomaly when executed concurrently. This verification technique may be used to optimize the execution of STM programs, by providing a serializable semantics to the program while letting the STM runtime mix the serializability and SI semantics, and use the latter whenever possible. The verification technique performs deep-heap analysis (also called shape analysis) based on separation logic [Rey02; DOY06] to compute memory locations in the read- and write-sets for each distinguished transaction in a Java program. The analysis can automatically compute *loop invariants* and only requires the specification of the state of the heap at the beginning of each transaction. Read and write-sets of transactions are then computed using *heap paths*, which capture dereferences through field labels, choice and repetition.

For instance, a *heap path* of the form $x.(left \mid right)^*.right$ describes the access to a field labeled $right$, on a memory location reachable from variable x after a number of dereferences through the $left$ or $right$ fields.

StarTM is a tool that implements the proposed verification technique, and analyzes Java Bytecode programs extended with STM annotations. StarTM was then validated with a transactional Linked List and a transactional Binary Search Tree, and also with a Java implementation of the STAMP Intruder benchmark [CMCKO08]. Our results show evidence that i) supporting the arguments of [RFF06], it is possible to safely execute concurrent transactions of a Linked List under snapshot isolation with noticeable performance improvements; ii) it is possible to build a transactional insert method in a Binary Search Tree that is safe to execute under SI; and iii) our automatic analysis of the STAMP Intruder benchmark found a new *write-skew* anomaly in the existing implementation.

Our technique can verify programs for the absence of *write-skew* anomalies only between pairs of transactions and if the program use acyclic data structures, such as tree-like data structures. The limitation of only detecting anomalies generated by pairs of transactions, precludes the detection of the SI read-only anomaly, which can only occur with at least three concurrent transactions. This

limitation can be solved by extending the verification model, although for the sake of simplicity we use a model to detect anomalies generated by only pairs of transactions.

The main contributions of the work described in this chapter are:

- The first program verification technique to statically detect the *write-skew* anomaly in transactional memory programs;
- The first verification technique to certify the absence of *write-skew* anomalies in transactional memory programs, even in presence of deep-heap manipulation, thanks to the use of shape analysis techniques based in separation logic;
- A model that captures fine-grained manipulation of memory locations based on *heap paths*; and
- An implementation of our technique in a software tool and its application to a set of intricate examples.

The remainder of the chapter describes the theory of our analysis technique and the validation experiments. We start by introducing the fundamental concepts of snapshot isolation in Section 4.2, and present the abstract write-skew condition in Section 4.3. Then we describe a step-by-step example of applying StarTM to a simple example in section 4.4. We then present the core language, in section 4.5, and the abstract domain for the analysis procedure in section 4.6. In section 4.7, we present the symbolic execution of programs against the abstract state representation. We finalize the chapter by presenting some experimental results in Section 4.8 and comparing our approach with others in Section 4.9.

4.2 Snapshot Isolation

Snapshot Isolation [BBGMOO95] is a relaxed isolation level where each transaction executes with respect to a private copy of the system state, taken in the beginning of the transaction and stored in a local buffer. All write operations are kept pending in the local buffer until they are committed in the global state. Reading modified items always refers to the pending values in the local buffer.

Considering that the lifetime of a successful transaction is the time span that goes from the moment it starts $start(T_i)$ to the moment it commits $commit(T_i)$. Two successful transactions T_1 and T_2 are said to be concurrent if:

$$[start(T_1), commit(T_1)] \cap [start(T_2), commit(T_2)] \neq \emptyset$$

During the execution of a transaction T_i, its write operations are not visible to any other concurrent transactions. When any transaction T_i is ready to commit, it obeys the *First-Commiter-Wins* rule, which states a transaction T_i can only commit if no other concurrent transaction T_k $(i \neq k)$ has committed modifications to data items pending to be committed by transaction T_i. Hence, for any two concurrent transactions modifying the same data item, only the first one to commit will succeed.

One of the significant advantages of the relaxed snapshot isolation level over serializability is that read-write conflicts are ignored. This could allow significant performance improvements in workloads with high contention between transactions.

```
1 void Withdraw(boolean b, int value) {
2   if (x + y > value)
3     if (b) x = x - value;
4     else y = y - value;
5 }
```

Figure 4.2: `Withdraw` program.

Although appealing for performance reasons, the application of SI may lead to non-serializable executions, resulting in a *write-skew* consistency anomaly [FLOOS05]. Consider the following example that suffers from the *write-skew* anomaly. A bank client can withdraw money from two possible accounts represented by two shared variables, x and y. The program listed in Figure 4.2 can be used in several transactions to perform bank operations customized by its input values. The behavior is based on a parameter b and on the sum of the two accounts. Let the initial value of x be 20 and the initial value of y be 80. If two transactions execute concurrently, one calling the `Withdraw(true, 30)` (T_1) and the other calling the `Withdraw(false, 90)` (T_2), then one possible execution history of the two transactions under SI is:

$$H = R_1(x, 20)\ R_2(x, 20)\ R_1(y, 80)\ R_2(y, 80)\ R_1(x, 20)\ W_1(x, -10)\ C_1 \\ R_2(y, 80)\ W_2(y, -10)\ C_2$$

After the execution of these two transactions the final sum of the two accounts will be -20, which is a negative value. Such execution would never be possible under serializable isolation level, as the last transaction to commit would abort because it read a value that was written by the first transaction.

4.3 Abstract Write-Skew

The write-skew anomaly can be defined by the existence of a cycle in a dependency serialization graph (Section 2.3.3), but in this particular work we will define it has a condition over the read- and write-sets of transactions. Although a write-skew may be triggered by the interaction of three or more transactions, we limit the detection of write-skews to only two transactions. By opting for only detecting write-skews between pairs of transactions, we can define the write-skew as a logical condition over the read- and write-sets of the two transactions, without using the dependency serialization graph. A write-skew occurs when two transactions are writing on disjoint memory locations but are also reading data that is being modified by the other. We formalize this description in the following definition:

Definition 4.1 (Concrete Write-Skew)**.** *Let* T_1 *and* T_2 *be two transactions, and let* $\mathcal{R}_i^c$ *and* $\mathcal{W}_i^c$ $(i = 1, 2)$ *be their corresponding read- and write-sets. There is a write-skew anomaly if*

$$\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset$$

If the above condition is true for any concurrent execution of two transactions T_1 and T_2, then there was a write-skew anomaly and the application state may have become inconsistent.

The purpose of this work is to prevent these situations by statically verifying the application

code and check if any two transactions may generate a write-skew anomaly when executed concurrently. Since the detection of write-skew anomalies only depends on the read- and write-sets of the two concurrent transactions, and as this sets are local to each transaction, we do not need to consider how the application threads interact with each other, and the code for each transaction can be verified separately.

A single transaction, declared in the application code, may generate different read- and write-sets each time it is executed. The verification technique we propose computes over- and under-approximations of the read- and write-set of each transaction, and use these approximations to check the satisfiability of an abstract write-skew condition, as defined below:

Definition 4.2 (Abstract Write-Skew)**.** *Let* T_1 *and* T_2 *be two transactions, and let* $\mathcal{R}_i$, $\mathcal{W}_i^{>}$ *and* $\mathcal{W}_i^{<}$ $(i = 1, 2)$ *be their corresponding abstract over-approximated read-, over-approximated (*may*) write- and under-approximated (*must*) write-sets. There is a write-skew anomaly if*

$$\mathcal{R}_1 \cap \mathcal{W}_2^{>} \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{>} \cap \mathcal{R}_2 \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{<} \cap \mathcal{W}_2^{<} = \emptyset$$

If the above condition is satisfiable then a write-skew anomaly may exist at runtime, otherwise the concurrent execution of these two transactions T_1 and T_2 will never generate a write-skew anomaly.

4.3.1 Soundness

Our approach is sound for the detection of the *write-skew* anomaly between pairs of transactions. We claim that, by analyzing the satisfiability test described in Definition 4.2, if no *write-skew* anomaly is detected by our algorithm then there is no possible execution of the program that contains a *write-skew*.

The question then remains whether an occurrence of a write-skew condition at runtime is captured by our test. To see this, let's assume that $\mathcal{R}_1^c$, $\mathcal{W}_1^c$, $\mathcal{R}_2^c$, $\mathcal{W}_2^c$ are concrete, exact read- and write- sets for transactions T_1 and T_2. Notice that a write-skew condition occurs between T_1 and T_2 if

$$\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset$$

Our static analysis computes abstract over-approximations of read-sets ($\mathcal{R}_1$ and $\mathcal{R}_2$), write-sets ($\mathcal{W}_1^{>}$ and $\mathcal{W}_2^{>}$), and under-approximation of write-sets ($\mathcal{W}_1^{<}$ and $\mathcal{W}_2^{<}$), which are related to the concrete read- and write-sets as follows:

$$\mathcal{R}_1^c \subseteq \mathcal{R}_1, \quad \mathcal{R}_2^c \subseteq \mathcal{R}_2, \quad \mathcal{W}_1^c \subseteq \mathcal{W}_1^{>}, \quad \mathcal{W}_2^c \subseteq \mathcal{W}_2^{>}, \quad \mathcal{W}_1^{<} \subseteq \mathcal{W}_1^c, \quad \mathcal{W}_2^{<} \subseteq \mathcal{W}_2^c$$

These set relations allow us to prove that the condition on abstract sets is implied by the condition on concrete sets:

$$\begin{aligned}(\mathcal{R}_1^c \cap \mathcal{W}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{R}_2^c \neq \emptyset \quad \wedge \quad \mathcal{W}_1^c \cap \mathcal{W}_2^c = \emptyset) \Rightarrow \\ (\mathcal{R}_1 \cap \mathcal{W}_2^{>} \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{>} \cap \mathcal{R}_2 \neq \emptyset \quad \wedge \quad \mathcal{W}_1^{<} \cap \mathcal{W}_2^{<} = \emptyset)\end{aligned}$$

Hence we can conclude that if a real write-skew exists in an execution this will be detected by our test, which implies that our method is sound. The implication above also shows that our method may present false positives, i.e., it may detect a write-skew that will never occur at runtime. This is a classical unavoidable effect of conservative methods based on abstract interpretation.

4.4 StarTM by Example

StarTM analyzes Java multithreaded programs that make use of memory transactions. The scope of a memory transaction is defined by the scope of a Java method annotated with `@Atomic`. In our case, this `@Atomic` annotation requires a mandatory argument with an abstract description of the initial state of the heap. Other methods called inside a transactional method do not require this initial heap-state description, as it is automatically computed by the symbolic execution.

To describe the abstract state of the heap, we use a subset of separation logic formulae composed of a set of predicates — among which a *points-to* ($\mapsto$) predicate — separated by the special separation conjunction ($*$) typical of separation logic. The user can define new predicates in a proper scripting language and also define abstraction functions which, in case of infinite state spaces, allows the analysis to converge. An abstraction function is defined by a set of abstraction rules as in the jStar tool [DPJ08]. The user defined predicates and abstraction rules are described in separate files and are associated with the transactions' code by the class annotations `@Predicates` and `@Abstractions`, which receive as argument the corresponding file names.

We use as running example the implementation of an ordered singly linked list, adapted from the Deuce [KSF10] samples, shown in Figure 4.3. The corresponding predicates and abstractions rules are defined in Figure 4.4.

The predicate `Node(x,y)`, which is defined in Figure 4.4 as

$$Node(x, y) \Leftrightarrow x \mapsto [next : y]$$

is valid if variable x points to a memory location where the corresponding $next$ field points to the same location as variable y, or both the $next$ field and y point to nil. Predicate `List(x,y)`, which is defined as

$$List(x, y) \Leftrightarrow x \neq y \wedge (Node(x, y) \vee \exists z'.\, Node(x, z') * List(z', y))$$

is valid if variables x and y point to distinct memory locations and there is a chain of nodes leading from the memory location pointed by x to the memory location pointed by y. The predicate is also valid when both y and the last node in the chain point to nil.

In Figure 4.3, we annotate the `add(int)` and `remove(int)` methods as transactions with the initial state described by the following formula:

```
| this->[head:h'] * List(h',nil)
```

This formula states that variable **`this`** points to a memory location that contains an object of class `List`, and whose field *head* points to the same memory location pointed by the existential variable[1] h', which is the entry point of a list with at least one element.

StarTM performs an inter-procedural symbolic execution of the program. The abstract domain used by the symbolic execution is composed by a separation logic formula describing the abstract heap structure, and the abstract read- and write-sets. The abstract write-set is defined by two sets: a *may* write-set and a *must* write-set. As the naming implies one over-approximates, and the other under-approximates the concrete write-set. The abstract read-set is an over-approximation of the concrete read-set. The read- and write-sets are defined as sets of *heap paths*. A memory location

[1] Throughout this chapter we consider primed variables as implicitly existentially quantified.

```
1  @Predicates(file="list_pred.sl")
2  @Abstractions(file="list_abs.sl")
3  public class List {
4
5    public class Node{ ... }
6
7    private Node head;
8
9    public List() {
10     Node min = new Node(Integer.MIN_VALUE);
11     Node max = new Node(Integer.MAX_VALUE);
12     min.next = max;
13     head = min;
14   }
15
16
17   @Atomic(state= "| this -> [head:h'] * List(h', nil)")
18   public void add(int value) {
19     boolean result;
20     Node prev = head;
21     Node next = prev.getNext();
22     while (next.getValue() < value) {
23       prev = next;
24       next = prev.getNext();
25     }
26     if (next.getValue() != value) {
27       Node n = new Node(value, next);
28       prev.setNext(n);
29     }
30   }
31
32   @Atomic(state= "| this -> [head:h'] * List(h', nil)")
33   public void remove(int value) {
34     boolean result;
35     Node prev = head;
36     Node next = prev.getNext();
37     while (next.getValue() < value) {
38       prev = next;
39       next = prev.getNext();
40     }
41     if (next.getValue() == value) {
42       prev.setNext(next.getNext());
43     }
44   }
45 }
```

Figure 4.3: Order Linked List code.

```
// list_pred.sl file

/*** Predicate definition ***/
Node(x,n) <=> x -> [next:n] ;;

List(x,y) <=> x != y /\
          ( Node(x,y) \/ E z'. Node(x,z') * List(z',y) );;



// list_abs.sl file
/*** Abstractions definition ***/
Node(x, y') * Node(y',z)~~>List(x, z):
  y' nin context;
  y' nin x;
  y' nin z
;;
...
List(x,y') * Node(y',z)~~>List(x, z):
  y' nin context;
  y' nin x;
  y' nin z
;;
```

Figure 4.4: Predicates and Abstraction rules for the linked list.

is represented by its path, in terms of field accesses, beginning from some shared variable. We assume that the parameters of a transactional method and the instance variable `this` are shared in the context of that transaction.

The sample of the results of our analysis, depicted in Figure 4.5, includes two possible pairs of read- and write-sets for method `add(int)`. The *may* write-set is denoted by label `WriteSet>` and the *must* write-set is denoted by label `WriteSet<`. The first result has an empty write-set[2], and thus corresponds to a read-only execution of the method `add(int)`, where the *heap path* in the read-set can be interpreted as follows. The *heap path* `this.head.(next)[*A].next.value` asserts that method `add(int)` reads the *head* field from the memory location pointed by variable *this* and following the memory location pointed by *head* it reads the *next* field, then for each memory location it reads the *next* and *value* fields and hops to the next memory location through the *next* field. In the last memory location accessed it only reads the *value* field. In general, we can interpret the meaning of an abstract read-set as all the memory locations represented by the *heap paths* present in the read-set and also by their prefixes.

The star $(*)$ operator has always a label attached, in case of `[*A]`, the label is `A`. This label is used to identify the subpath guarded by the star and can be interpreted, in this case, as $A = (next)^*$. This label is existentially quantified in a pair of read- and write-sets.

The second pair of read- and write-sets of method `add(int)` in Figure 4.5 contains the same read-set and a different write-set. In this case the *may* and *must* write-sets are equal. The *heap path* `this.head.(next)[*B].next` asserts that the *next* field, of the memory location represented by the path $this.head.(next)^*_B$, was written.

It is important to notice that the interpretations of the read- and write-set are different. In the

[2]If the context is not ambiguous we will always refer to both the *may* and *must* write-sets.

```
# Method boolean add(int value)
Result 1:
ReadSet:   { this.head.(next)[*A].next.value }
WriteSet>: { }
WriteSet<: { }

Result 2:
ReadSet:   { this.head.(next)[*B].next.value }
WriteSet>: { this.head.(next)[*B].next }
WriteSet<: { this.head.(next)[*B].next }


# Method boolean remove(int value)
Result 1:
ReadSet:   { this.head.(next)[*C].next.value }
WriteSet>: { }
WriteSet<: { }

Result 2:
ReadSet:   { this.head.(next)[*D].next.value,
             this.head.(next)[*D].next.next }
WriteSet>: { this.head.(next)[*D].next }
WriteSet<: { this.head.(next)[*D].next }
```

Figure 4.5: Sample of StarTM result output for the Linked List example.

read-set we consider that all the path prefixes of all *heap path* expressions were read, while in the write-set we consider that there was a single write operation in the last field of each *heap path* expression.

The *may* write-set may contain *heap paths* of the form $this.head.(next)^{*}_{B}$. In this case, the interpretation of this expression is that the field $next$ is written in every memory location represented by the path $this.head.(next)^{*}_{B}$. More details on *heap path* expressions are given in Section 4.6.2.

The analysis also originates two possible results for method `remove(int)`. The first result for this method is similar to the first result for method `add(int)`. In the second result for method `remove(int)`, the field $next$ is read for all memory locations including the last memory location where field $value$ was accessed, since the star label is the same in the two *heap path* expressions in the read-set. The write-set is the same as in the `add(int)` method.

We can now check for the possible occurrence of a *write-skew* anomaly by testing the condition presented in Definition 4.2. We will consider that each result (a pair of a read- and a write-set) corresponds to a single transaction instance. From the abstract write-skew condition we may trivially ignore the results with an empty write-set. Hence, only result pairs with non-empty write-sets need to be checked.

We denote the second result of the `add(int)` method as T_{add}, and the second result of the `remove(int)` method as T_{rem}. To detect the possible existence of a *write-skew* we need to check the following pairs:

$$(\mathrm{T}_{add}, \mathrm{T}_{add}), (\mathrm{T}_{rem}, \mathrm{T}_{rem}), (\mathrm{T}_{add}, \mathrm{T}_{rem})$$

Let's examine in detail the pair $(\mathrm{T}_{add}, \mathrm{T}_{rem})$. We simplify the description of the read-set of each transaction by ignoring the field `value`, since neither transactions writes to that field and thus

```
32  @Atomic(state= "| this -> [head:h'] * List(h', nil)")
33  public void remove(int value) {
34    boolean result;
35    Node prev = head;
36    Node next = prev.getNext();
37    while (next.getValue() < value) {
38      prev = next;
39      next = prev.getNext();
40    }
41    if (next.getValue() == value) {
42      prev.setNext(next.getNext());
43      next.setNext(null);
44    }
45  }
```

Figure 4.6: Dummy write access in `remove(int)` method.

we will focus only on interactions with the field `next`. We assume that the shared variable `this` points to the same object in both transactions, otherwise conflicts would never arise. The read- and write-set for transactions T_{add}, and T_{rem} (relative to field *next*) are

$$\mathcal{R}_{add} = \{this.head,\ this.head.B,\ this.head.B.next\}$$
$$\mathcal{W}^{>}_{add} = \mathcal{W}^{<}_{add} = \{this.head.B.next\}$$
$$\mathcal{R}_{rem} = \{this.head,\ this.head.D,\ this.head.D.next,\ this.head.D.next.next\}$$
$$\mathcal{W}^{>}_{rem} = \mathcal{W}^{<}_{rem} = \{this.head.D.next\}$$

Given these read- and write-sets, if an instantiation of B and D exist that satisfies the *write-skew* condition then the concurrent execution of these two transactions may cause a *write-skew* anomaly. In this particular case, the assertion $B = D.next$, which means that the memory locations represented by B and by $D.next$ are the same, satisfies the *write-skew* condition. Hence, the concurrent execution of the `add(int)` method with the `remove(int)` method may generate a write-skew anomaly.

The *write-skew* anomaly can be corrected by making an additional write operation between lines 42 and 43 of the code (`next.setNext(null)`) shown in Figure 4.3. This change is illustrate in Figure 4.6. This write operation, although unnecessary in terms of the list semantics, is essential to make the list implementation safe under snapshot isolation as we shall see. Given the new list implementation from Figure 4.6, the result of the analysis by StarTM is depicted in Figure 4.7. Notice that the write-set has two *heap paths* describing that the transaction writes the *next* field of the penultimate and last memory locations. Now, the new read- and write-set for transactions T_{add}, and T_{rem} (relative to field *next*) are

$$\mathcal{R}_{add} = \{this.head,\ this.head.B,\ this.head.B.next\}$$
$$\mathcal{W}^{>}_{add} = \mathcal{W}^{<}_{add} = \{this.head.B.next\}$$
$$\mathcal{R}_{rem} = \{this.head,\ this.head.D,\ this.head.D.next, this.head.D.next.next\}$$
$$\mathcal{W}^{>}_{rem} = \mathcal{W}^{<}_{rem} = \{this.head.D.next,\ this.head.D.next.next\}$$

```
# Method boolean remove(int value)
Result 2:
ReadSet:   { this.head.(next)[*D].next.value,
             this.head.(next)[*D].next.next }
WriteSet>: { this.head.(next)[*D].next,
             this.head.(next)[*D].next.next }
WriteSet<: { this.head.(next)[*D].next,
             this.head.(next)[*D].next.next }
```

Figure 4.7: Sample of StarTM result output for corrected `remove(int)` method.

In this case, it is not possible to find an instantiation for B and D, such that the *write-skew* condition is true. Hence, these transactions can execute concurrently under snapshot isolation without ever triggering the *write-skew* anomaly.

4.5 Core Language

In this section we introduce the core language that will be used as the base language to define the static analysis algorithms. This language corresponds to a very simple object-oriented language that captures essential features of the Java programming language, such as object's field dereference, method invocation, and object creation.

4.5.1 Syntax

In this section we define the core language syntax. We include the subset of Java that captures essential features such as object creation (new), field dereferencing ($x.f$), assignment ($x := e$), and function invocation ($func(\vec{x})$). The syntax of the language is defined by the grammar in Figure 4.8. A program in this language corresponds to a set of functions definitions $func(\vec{x}) = S$ where $func$ is the function identifier, $\vec{x}$ is a shorthand for a list of function parameters $x_1, \ldots, x_n$, and S is the function body or statement.

Although this language will be used to verify properties of transactional memory programs, we do not need to explicitly represent transactions nor any parallel constructs. We assume that all functions can be executed as transactions, and for the purpose of the verification of static properties, all functions may execute concurrently with each others. Moreover, and as stated in Section 4.3, the verification of snapshot isolation anomalies can be done as if no interferences of other concurrent transactions occur.

The values of this language are composed by integers, memory locations (pointers), and the special value nil to represent null pointers. Boolean values may be encoded as integers. The expression $e \oplus_a e$ denotes any arithmetic binary operation such as addition, subtraction, or multiplication. The expression $e \oplus_b e$ denotes any boolean binary operation such as equality, or inequality.

In this language, objects are created using the new construct, and each object has a countable set of fields that can be accessed by two syntactic constructs: in the $x := y.f$ statement, variable x is assigned with the value associated with field f in the object pointed by variable y; in the $x.f := e$ statement, the value of expression e is associated with field f in the object pointed by variable x.

e	::=		(*expression*)
		x	(variables)
	|	n	(constant)
	|	$e \oplus_a e$	(arithmetic op)
	|	null	(null value)
b	::=		(*boolean exp*)
		$e \oplus_b e$	(boolean op)
	|	true	(true value)
	|	false	(false value)
A	::=		(*assignments*)
		$x := e$	(local)
	|	$x := y.f$	(heap read)
	|	$x := \mathit{func}(\vec{y})$	(function call)
	|	$x.f := e$	(heap write)
	|	$x :=$ new	(allocation)
S	::=		(*statements*)
		$S\,;S$	(sequence)
	|	A	(assignment)
	|	if b then S else S	(conditional)
	|	while b do S	(loop)
	|	return e	(return)
	|	skip	(Skip)
P	::=	$\big(\mathit{func}(\vec{x}) = S\big)^+$	(*program*)

Figure 4.8: Core language syntax.

Function definitions should end with the return e statement, and may be invoked in an assignment statement $x := \mathit{func}(\vec{x})$. The skip statement denotes a *no-op* operation.

In Figure 4.9 we show an example of a program that creates and manipulates an ordered linked list of integers, written using the syntax of the core language.

4.5.2 Operational Semantics

The operational semantics of the core language is defined by a reduction relation over configurations of the form $\langle s, h, S\rangle$, where $s \in$ Stacks is a stack (a mapping from variables to values), $h \in$ Heaps is a (concrete) heap (a mapping from locations to values through field labels), and S is a statement. We assume a countable set of program variables Vars (ranged over by $x, y, \ldots$).

$$\mathsf{Values} = \mathbb{Z} \cup \mathsf{Locations} \cup \{\mathsf{nil}\}$$

$$\mathsf{Stacks} = \mathsf{Vars} \to \mathsf{Values}$$

$$\mathsf{Heaps} = \mathsf{Locations} \rightharpoonup \mathsf{Fields} \to \mathsf{Values}$$

Each function is identified by a name $f \in$ Funcs, Funcs is a countable set of function names, and a map FuncMap : Funcs $\to$ Params $\times$ Stmt. Function FuncMap is used to retrieve the function body and respective parameters given the function identifier. We define a semantic function $\mathcal{A}$: Exp $\to$ Stacks $\to$ Values to evaluate expressions, where $\oplus_a$ represents the arithmetic binary

```
ll_create() = (
  list := new;
  pivot_min := new;
  pivot_max := new;
  pivot_min.value := -2147483648;
  pivot_min.next := pivot_max;
  pivot_max.value := 2147483647;
  pivot_max.next := null;
  list.head := pivot_min;
  return list
)
```

```
ll_add(list, value) = (
  node := new;
  node.value := value;
  node.next := null;
  prev := list.head;
  next := prev.next;
  while next.value < value do (
    prev := next;
    next := prev.next
  );
  if next.value != value then (
    node.next := next;
    prev.next := node;
    return true
  )
  else
    return false
)
```

```
ll_remove(list, value) = (
  node := new;
  node.value := value;
  node.next := null;
  prev := list.head;
  next := prev.next;
  while next.value < value do (
    prev := next;
    next := prev.next
  );
  if next.value != value then (
    prev.next := next.next;
    return true
  )
  else
    return false
)
```

Figure 4.9: Linked list example in the core language.

operations $+, -, \times, \ldots$

$$\mathcal{A}[\![e]\!]_s = \begin{cases} n, & \text{if } e = n \\ s(x), & \text{if } e = x \\ \mathsf{nil}, & \text{if } e = \mathsf{null} \\ \mathcal{A}[\![e_1]\!]_s \oplus_a \mathcal{A}[\![e_2]\!]_s, & \text{if } e = e_1 \oplus_a e_2 \end{cases}$$

Likewise, boolean expressions are evaluated according to the semantic function $\mathcal{B} : \mathsf{BExp} \to \{true, false\}$, where $\oplus_b$ represents the boolean binary operations $=, \neq, <, \leq, \ldots$

$$\mathcal{B}[\![b]\!]_s = \begin{cases} true, & \text{if } b = \mathsf{true} \\ false, & \text{if } b = \mathsf{false} \\ \mathcal{A}[\![e_1]\!]_s \oplus_b \mathcal{A}[\![e_2]\!]_s, & \text{if } b = e_1 \oplus_b e_2 \end{cases}$$

The small step structural operational semantics of the language is defined by the set of rules in Figure 4.10.

The structural operation rules define the behavior of the program over a stack and a heap.

$$\boxed{\langle s, h, S\rangle \Longrightarrow \langle s', h', S'\rangle}$$

$$\frac{\langle s, h, S_1\rangle \Longrightarrow \langle s', h', S_1'\rangle}{\langle s, h, S_1\ ; S_2\rangle \Longrightarrow \langle s', h', S_1'\ ; S_2\rangle}(\text{SEQ 1}) \qquad \frac{\langle s, h, S_1\rangle \Longrightarrow \langle s', h', \mathsf{skip}\rangle}{\langle s, h, S_1\ ; S_2\rangle \Longrightarrow \langle s', h', S_2\rangle}(\text{SEQ 2})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{true}}{\langle s, h, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Longrightarrow \langle s, h, S_1\rangle}(\text{COND 1})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{false}}{\langle s, h, \mathsf{if}\ e\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\rangle \Longrightarrow \langle s, h, S_2\rangle}(\text{COND 2})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{true}}{\langle s, h, \mathsf{while}\ e\ \mathsf{do}\ S\rangle \Longrightarrow \langle s, h, S\ ; \mathsf{while}\ e\ \mathsf{do}\ S\rangle}(\text{LOOP 1})$$

$$\frac{\mathcal{B}[\![e]\!]_s = \mathit{false}}{\langle s, h, \mathsf{while}\ e\ \mathsf{do}\ S\rangle \Longrightarrow \langle s, h, \mathsf{skip}\rangle}(\text{LOOP 2})$$

$$\frac{\mathcal{A}[\![e]\!]_s = v}{\langle s, h, \mathsf{return}\ e\rangle \Longrightarrow \langle s[\mathit{ret} \mapsto v], h, \mathsf{skip}\rangle}(\text{RETURN})$$

$$\frac{\mathcal{A}[\![e]\!]_s = v}{\langle s, h, x := e\rangle \Longrightarrow \langle s[x \mapsto v], h, \mathsf{skip}\rangle}(\text{ASSIGN})$$

$$\frac{s(y) = l \qquad l \in \mathsf{dom}(h) \qquad h(l)(f) = v}{\langle s, h, x := y.f\rangle \Longrightarrow \langle s[x \mapsto v], h, \mathsf{skip}\rangle}(\text{HEAP READ})$$

$$\frac{s(x) = l \qquad l \in \mathsf{dom}(h) \qquad \mathcal{A}[\![e]\!]_s = v}{\langle s, h, x.f := e\rangle \Longrightarrow \langle s, h(l)[f \mapsto v], \mathsf{skip}\rangle}(\text{HEAP WRITE})$$

$$\frac{\mathsf{FuncMap}(\mathit{func}) = (\vec{x}, S) \qquad S' = S\{\vec{y}/\vec{x}\}}{\langle s, h, x := \mathit{func}(\vec{y})\rangle \Longrightarrow \langle s, h, S'; x := \mathit{ret}\rangle}(\text{FCALL})$$

$$\frac{l \notin \mathsf{dom}(h)}{\langle s, h, x := \mathsf{new}\rangle \Longrightarrow \langle s[x \mapsto l], h[l \mapsto \_], \mathsf{skip}\rangle}(\text{ALLOCATION})$$

Figure 4.10: Structural operation semantics.

The ALLOCATION rule creates a new object by generating a fresh location l that was not part of the domain of the current heap. Both the HEAP READ and HEAP WRITE rules require that the object's location l exist in the domain of the heap, otherwise the program gets stuck, which may correspond to a runtime error such as dereferencing a null pointer. The ASSIGN rule only changes the stack, leaving the heap untouched. The FCALL rule denotes the invocation of function func

$$
\begin{array}{lcll}
e & ::= & & \textit{(expressions)} \\
 & & x, y, \ldots \in \mathsf{Vars} & \text{(program variables)} \\
 & | & x', y', \ldots \in \mathsf{Vars}' & \text{(existential variables)} \\
 & | & \mathsf{nil} & \text{(null value)} \\
\rho & ::= & f_1 : e,\ \ldots,\ f_n : e & \textit{(record)} \\
 & & & \\
S & ::= & e \mapsto [\rho] \ \mid\ p(\vec{e}) & \textit{(spatial predicates)} \\
P & ::= & e = e & \textit{(pure predicates)} \\
\Pi & ::= & \mathsf{true} \ \mid\ P \wedge \Pi & \textit{(pure part)} \\
\Sigma & ::= & \mathsf{emp} \ \mid\ S * \Sigma & \textit{(spatial part)} \\
 & & & \\
\mathcal{H} & ::= & \Pi | \Sigma & \textit{(symbolic heap)}
\end{array}
$$

Figure 4.11: Separation logic syntax.

where $(\vec{x}, S)$ corresponds to the parameter's list and function body respectively. The parameters $\vec{x}$ are capture-avoiding substituted by the arguments in $\vec{y}$ in statement S. After executing the substituted statement S', a special variable ret contains the function return value. The return of a function (RETURN rule) is defined as assigning to a special return variable ret the value of expression e;

The structural operation semantics rules defines the concrete semantics of the core language. In the following sections we will define the abstract semantics, or symbolic execution rules, for the same language to compute the approximations of read- and write-sets of each transaction.

4.6 Abstract States

We define an abstract state as the tuple $(\mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W})$, where $\mathcal{H}$ is a *symbolic heap* defined using a fragment of separation logic formulae, $\mathcal{M}$ is a map between variables and *heap path* expressions, and $\mathcal{R}$ and $\mathcal{W}$ are read- and write-sets respectively. The write-set $\mathcal{W}$ in our analysis is actually composed by two sets: a *may* write-set, denoted by $\mathcal{W}^{>}$, which over-approximates the concrete write-set, and a *must* write-set, denoted by $\mathcal{W}^{<}$, which under-approximates the concrete write-set.

The fragment of separation logic formulae that we use to describe symbolic heaps is defined by the grammar in Figure 4.11. Satisfaction of a formula $\mathcal{H}$ by a stack s and heap h is denoted $s, h \models \mathcal{H}$ and defined by structural induction on $\mathcal{H}$ (see Figure 4.12). There, as usual $[\![p]\!]$ is a component of the least fixed point of a monotone operator constructed from a inductive definition set; see [BBC08] for details. In this heap model a location maps to a record of values. The formula $e \mapsto [\rho]$ can mention any number of fields in ρ, and the values of the remaining fields are implicitly existentially quantified.

4.6.1 Symbolic Heaps

Symbolic heaps are abstract models of the heap of the form $\mathcal{H} = \Pi|\Sigma$, where Π is called the *pure part* and Σ the *spatial part*. We use prime variables $(x'_1, \ldots, x'_n)$ to implicitly denote existentially quantified variables that occur in $\Pi|\Sigma$. The pure part Π is a conjunction of pure predicates which states facts about the stack variables and existential variables (e.g., $x = \mathsf{nil}$). The spatial part Σ is the $*$ conjunction of spatial predicates, i.e., related to heap facts. In separation logic, the formula $S_1 * S_2$ holds in a heap that can be split into two disjoint parts, one of them described exclusively

$$
\begin{array}{rl}
s,h \models \mathsf{emp} & \text{iff } \mathsf{dom}(h) = \emptyset \\
s,h \models x \mapsto [f_1 : e_1,\ , f_n : e_n] & \text{iff } h = [s(x) \mapsto r] \text{ where } r(f_i) = s(e_i) \text{ for } i \in [1,n] \\
s,h \models p(\vec{e}) & \text{iff } (s(\vec{e}), h) \in [\![p]\!] \\
s,h \models \Sigma_0 * \Sigma_1 & \text{iff } \exists h_0, h_1.\ h = h_0 * h_1 \text{ and } s,h_0 \models \Sigma_0 \text{ and } s,h_1 \models \Sigma_1 \\
s,h \models e_1 = e_2 & \text{iff } s(e_1) = s(e_2) \\
s,h \models \Pi_1 \wedge \Pi_2 & \text{iff } s,h \models \Pi_1 \text{ and } s,h \models \Pi_2 \\
\\
s,h \models \Pi|\Sigma & \text{iff } \exists \vec{v'}.\ \big(s(\vec{x'} \mapsto \vec{v'}), h \models \Pi\big) \text{ and } \big(s(\vec{x'} \mapsto \vec{v'}), h \models \Sigma\big) \\
 & \quad \text{where } \vec{x'} \text{ is the collection of existential variables} \\
 & \quad \text{in } \Pi|\Sigma
\end{array}
$$

Figure 4.12: Separation Logic semantics.

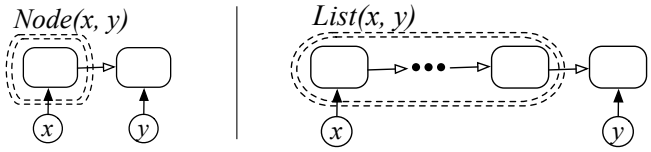


Figure 4.13: Graph representation of the $Node(x, y)$ and $List(x, y)$ predicates.

by S_1 and the other described exclusively by S_2.

In symbolic heaps, memory locations are either pointed directly by program variables (e.g., v) or existential variables (e.g., v'), or they are abstracted by predicates. Predicates are abstractions for the graph-like structure of a set of memory locations. For example, the predicate $Node(x, y)$, in Figure 4.13, abstracts a single memory location pointed by variable x, while the predicate $List(x, y)$ abstracts a set of an unbound number of memory locations, where each location is linked to another location of the set by the $next$ field.

A predicate $p(\vec{e})$ has at least one parameter, from its parameter set, that is the entry point for reaching every memory location that the predicate abstracts. We denote this kind of parameter as *entry* parameters. Also, there is a subset of parameters that correspond to the exit points of the memory region abstracted by the predicate. These parameters denote variables pointing to memory locations that are outside the predicate but the predicate has memory locations with links to these *outsider* locations. In Figure 4.13 we can observe that the predicate $List(x, y)$ has one *entry* parameter x and one *exit* parameter y.

We can infer the entry and exit parameters of a predicate by analyzing its body. The predicate body is composed by a disjunction of spatial formulas, which are composed by predicates, including the points-to ($\mapsto$) predicate. When defining a predicate, the name of the predicate may appear in its body, thus creating an inductive predicate definition. We denote $\mathsf{nonRec}(P)$ as the set of predicates with a different name from the one that is being defined, and $\mathsf{rec}(P)$ as the set of predicates with the same name as the one that is being defined. We define an inductive function $\delta^{+}_{P}(x)$ to assert if parameter x is an entry parameter of predicate P.

Definition 4.3 (Entry Parameter)**.** *Given a predicate P with a set of parameters $\vec{x}$, variable $x \in \vec{x}$ is an*

entry parameter of predicate P if:

$$\delta_P^+(x) \Leftrightarrow \begin{cases} \mathit{true} & \mathit{if}\ P = x \mapsto [\rho] \\ \exists p \in \mathsf{nonRec}(P) : \delta_p^+(\mathsf{param}(p, x)) & \mathit{otherwise} \end{cases}$$

We resort to an auxiliary function $\mathsf{param}(p, x)$, which returns the parameter name, that is associated with variable x within the body of predicate p. Dually we define an inductive function $\delta_P^-(x)$ to assert if parameter x is an exit parameter of predicate P.

Definition 4.4 (Exit Parameter)**.** *Given a predicate P with a set of parameters $\vec{x}$, variable $x \in \vec{x}$ is an exit parameter of predicate P if:*

$$\delta_P^-(x) \Leftrightarrow \begin{cases} \mathit{true} & \mathit{if}\ P = y \mapsto [\rho] \wedge x \in \mathrm{FV}(\rho) \\ \exists p \in \mathsf{nonRec}(P) : \delta_p^-(\mathsf{param}(p, x)) & \\ \quad \wedge\ \forall p \in \mathsf{nonRec}(P) \cup \mathsf{rec}(P) : \neg\, \delta_p^+(\mathsf{param}(p, x)) & \mathit{otherwise} \end{cases}$$

The free variables of a record $FV(\rho)$ are defined as $\{x_1, \ldots, x_n\}$ where $\rho = f_1 : x_1, \ldots, f_n : x_n$.

In summary a parameter variable x is an entry parameter if it occurs in the left side of a points-to predicate, or if it is used as an argument bind with an entry parameter of a predicate different from the one being defined. A parameter variable x is an exit parameter if it occurs in the right side of a points-to predicate, or if it is used as an argument bind with an exit parameter of a predicate, including the one being defined, and it must not be an entry parameter of any other predicate.

Using these functions one can create a directed graph, denoted as symbolic heap graph, based on the information present in a symbolic heap, where predicates correspond to nodes and variables correspond to edges with some restrictions. A variable z can only be an edge between two nodes, associated with predicates P_1 and P_2 if it is an exit parameter of predicate P_1 and an entry parameter of predicate P_2.

$$\mathsf{edge}(P_1(\vec{x}), z, P_2(\vec{y})) \Leftrightarrow z \in \vec{x} \wedge z \in \vec{y} \wedge \delta_{P_1}^-(\mathsf{param}(P_1, z)) \wedge \delta_{P_2}^+(\mathsf{param}(P_2, z))$$

An heap path expression may be computed by a transformation function over a single path of this graph.

4.6.2 Heap Paths

We are going to represent a memory location as a sequence of fields, starting from a program variable. If we successively dereference the field labels that appear in the sequence, we reach the memory location denoted by the sequence. We call these sequences of field labels, prefixed by a variable name, a *heap path*. For instance, the path $x.\mathit{left}.\mathit{right}$, denotes the memory location that is reachable by dereferencing the field *left* of the location pointed by variable x, and by dereferencing the field *right* of the location represented by $x.\mathit{left}$.

We can also represent sequences of field dereferences in a *heap path* by using the Kleene star ($*$) and choice ($|$) operators. For instance, the path $x.(\mathit{left} \mid \mathit{right})^*$ denotes a memory location that can be reached by starting on variable x and then dereferencing either the *left* or *right* field on each visited memory location.

$$
\begin{array}{lcll}
H & ::= & v \mid v.P & (\textit{heap path}) \\
P & ::= & f \mid f.P \mid C^*_A.P & (\textit{subpath}) \\
C & ::= & f \mid f \text{ “|” } C & (\text{choice})
\end{array}
$$

Figure 4.14: *Heap Path* syntax.

$$
\begin{array}{rll}
\mathcal{S}[\![v]\!]_{s,h,l} = \{l'\} & & \text{where } l' = s(v) \\
\mathcal{S}[\![v.P]\!]_{s,h,l} = \mathcal{S}[\![P]\!]_{s,h,l'} & & \text{where } l' = s(v) \\
\mathcal{S}[\![f]\!]_{s,h,l} = \{l'\} & & \text{where } l' = h(l,f) \\
\mathcal{S}[\![f.P]\!]_{s,h,l} = \mathcal{S}[\![P]\!]_{s,h,l'} & & \text{where } l' = h(l,f) \\
\mathcal{S}[\![C^*.P]\!]_{s,h,l} = \mathcal{S}[\![f_1.C^*.P]\!]_{s,h,l} \cup ... \cup \mathcal{S}[\![f_n.C^*.P]\!]_{s,h,l} \cup \mathcal{S}[\![P]\!]_{s,h,l} & & \text{where } C = f_1|...|f_n
\end{array}
$$

Figure 4.15: *Heap Path* semantics.

The syntax of *heap paths* is depicted in Figure 4.14 and corresponds to a very restrictive subset of the regular expressions syntax. A *heap path* always starts with a variable name (v) followed by sequences of field labels (f), repeating subpath expressions under a Kleene operator (C^*), and choices of field labels (C). We syntactically restrict *heap paths*, with respect to regular expressions, by only allowing choices of field labels guarded by a Kleene operator, and repetitions of choices of single field labels (not sequences). For instance, the path $x.(\mathit{left} \mid \mathit{right}^*)$ is not a valid *heap path* expression.

Each repeating subpath is always associated with a label. This is used to identify the subpath guarded by the star and we can rewrite $C^*_A.P$ as $A.P$ where $A = C^*$. As we shall see later, this label will be used to identify subpath expressions that denote the same concrete path in the heap. We may also denote the repetition sequence with a bar on top of the star, e.g., $x\,.C^{\bar{*}}_A$. This is used to distinguish between different interpretations, of *heap path* expressions contained in read- and write-sets.

We now define the semantics of *heap paths* with relation to concrete stacks and heaps through function $\mathcal{S}[\![H]\!]_{s,h,l}$ in Figure 4.15. According to this definition a *heap path* expression denotes the set of all memory locations that are reachable by following the path in a concrete memory, $\mathcal{S}[\![H]\!] \subseteq \mathsf{Locations}$. Abstract read- or write-sets are sets of *heap paths*. We write HPaths to denote the set of all *heap path* expressions.

4.6.3 Abstract Read- and Write-Sets

A heap path represents an abstract memory location, which might correspond to a set of concrete memory locations. A memory access, which can be a read or a write access, is represented as a pair composing a heap path and a field, which we will call as heap path access. We write a heap path access pair as a simple concatenation of the heap path with the field. For instance, a heap path access of the form $(v.P, f)$ may be simply denoted as $v.P.f$.

We represent the abstract read- and write-sets as sets of heap path accesses. This sets are constructed during the static analysis of transactions whenever a heap read or write access is analyzed.

Read-sets, may write-sets, and must write-sets are interpreted differently. For read-sets, we always consider the saturation of the read-set with the denotations of all prefixes of its heap

paths. For *must* write-sets, we consider one under-approximation where a heap-path H represents exactly one location in the set $\mathcal{S}[\![H]\!]$. For *may* write-sets, we consider the over-approximation by saturating the set with the expansion of the $\bar{*}$ repetition annotation. For instance, a heap path expression $x.C^{\bar{*}}.f$ in a may *write-set*, denotes write operations on all fields f for all locations of the set $\mathcal{S}[\![x.C^{\bar{*}}]\!]$.

4.6.4 From Symbolic Heaps to Heap Paths

During the static analysis procedure, we generate *heap paths* based on the information given by the symbolic heap. Recall that the only information given by the user to the verification tool is a description of the state at the beginning of the transaction using a symbolic heap, everything else is inferred.

Given a memory location l pointed by some variable x, if there is a path in the symbolic heap from some other variable s, where $s \in \mathsf{SVars}$, to variable x, then we can generate a *heap path* that represents the path from the shared variable s to the memory location l. Moreover, the computation of a *heap path* from the symbolic heap requires a transformation function (Γ) that given a predicate and its arguments returns a *heap path*.

We can use the symbolic heap graph defined in Section 4.6.1, where each node corresponds to a predicate, and each edge corresponds to a link between each predicate through a variable. We can compute a heap path from one shared variable to another variable by concatenating the heap paths computed for each node presented in a path in this symbolic heap graph.

Given a sequence of edges that corresponds to the path between a shared variable $s \in \mathsf{SVars}$ and a program variable $x \in \mathsf{Vars}$

$$(P_1, x_1, P_2), (P_2, x_2, P_3), \ldots, (P_n, x_n, P_{n+1})$$

where s in an entry parameter of P_1 and will be denoted as x_0, and variable $x = x_n$. The heap path is computed by the concatenation of the *sub*-heap paths constructed from the definitions of the predicates that are present in the sequence of edges, using the function Γ:

$$\bigodot_{i=1}^{n} \Gamma(P_i, x_{i-1}, x_i)$$

The big operator $\odot$ corresponds to the lift of a single concatenation operation $x.P \odot z.P'$ to a set of heap paths. The concatenation operation $x.P \odot z.P'$ concatenates the path described by P' to the *heap path* $x.P$ resulting in the heap path $x.P.P'$. Note that this concatenation is sound, given the pre-condition that $x.P$ represents the same memory location as variable z.

For each predicate P_i we construct a heap path from its entry parameter x_{i-1} to its exit parameter x_i using the function $\Gamma(P_i, x_{i-1}, x_i)$. Function Γ operates over the predicate definition, as also the δ^+ and δ^- functions.

The definition of function Γ for the points-to predicate is the following:

$$\Gamma(x \mapsto [\ldots, f : y, \ldots], x, y) = x.f$$

To assist the definition of function Γ for inductive predicates, we first define the structure of a predicate definition. As we previously stated, an inductive predicate is a disjunction of spatial

separated predicates:

$$P(\vec{x}) \Leftrightarrow p * \ldots * p' \mid \ldots \mid p'' * \ldots * p''' * r$$

Since we are describing an inductive predicate, some disjunctive branches may contain a recursive reference to the predicate P being defined, which are denoted as r. The key idea to define function Γ is that we separate each disjunctive branch, and for each branch we create a symbolic heap graph and generate a heap path expression, as described previously using the Γ function recursively.

Disjunctive branches are dealt differently depending on whether they include or not recursive references. Consider the following disjunctive branch without a recursive reference:

$$\underbrace{p_1 * \ldots * p_n}_{\phi}$$

We denote the heap path expression of the disjunctive branch as ϕ. In the case of a disjunctive branch with a recursive reference we compute the heap path expression by only considering the non-recursive references:

$$\underbrace{p_1 * \ldots * p_n}_{\phi^r} * r$$

In this case the heap path expression is denoted as ϕ^r. By processing each disjunctive branch, we get a set of heap path expressions:

$$\{\phi_1, \ldots, \phi_n, \phi_1^r, \ldots, \phi_n^r\}$$

The final heap path expression corresponds to a special composition of all "sub" heap path expressions:

$$\phi_1 \mid \ldots \mid \phi_n \mid \phi_1^r \odot (\phi_1^r \mid \ldots \mid \phi_n^r)^*_{A_1} \odot (\phi_1 \mid \ldots \mid \phi_n) \mid \ldots \mid \phi_n^r \odot (\phi_1^r \mid \ldots \mid \phi_n^r)^*_{A_n} \odot (\phi_1 \mid \ldots \mid \phi_n)$$

where $x_1.P_1 \odot (x_2.P_2 \mid \ldots \mid x_n.P_n)^* = x_1.P_1.(P_2 \mid \ldots \mid P_n)^*$. Labels $A_1, \ldots, A_n$ are fresh in the context of the symbolic state where the *heap path* is computed. Notice that *heap path* expressions containing repetitions and choices are only generated when transforming inductive predicates into *heap paths*. Although the composition originates a rather complex expression, most of the times this expression can be simplified as we will see in the following examples.

Consider the examples of a *heap path* generated for a list segment and a tree segment predicates:

Example 4.1 (Heap Path of the List Segment Predicate)**.**

$$lseg(x, y) \Leftrightarrow x \mapsto [next : y] \vee \exists z'.\ x \mapsto [next : z'] * lseg(z', y)$$

Given the lseg predicate definition, the set of disjunctive branches with the respective "sub" heap path is:

$$\underbrace{x \mapsto [next : y]}_{\phi = x.next} \quad , \quad \underbrace{x \mapsto [next : z']}_{\phi^r = x.next} * lseg(z', y)$$

Thus, the final heap path expression is composed as:

$$\phi \mid \phi^r \odot \phi^{r*} \odot \phi = x.next \mid x.next \odot (x.next)^*_A \odot x.next = x.next \mid x.next\ .next^*_A.next$$

This expression can be further simplified as:

$$x.next \mid x.next\ .next^*_A.next = x.next \mid x.next\ .next^+_A = x.next^+_A$$

We abbreviate repeating sequences with at least one field label using symbol $+$ (e.g. $next^+$). The final result for the heap path *that represents the memory location pointed by y and reachable from x is:*

$$\Gamma(lseg(x,y),x,y) = x\ .\ next^+_A$$

Example 4.2 (Heap Path of the Tree Segment Predicate)**.**

$$tnd(x,l,r) \Leftrightarrow x \mapsto [left : l, right : r]$$

$$tree(x) \Leftrightarrow \exists l', r'.\ \ tnd(x,l',r') * tree(l') * tree(r')$$

$$\begin{aligned} tseg(x,y) \Leftrightarrow\ & \exists z'.\ \big(\ tnd(x,y,z') \vee tnd(x,z',y)\big) * tree(y) * tree(z') \\ & \vee \exists z', w'.\ \big(\ tnd(x,z',w') \vee tnd(x,w',z')\big) * tseg(z',y) * tree(w') \end{aligned}$$

Given the $tseg$ predicate definition, the set of disjunctive branches with the respective "sub" heap path is:

$$\underbrace{tnd(x,y,z')}_{\phi_1 = x.left} * tree(y) * tree(z')$$

$$\underbrace{tnd(x,z',y)}_{\phi_2 = x.right} * tree(y) * tree(z')$$

$$\underbrace{tnd(x,z',w')}_{\phi^r_1 = x.left} * tseg(z',y) * tree(w')$$

$$\underbrace{tnd(x,w',z')}_{\phi^r_2 = x.right} * tseg(z',y) * tree(w')$$

Thus, the final heap path expression is composed as:

$$\begin{aligned} & \phi_1 \mid \phi_2 \mid \phi^r_1 \odot (\phi^r_1 \mid \phi^r_2)^*_A \odot (\phi_1 \mid \phi_2) \mid \phi^r_2 \odot (\phi^r_1 \mid \phi^r_2)^*_A \odot (\phi_1 \mid \phi_2) \\ = {} & x.left \mid x.right \mid x.left \odot (x.left \mid x.right)^*_A \odot (x.left \mid x.right) \\ & \qquad \mid x.right \odot (x.left \mid x.right)^*_A \odot (x.left \mid x.right) \\ = {} & x.left \mid x.right \mid x.left\ .(left \mid right)^*_A.(left \mid right) \\ & \qquad \mid x.right\ .(left \mid right)^*_A.(left \mid right) \\ = {} & x.left \mid x.right \mid x.left\ .(left \mid right)^+_A \mid x.right\ .(left \mid right)^+_A \\ = {} & x.(left \mid right)^+_A \end{aligned}$$

4.7 Abstract Semantics

Next, we define the abstract semantics, or symbolic execution rules, for the core language presented in Section 4.5 taking inspiration from [DOY06]. In our case, the abstract semantics defines the effect of statements on abstract states composed by a symbolic heap, a path map, and a read- and write-set. We represent an abstract state as: $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W} \rangle \in (\mathsf{SHeaps} \times (\mathsf{Vars} \rightharpoonup \mathsf{HPaths}) \times \mathsf{Rs} \times \mathsf{Ws})$ where SHeaps is the set of all symbolic heaps, $(\mathsf{Vars} \rightharpoonup \mathsf{HPaths})$ is the map between program variables and *heap path* expressions, Rs is the set of all read-sets, and Ws is the set of pairs of all *may* and *must* write-sets. We write SStates for denoting the set of all abstract states.

The path map $\mathcal{M}$ is a map that associates variables to *heap path* expressions. In each step of the symbolic execution, a variable x in this map is associated with a *heap path* expression that represents the memory location pointed by x. The purpose of this map is to keep a *heap path* expression less abstract than the one that we can capture from the symbolic heap. For instance, in the map, we may have the information that we only accessed the *left* field of each node of a tree, but from the symbolic heap we get the information that we accessed the *left* or *right* fields in each node. The symbolic execution will always maintain the invariant $S_p \subseteq G_p$ where S_p is the *heap path* in the path map and G_p is the *heap path* from the symbolic heap, for a variable x. The subset relation means that all paths described by S_p are described by G_p, and thus S_p is more precise than G_p.

Each transactional method is annotated with the `@Atomic` annotation describing the initial symbolic heaps for that transaction. The symbolic execution will analyze only transactional methods and all methods present in the invocation tree that occurs inside their body. In the beginning of the analysis we have the specification of the symbolic heaps for each transactional method. An empty path map and empty read- and write-sets are associated to each initial symbolic heap, thus creating a set of initial abstract states for each transactional method. The complete information for each method is composed by:

- the initial abstract states, which can be given by the programmer or be computed by the analysis;
- the final abstract states resulting from the method's execution. These final abstract states are computed by the analysis and, in the special case of the transactional methods, are the final result of the analysis.

For each method, given one initial abstract state, the analysis may produce more than one abstract states. The abstract semantics is defined by a function exec that yields a set of abstract states or an error ($\top$), given a method body (from Stmt) and an initial abstract state (from SStates):

$$\mathsf{exec} : \mathsf{Stmt} \times \mathsf{SStates} \to \mathcal{P}(\mathsf{SStates}) \cup \{\top\}$$

To support inter-procedural analysis we also need the auxiliary function spec, that given a method signature ($\mathit{func}(\vec{x}) \in \mathsf{Sig}$), yields a mapping from symbolic heaps to sets of abstract states: $\mathsf{SHeaps} \to \mathcal{P}(\mathsf{SStates})$.

$$\mathsf{spec} : \mathsf{Sig} \to (\mathsf{SHeaps} \to \mathcal{P}(\mathsf{SStates}))$$

For non-transactional methods, called inside transactions, the initial abstract state is computed

in the course of the symbolic execution, which is inferred from the abstract state of the calling context. Recursive functions are currently not supported by our analysis technique.

4.7.1 Past Symbolic Heap

Our analysis require a special kind of predicates, which we call *past predicates*, denoted as $\widehat{p}(\vec{e})$ or $x \widehat{\mapsto} [\rho]$. The past symbolic heap is composed by predicates and past predicates. The latter ones have an important role in the correctness for computing *heap paths*. *Heap paths* must always be computed with respect to the initial snapshot of memory, which is shared between transactions, and corresponds to the initial symbolic heap. Otherwise we may fail to detect some shared memory access due to some memory privatization pattern. We illustrate this problem by means of an example:

Example 4.3. *Given an initial symbolic heap, where* $x \in \mathsf{SVars}$ *is a shared variable:*

$$\{\}|\ List(x, y) * y \mapsto [next : z] * z \mapsto \mathsf{nil}$$

The heap paths *representing the locations pointed by each variable are:*

$$x \equiv x \qquad y \equiv x.(next)^{+}_{A} \qquad z \equiv x.(next)^{+}_{A}.next$$

If we update the location pointed by y *by assigning its* $next$ *field to* nil *we get*

$$\{\}|\ List(x, y) * y \mapsto [next : \mathsf{nil}] * z \mapsto \mathsf{nil}$$

After the update, the heap paths *representing the locations pointed by* x *and* y *remain the same. However,* z *is no longer reachable from a shared variable, and hence, we have lost the information that in the context of a transaction,* z *is still a shared memory location subject to concurrent modifications.*

This example shows that the *heap path* representing a memory location, that is reachable by a shared variable in the beginning of the transaction, must not be changed by the updates in the structure of the heap. So, in order to compute the correct *heap path* we need to use a "past view" of the current symbolic heap. To get the past view we need *past* predicates, which are added to the symbolic heap whenever an update is made to the structure of the heap. In the case of the previous example, the result of updating variable y would give the following symbolic heap:

$$\{\}|\ List(x, y) * y \mapsto [next : \mathsf{nil}] * y \widehat{\mapsto} [next : z] * z \mapsto \mathsf{nil}$$

The *past* predicate $y \widehat{\mapsto} [next : z]$ denotes that there was a *link* between variable y and z in the initial symbolic heap. Now, if there is a read access to a field of the memory location pointed by variable z, we compute the *heap path* of this location in the past view of the symbolic heap. We define a function that given a symbolic heap returns the past view of such symbolic heap:

Definition 4.5 (Past Symbolic Heap)**.** *Let* $\mathsf{Past}(H)$ *be the set of past predicates in* H*, and* $\mathsf{NPast}(\Pi|\Sigma) = \{S \mid \Sigma = S * \Sigma' \wedge \neg\, \mathsf{hasPast}_{\Pi|\Sigma}(S)\}$*. Then we define the past symbolic heap by*

$$\mathsf{PastSH}(\Pi|\Sigma) \triangleq \Pi|\ \circledast_{S \in \mathsf{NPast}(\Pi|\Sigma)} S * \circledast_{\widehat{S} \in \mathsf{Past}(\Pi|\Sigma)} \widehat{S}$$

This function makes use of the hasPast function to assert if there is already a *past predicate*, in the symbolic heap, with the same *entry* parameters. We define hasPast as:

Definition 4.6 (Has Past)**.**

$$\begin{array}{lcl} \mathsf{hasPast}_{\mathcal{H}}(x \mapsto [\rho]) & \Leftrightarrow & \mathcal{H} \vdash x \mathrel{\widehat{\mapsto}} [\rho] * true \\ \mathsf{hasPast}_{\mathcal{H}}(p(\vec{i}, \vec{o})) & \Leftrightarrow & \forall i \in \vec{i} : \delta_p^+(i) \wedge \exists i \in \vec{i} : \mathcal{H} \vdash \widehat{p}(\ldots, i, \ldots) * true \end{array}$$

The result of the past heap function applied to the previous example is:

$$\begin{array}{c} \mathsf{PastSH}(\{\} | \; List(x, y) * y \mapsto [next : \mathsf{nil}] * y \mathrel{\widehat{\mapsto}} [next : z] * z \mapsto \mathsf{nil}) \\ \triangleq \{\} | \; List(x, y) * y \mapsto [next : z] * z \mapsto \mathsf{nil} \end{array}$$

Which corresponds to the initial symbolic heap of Example 4.3. Thus we can calculate correctly the *heap paths* of the locations pointed by x, y and z.

We also define a function $\mathsf{genPast}_{\mathcal{H}}(x \mapsto [\rho])$ that if the symbolic heap $\mathcal{H}$ does not contain a past points-to predicate for a points-to predicate $x \mapsto [\rho]$, it creates a new past predicate $x \mathrel{\widehat{\mapsto}} [\rho]$.

Definition 4.7 (Generate Past Predicate)**.**

$$\mathsf{genPast}_{\mathcal{H}}(x \mapsto [\rho]) \triangleq \begin{cases} \mathsf{emp} & \textit{if } \mathsf{hasPast}_{\mathcal{H}}(x \mapsto [\rho]) \\ x \mathrel{\widehat{\mapsto}} [\rho] & \textit{otherwise} \end{cases}$$

4.7.2 Symbolic Execution Rules

The symbolic execution is defined by the rules shown in Figure 4.16. For the sake of simplicity, these rules are defined over a single symbolic heap, although we can easily lift to a set of symbolic heaps. The abstract semantics of conditional statements is the union of the resulting symbolic heaps of each branch. The resulting symbolic heap of a loop statement, which corresponds to the loop invariant, is computed using a fix-point computation over the abstract semantics rules defined in Figure 4.16.

The rule ASSIGN, when executed in a state $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W} \rangle$ adds the information that in the resulting state, x is equal to e. As in standard Hoare/Floyd style assignment, all the occurrences of x, in $\mathcal{H}$ and e, are replaced by a fresh existential quantified variable x'. We also compute a new path map where we associate variable x with the *heap path* of expression e. If e is null then we associate variable x with empty ϵ. The read- and write-set are not changed because there are no changes in the heap.

The HEAP READ rule adds an equality, to the resulting state, between x and the content of the field f of the location pointed by y. Every time we access the heap, for reading or writing, we compute a new path map. In this case we generate a *heap path* for variable y using the symbolic heap and the current path map. Note that the *heap path* generated is computed in the past symbolic heap as described in Section 4.7.1. This operation, denoted as genPath, is also responsible for abstracting the representation of *heap paths*, as we will describe it in detail in Section 4.7.4. Given the new computed *heap path* p, we compute a new path map by associating path p with variable y and to all its aliases. We use the function updateMap to perform these operations. Then we associate variable x with the result of the concatenation of path p with field f, where p represents the

$$\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, S\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R}', \mathcal{W}'\rangle \quad \vee \quad \langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, S\rangle \Longrightarrow \top$$
$$I(e) ::= e.f := x \mid x := e.f$$

$$\frac{\mathcal{H} \vdash y = \mathsf{nil}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, I(y)\rangle \Longrightarrow \top}(\textsc{Heap Error})$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := e\rangle \Longrightarrow \langle x = e[x'/x] \wedge \mathcal{H}[x'/x], \mathcal{M}[x \mapsto \mathcal{M}(e)], \mathcal{R}, \mathcal{W}\rangle}(\textsc{Assign})$$

$$\frac{p = \mathsf{genPath}(\mathsf{PastSH}(\mathcal{H}), \mathcal{M}, y) \qquad \mathcal{M}' = \mathsf{updateMap}(\mathcal{M}, \mathcal{H}, y, p)[x \mapsto p.f] \\ \mathcal{H}' = x = z[x'/x] \wedge \mathcal{H}[x'/x] \qquad x' \text{ is fresh}}{\langle \mathcal{H} * y \mapsto [f : z], \mathcal{M}, \mathcal{R}, \mathcal{W}, x := y.f\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R} \cup \{p.f\}, \mathcal{W}\rangle}(\textsc{Heap Read})$$

$$\frac{p = \mathsf{genPath}(\mathsf{PastSH}(\mathcal{H} * x \mapsto [f : z]), \mathcal{M}, x) \qquad \mathcal{M}' = \mathsf{updateMap}(\mathcal{M}, \mathcal{H}, x, p) \\ \mathcal{H}' = \mathcal{H} * x \mapsto [f : e] * \mathsf{genPast}_{\mathcal{H}}(x \mapsto [f : z])}{\langle \mathcal{H} * x \mapsto [f : z], \mathcal{M}, \mathcal{R}, \mathcal{W}, x.f := e\rangle \Longrightarrow \langle \mathcal{H}', \mathcal{M}', \mathcal{R}, \mathcal{W} \uplus \{p.f\}\rangle}(\textsc{Heap Write})$$

$$\frac{x' \text{ is fresh}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := \mathsf{new}\rangle \Longrightarrow \langle \mathcal{H}[x'/x] * x \mapsto [], \mathcal{M}[x \mapsto \epsilon], \mathcal{R}, \mathcal{W}\rangle}(\textsc{Allocation})$$

$$\frac{}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, \mathsf{return}\, e\rangle \Longrightarrow \langle ret = e \wedge \mathcal{H}, \mathcal{M}[ret \mapsto \mathcal{M}(e)], \mathcal{R}, \mathcal{W}\rangle}\ (\textsc{Return})$$

$$\frac{\mathcal{H} \vdash \mathcal{H}'[\vec{y}/\vec{z}] * Q \quad \langle \mathcal{H}'', \mathcal{M}', \mathcal{R}', \mathcal{W}'\rangle \in \mathsf{spec}(func(\vec{z}))(\mathcal{H}') \quad \mathcal{H}''' = Q * \mathcal{H}''[\vec{y}/\vec{z}] \\ \mathcal{R}'' = \mathcal{R}'[\vec{y}/\vec{z}] \qquad \mathcal{W}'' = \mathcal{W}'[\vec{y}/\vec{z}] \qquad \mathcal{M}'' = \mathsf{updateAllMap}(\mathcal{R}'' \cup \mathcal{W}'', \mathcal{M}, \mathcal{H}''') \\ r.P' = \mathcal{M}'(ret) \qquad \mathcal{M}''' = \mathcal{M}''[x \mapsto \mathsf{genPath}(\mathsf{PastSH}(\mathcal{H}'''), \mathcal{M}'', r).P'] \\ \mathcal{R}''' = \mathcal{R} \cup \{\mathcal{M}'''(v).P \mid v.P \in \mathcal{R}''\} \qquad \mathcal{W}''' = \mathcal{W} \uplus \{\mathcal{M}'''(v).P \mid v.P \in \mathcal{W}''\}}{\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W}, x := func(\vec{y})\rangle \Longrightarrow \langle x = ret \wedge \mathcal{H}''', \mathcal{M}''', \mathcal{R}''', \mathcal{W}'''\rangle}(\textsc{FCall})$$

$$\mathsf{alias}_{\mathcal{H}}(x) \triangleq \{y \mid \mathcal{H} \vdash x = y\} \cup \{x\}$$

$$\mathsf{updateMap}(\mathcal{M}, \mathcal{H}, x, p) \triangleq \{v \mapsto s \mid v \mapsto s \in \mathcal{M} \wedge v \notin \mathsf{alias}_{\mathcal{H}}(x)\} \cup \{a \mapsto p \mid a \in \mathsf{alias}_{\mathcal{H}}(x)\}$$

$$\mathsf{updateAllMap}(V, \mathcal{M}, \mathcal{H}) \triangleq \\ \{s \mid v.P \in V \wedge p = \mathsf{genPath}(\mathsf{PastSH}(\mathcal{H}), \mathcal{M}, v) \ \wedge \ s \in \mathsf{updateMap}(\mathcal{M}, \mathcal{H}, v, p)\}$$

Figure 4.16: Operational Symbolic Execution Rules.

memory location pointed to by y. Finally, we add to the read-set the memory access represented by the *heap path* p and the field f.

The HEAP WRITE rule denotes an update to the value of field f in the location pointed by x. Variable x is associated with the generated *heap path* p ($\mathsf{updateMap}(\mathcal{M}, \mathcal{H}, x, p)$) in a new path map. The symbolic heap is extended with a past predicate representing the link between variable x and the record $[f : z]$ that just ceased to exist. The resulting write-set is extended with the field access $\{p.f\}$ ($\mathcal{W} \Cup \{p.f\}$). The operation $\mathcal{W} \Cup \{p.f\}$, denotes the adding of $\{p.f\}$ to both components of the write set $\mathcal{W}$, to the *may* write-set $\mathcal{W}^{>}$ and to the *must* write-set $\mathcal{W}^{<}$. While adding an *heap path* access $p.f$ to the *must* write-set $\mathcal{W}^{<}$ is straightforward, adding $p.f$ to the *may* write-set $\mathcal{W}^{>}$ is a bit more involved. If $\mathcal{W}^{>}$ already contains $p.f$, then we replace all repeating sequences in p, by repeating sequences of the kind $\bar{*}$. For instance, in the previous example, if $p.f = x\,.\,next^{*}_{A}\,.\,next$ is already in $\mathcal{W}^{>}$, the may write-set after adding $p.f$ contains $x\,.\,next^{\bar{*}}_{A}\,.\,next$ instead. With this operation we are conservatively over-approximating the write-set by saying that the transaction writes on all locations denoted by path p.

When a new memory location is allocated, rule ALLOCATION, and is assigned to variable x we update the path map entry for variable x with *empty* (ϵ).

In the FCALL rule, the function spec is used to get the abstract state $\langle \mathcal{H}'', \mathcal{M}', \mathcal{R}', \mathcal{W}' \rangle$ which corresponds to one of the final states of the symbolic execution of a function $func$. The read- and write-set are composed by *heap path* expressions, where each expression $v.P$ represents a memory location where variable v is the root of the path. This variable is a root variable in the context of function $func$ but in the context of the function that is being analyzed where $func$ was invoked, variable v might point to a memory location that is represented by a *heap path* expression $v'.P'$ where $v' \neq v$. This means that a memory location that is represented by the expression $v.P$ in the context of $func$, is represented by the expression $v'.P'.P$ in the context of the calling site of $func$ where $v'.P'$ is the expression that represent the memory location pointed by v in the context of the calling site. We need to update all *heap path* expressions of all variables that are in the returned read- ($\mathcal{R}'$) and write-set ($\mathcal{W}'$). We use the $\mathsf{updateAllMap}$ function to iterate over all variables and generate a new *heap path* expression and update the path map accordingly. The return value of function $func$ is assigned to variable x and therefore we update the path map entry for variable x with the *heap path* expression that represents memory location pointed by the special return variable ret in the context of the calling site. In the last step, we merge the read- and write-sets using the updated path map $\mathcal{M}'''$ by concatenating the *heap path* $\mathcal{M}'''(v)$ with the remaining path returned from the read- ($\mathcal{R}'$) or write-set ($\mathcal{W}'$). The final symbolic heap $\mathcal{H}'''$ is computed in the typical way for inter-procedural analysis using separation logic that is by combining the frame of the function call (in this case Q)[3], and the postcondition of the spec $\mathcal{H}''$ [DPJ08].

Since we are not aiming at verifying execution errors, we silently ignore the symbolic error states ($\top$) produced by HEAP ERROR rule in our analysis.

4.7.3 Rearrangement Rules

The symbolic execution rules manipulate object's fields. When these are hidden inside abstract predicates both HEAP READ and HEAP WRITE rules require the analyzer to expose the fields they are operating on. This is done by the function rearr defined as:

[3]The frame of a call is the part of the calling heap which is not related with the precondition of the callee.

Definition 4.8 (Rearrangement)**.**

$$\mathsf{rearr}(\mathcal{H}, x.f) \triangleq \{\mathcal{H}' * x \mapsto [f : y] \mid \mathcal{H} \vdash \mathcal{H}' * x \mapsto [f : y]\}$$

4.7.4 Fixed Point Computation and Abstraction

Following the spirit of abstract interpretation [CC77] and the jStar work [DPJ08] to ensure termination of symbolic execution, and to automatically compute loop invariants, we apply abstraction on sets of abstract states. Typically, in separation logic based program analyses, abstraction is done by rewriting rules, also called abstraction rules which implement the function $\mathsf{abs} : \mathsf{SHeaps} \rightarrow \mathsf{SHeaps}$. For each analyzed statement we apply abstraction after applying the execution rules. The abstraction rules accepted by StarTM have the form:

$$\frac{premises}{\mathcal{H} \vdash \mathsf{emp} \rightsquigarrow \mathcal{H}' \vdash \mathsf{emp}} (\textsc{Abstraction Rule})$$

This rewrite is sound if the symbolic heap $\mathcal{H}$ implies the symbolic heap $\mathcal{H}'$. An example of some abstraction rules, for the $List(x, y)$ predicate, is shown in Figure 4.4. Each rule is only triggered when the premises are satisfied in the current symbolic heap. Past predicates are also abstracted in order to ensure the convergence of the analysis.

The *heap path* expressions that are stored in the path map ($\mathcal{M}$) need also to be abstracted because otherwise we would get expressions with infinite sequences of fields. Since the symbolic heap is abstracted we can use it to compute an abstract *heap path* expression. The abstraction procedure is done by the $\mathsf{genPath}(\mathcal{H}, \mathcal{M}, v)$ function. This function receives a symbolic heap $\mathcal{H}$, a path map $\mathcal{M}$, and a variable v for which will be computed the *heap path* representing the memory location pointed by such variable.

The *heap path* stored in the path map $\mathcal{M}$ for variable v will be denoted as S, and the *heap path* computed from the symbolic heap will be denoted as G. The analysis will always ensure the invariant $S \subseteq G$. This subset relation means that all paths described by S are also described by G.

The result of this function is a *heap path*, denoted as E which satisfies the following invariant: $S \subseteq E \subseteq G$. Since the symbolic heap is proven to converge into a fixed point, the *heap path* E will also converge into a fixed point because it is a subset of G.

The procedure to compute the path E is based on a pattern matching approach. Taking G as the most abstract path we generate a pattern from it that must match in S. This pattern is generated by taking G and substituting all its repeating sequences with wildcards. For instance, if $G = x.(\mathit{left} \mid \mathit{right})^+_A.\mathit{right}$ then the pattern would be $Pt = x.\alpha.\mathit{right}$ where α is a wildcard. We also denote α_G as the subpath in G that is associated to the wildcard α, and in this case, $\alpha_G = (\mathit{left} \mid \mathit{right})^+_A$.

We take this pattern and try to apply it to S and check which subpath expression of S matches the wildcard. For instance, if $S = x.\mathit{left}.\mathit{left}.\mathit{right}$, then the wildcard α of pattern $Pt = x.\alpha.\mathit{right}$ will match $\mathit{left}.\mathit{left}$ denoted as α_S. The pattern can only be matched successfully if the wildcard in S (α_S) and the wildcard in G (α_G) satisfy the following invariant: $\alpha_S \subseteq \alpha_G$, which is the case in our example.

Now we apply an abstraction operation over the wildcard to generate a more abstract subpath. We denote this operation as compress and is defined in Figure 4.17. The result of applying the abstraction function to wildcard α_S is $\mathsf{compress}(\alpha_S) = \mathit{left}^+_B$. Notice that the abstracted subpath

$$
\begin{array}{rll}
\mathsf{compress}(f_1.f_2) = (f_1)^+_A & \text{if } f_1 = f_2 & \text{where } A \text{ is fresh} \\
\mathsf{compress}(f_1.f_2) = (f_1|f_2)^+_A & \text{if } f_1 \neq f_2 & \text{where } A \text{ is fresh} \\
\mathsf{compress}((\mathcal{C})^+_C.f_1) = (\mathcal{C})^+_C & \text{if } f_1 \in \mathcal{C} & \\
\mathsf{compress}((\mathcal{C})^+_C.f_1) = (\mathcal{C}|f_1)^+_C & \text{if } f_1 \notin \mathcal{C} & \\
\mathsf{compress}(f_1.f_2.P) = \mathsf{compress}(\mathsf{compress}(f_1.f_2).P) & &
\end{array}
$$

Figure 4.17: Compress abstraction function.

satisfies the invariant $\alpha_S \subseteq \mathsf{compress}(\alpha_S) \subseteq \alpha_G$. Finally, we substitute the wildcards in the pattern for the computed abstract subpath expressions. In our example we get the final expression $E = x.\,left^+_B\,.right$ which is a subset of G.

4.7.5 Write-Skew Detection

The result of the symbolic execution is a set of symbolic states $\langle \mathcal{H}, \mathcal{M}, \mathcal{R}, \mathcal{W} \rangle$ for each transactional method. In this section, we define the *write-skew* test, which is based in the abstract read- and write-set $(\mathcal{R}, \mathcal{W})$ and in the satisfiability of the condition of Definition 4.2 (see example in Figure 4.5).

Recall that the interpretation of read-sets contain all prefixes of its heap paths. Hence, to compute the satisfiability of the *write-skew condition* we must compute the set of prefixes of the heap-paths in both read-sets. We define $\mathsf{prefix}(x.P)$ for a heap path expression $x.P$ as follows:

$$
\begin{array}{ll}
\mathsf{prefix}(P.f) \triangleq \{P.f\} \cup \mathsf{prefix}(P) & \mathsf{prefix}(P.C^*_A) \triangleq \{P.C^*_A\} \cup \mathsf{prefix}(P) \\
\mathsf{prefix}(x.f) \triangleq \{x.f\} & \mathsf{prefix}(x.C^*_A) \triangleq \{x.C^*_A\}
\end{array}
$$

and we lift the prefix definition for sets of heap paths $\mathsf{prefix}(\mathcal{R})$ as

$$\mathsf{prefix}(\mathcal{R}) \triangleq \bigcup_{p \in \mathcal{R}} \mathsf{prefix}(p).$$

For instance, the prefixes of the read-set $\mathcal{R} = \{this.head.(next)^*_A.next\}$ are:

$$\mathsf{prefix}(\mathcal{R}) = \{this.head,\ this.head.A,\ this.head.A.next\}$$

For the sake of simplicity, we denote repeating sequences by their unique label. Given the sets, $\mathcal{R}^\star_1 = \mathsf{prefix}(\mathcal{R}_1)$, $\mathcal{R}^\star_2 = \mathsf{prefix}(\mathcal{R}_2)$, $\mathcal{W}^<_1$, $\mathcal{W}^>_1$, $\mathcal{W}^<_2$, and $\mathcal{W}^>_2$, the *write-skew* condition is the following:

$$\mathcal{R}^\star_1 \cap \mathcal{W}^>_2 \neq \emptyset \quad \wedge \quad \mathcal{W}^>_1 \cap \mathcal{R}^\star_2 \neq \emptyset \quad \wedge \quad \mathcal{W}^<_1 \cap \mathcal{W}^<_2 = \emptyset$$

From this condition we generate a set of (in)equations, on the labels of repeating sequences, necessary to reach satisfiability. For instance, given the sets:

$$
\begin{aligned}
\mathcal{R}^\star &= \{this.head,\ this.head.A,\ this.head.A.next,\ this.head.A.next.next\} \\
\mathcal{W}^> &= \{this.head.B.next\}
\end{aligned}
$$

Table 4.2: StarTM applied to STM benchmarks.

Benchmark	Method	Time (sec.)	LOC	States	*Write-Skews*
List	add	5	16	2	(add, remove) (remove, remove)
	remove		14	2	
	contains		11	1	
	revert		11	4	
List Safe	add	6	16	2	-
	remove		15	2	
	contains		11	1	
Tree	treeAdd	11	21	3	-
	treeContains		15	2	
Intruder	atomicGetPacket	24	9	2	(atomicProcess, atomicGetComplete)
	atomicProcess		173	7	
	atomicGetComplete		15	2	

The condition $\mathcal{R}^\star \cap \mathcal{W}^> \neq \emptyset$ is satisfied if there is a possible instantiation of A and B such that:

$$B.next \leq A \quad \vee \quad B = A \quad \vee \quad B = A.next$$

In inequation $B.next \leq A$, the operator $\leq$ denotes prefixing, in this case that $B.next$ is a prefix of A. After generating the (in)equation system on labels (A, B) needed to satisfy the *write-skew* condition, we use a SMT solver to check their satisfiability. If a solution is found, it means that a *write-skew* may occur between the two transactions being analyzed. Notice that when comparing read- and write-sets we make the correspondence between concrete paths in the heap through the unique labels of repeating sequences.

4.8 Experimental Results

StarTM is a prototype implementation of our static analysis algorithm applied to Java bytecode, using the Soot toolkit [VRCGHLS99] and the CVC3 SMT solver [BT07]. We applied StarTM to three STM benchmarks: an ordered linked list, a binary search tree, and the Intruder test program of the STAMP benchmark. In the case of the list we tested two versions: the unsafe version called List and the safe version called List Safe. The List Safe version has an additional update in the `remove` method as discussed in Section 4.4.

Table 4.2 shows the detailed results of our verification for each transactional method of the examples above. The results were obtained in a Intel Dual-Core i5 650 computer, with 4 GB of RAM. We show the time (in seconds) taken by StarTM to verify each example, the number of lines of code, and the number of states produced during the analysis. The last column in the table shows the pairs of transactions that may actually trigger a *write-skew* anomaly.

The expected results for the two versions of the linked list benchmark were confirmed by our tool. The tool detects the existence of two *write-skew* anomalies, in the unsafe version of the linked list, resulting from the concurrent execution of the `add` and `remove` methods. The safe version is proven to be completely safe when executing all transactions under SI.

In the case of the Tree benchmark, the `treeAdd` method performs a tree traversal and inserts a new leaf node. StarTM proves that the concurrent execution of all transactions of the Tree

benchmark is safe.

StarTM detects a *write-skew* anomaly in the Intruder example, which is triggered by the concurrent execution of `atomicProcess` and `atomicGetComplete` transactions. This happens when the transaction `atomicProcess` pushes an element into a stack, implemented using an array with two integer pointers controlling the start and end of the stack, and the transaction `atomicGetComplete` pops an element from the same stack, which result in writes on different parts of the memory. However, the Intruder example is not entirely analyzed. A small part of the code cannot analyzed due to the use of arrays and cyclic data-structures, neither currently supported by our tool.

4.9 Related Work

Software Transactional Memory (STM) [ST95; HLMWNS03] systems commonly implement opacity to ensure the correct execution of concurrent programs. To the best of our knowledge, SI-STM [RFF06] is the only existing implementation of a STM using snapshot isolation. This work focuses on improving the transactional processing throughput by using a snapshot isolation algorithm. It proposes a SI safe variant of the algorithm, where anomalies are dynamically avoided by enforcing additional validation of read-write conflicts. Our approach avoids this validation by using static analysis and correcting the anomalies before executing the program.

In our work, we aim at providing opacity semantics under a run-time based on snapshot isolation for STM. This is achieved by performing a static analysis of the program and asserting that no SI anomalies will ever occur when executing a transactional application. This allows to avoid tracking read accesses in both read-only and read-write transactions, thus increasing performance throughput.

The use of snapshot isolation in databases is a common place, and there are some previous works on the detection of SI anomalies in this domain. Fekete et al. [FLOOS05] developed the theory of SI anomalies detection and proposed a syntactic analysis to detect SI anomalies for the database setting. They assume applications are described in some form of pseudo-code, without conditional (*if-then-else*) and cyclic structures. The proposed analysis is informally described and applied to the database benchmark TPC-C [Tra10] proving that its execution is safe under SI. A sequel of that work [JFRS07], describes a prototype which is able to automatically analyze database applications. Their syntactic analysis is based on the names of the columns accessed in the SQL statements that occur within the transaction.

Although targeting similar results, our work deals with different problems. The most significant one is related to the full power of general purpose languages and the use of dynamically allocated heap data structures. To tackle this problem, we use separation logic [Rey02; DOY06] to model operations that manipulate heap pointers. Separation logic has been the subject of research in the last few years for its use in static analysis of dynamic allocation and manipulation of memory, allowing one to reason locally about a portion of the heap. It has been proven to scale for larger programs, such as the Linux kernel [CDOY09].

The approach described in [RCG09] has a close connection to ours. It defines an analysis to detect memory independences between statements in a program, which can be used for parallelization. They extended separation logic formulae with labels, which are used to keep track of memory regions through an execution. They can prove that two distinct program fragments use disjoint memory regions on all executions, and hence, these program fragments can be safely

parallelized. In our work, we need a finer grain model of the accessed memory regions. We also need to distinguish between read and write accesses to shared and separated memory regions.

The work in [PRV10] informally describes a similar static analysis to approximate read- and write-sets using escape graphs to model the heap structure. Our shape analysis is based on separation logic, and, as far as we understand, heap-paths give a more fine-grain representation of memory locations at a possible expense in scalability.

Some aspects of our work are inspired by jStar [DPJ08]. jStar is an automatic verification tool for Java programs, based on separation logic, that enables the automatic verification of entire implementations of several design patterns. Although our work has some aspect in common with jStar, the properties being verified are completely different.

4.10 Concluding Remarks

We described a novel and sound approach to automatically verify the absence of the *write-skew* snapshot isolation anomaly in transactional memory programs. Our approach is based on a general model for fine grain abstract representation of accesses to dynamically allocated memory locations. By using this representation, we accurately approximate the concrete read- and write-sets of memory transactions, and capture *write-skew* anomalies as a consequence of the satisfiability of an assertion based on the output of the analysis, the abstract read- and write-sets.

We present StarTM, a prototype implementation of our theoretical framework, unveiling the potential for the safe optimization of transactional memory Java programs by relaxing isolation between transactions. Our approach is not without limitations. Issues that require further developments range from the generalization of the *write-skew* condition for more than two transactions, the support for richer dynamic data structures, to the support for array data types. Together with a runtime system support for mixed isolation levels, we believe that our approach can scale up to significantly optimize real-world transactional memory systems.

Publications The contents of this chapter were partially published in:

- [DLP11] **Efficient and correct transactional memory programs combining snapshot isolation and static analysis**. Ricardo J. Dias, João M. Lourenço, and Nuno M. Preguiça. In proceedings of HotPar 2011 (Workshop), May 2011.
- [DDSL12] **Verification of snapshot isolation in transactional memory java programs**. Ricardo J. Dias, Dino Distefano, João C. Seco, and João M. Lourenço. In proceedings of ECOOP 2012, June 2012.

5

Support of In-Place Metadata in Transactional Memory

An efficient technique to implement a snapshot isolation based transactional memory algorithm is to use multi-version concurrency control techniques. In a multi-version algorithm, several versions of the same data item may exist. In the particular case of transactional memory, several versions of the same memory block may exist. The efficient implementation of a multi-version algorithm requires a *one-to-one* correspondence between the memory block and the list of past versions. In this Chapter we propose an extension to a well known Java STM framework — the Deuce — that allows to efficiently implement multi-version algorithms and compare them against other kinds of STM algorithms. This chapter also includes the description and evaluation of an implementation of the proposed extension.

5.1 Introduction

Software Transactional Memory (STM) algorithms differ in the properties and guarantees they provide. Among others differences, one can list distinct strategies used to read (visible or invisible) and update memory (direct or deferred), the consistency (opacity or snapshot isolation) and progress guarantees (blocking or non-blocking), the policies applied to conflict resolution (contention management), and the sensitivity to interactions with non-transactional code (weak or strong atomicity). Some STM frameworks (e.g., DSTM2 [HLM06] and Deuce [KSF10]) address the need of experimenting with new STM algorithms and their comparison, by providing a unique transactional interface and different alternative implementations of STM algorithms. However, STM frameworks tend to favor the performance for some classes of STM algorithms and disfavor others. For instance, the Deuce framework favors algorithms like TL2 [DSS06] and LSA [RFF06], which are resilient to false sharing of transactional metadata (such as ownership

records) stored in an external table, and disfavor multi-version algorithms, which require unique metadata per memory location. This chapter addresses this issue by proposing an extension to the Deuce framework that allows the efficient support of transactional metadata records per memory location, opening the way to more efficient implementations of multi-version algorithms and consequently of snapshot isolation algorithms.

STM algorithms manage information per transaction (frequently referred to as a *transaction descriptor*), and per memory location (or object reference) accessed within that transaction. The transaction descriptor is typically stored in a thread-local memory space and maintains the information required to validate and commit the transaction. The per memory location information depends on the nature of the STM algorithm, and may be composed by, e.g., locks, timestamps or version lists, will henceforth be referred as *metadata*. Metadata is stored either adjacent to each memory location (*in-place* strategy), or in an external table (*out-place* or *external* strategy). STM libraries for imperative languages, such as C, frequently use the out-place strategy, while those addressing object-oriented languages bias towards the in-place strategy.

The out-place strategy is implemented by using a table-like data structure that efficiently maps memory references to its metadata. Storing the metadata in such a pre-allocated table avoids the overhead of dynamic memory allocation, but incurs in the overhead for evaluating the location-to-metadata mapping function. The bounded size of the external table also induces a false sharing situation, where multiple memory locations share the same table entry and hence the same metadata, in a *many-to-one* relation between memory locations and metadata units.

The in-place strategy is usually implemented using the *decorator* design pattern [GHJV94], by extending the functionality of an original class by wrapping it in a *decorator* class that contains the required metadata. This technique allows the direct access to the object metadata without significant overhead, but is very intrusive to the application code, which must be heavily rewritten to use the decorator classes instead of the original ones. The *decorator* pattern based technique bears two other problems: additional overhead for non-transactional code, and multiple difficulties while working with primitive and array types. The in-place strategy implements a *one-to-one* relation between memory locations and metadata units, thus no false sharing occurs. Riegel et al. [RB08] briefly describe the trade-offs of using in-place *versus* out-place strategies.

Deuce is among the most efficient STM frameworks for the Java programming language and provides a well defined interface that is used to implement several STM algorithms. On the application developer's side, a memory transaction is defined by adding the annotation `@Atomic` to a Java method, and the framework automatically instruments the application's bytecode to intercept the read and write memory accesses by injecting call-backs to the STM algorithm. These call-backs receive the referenced memory address as argument, hence limiting the range of viable STM algorithms to be implemented by forcing an out-place strategy. To implement an algorithm in Deuce that requires a one-to-one relation between metadata and memory locations, such as a multi-version algorithm, one needs to use a external table that handles collisions, which significantly degrades the throughput of the algorithm.

In the remaining of this chapter we present a novel approach to support the in-place metadata strategy that does not use the decorator pattern, and thoroughly evaluate its implementation in Deuce. This extension allows the efficient implementation of algorithms requiring a one-to-one relation between metadata and memory locations, such as multi-version algorithms. The developed extension has the following properties:

Efficiency The extension fully supports primitive types, even in transactional code. It does not

rely on an external mapping table, thus providing fast direct access to the transactional metadata. Transactional code does not require the extra memory dereference imposed by the decorator pattern. Non-transactional code is in general oblivious to the presence of metadata in objects, hence no significant performance overhead is introduced. And we propose a solution for supporting transactional n-dimensional arrays with a negligible overhead for non-transactional code.

Flexibility The extension supports both the original out-place and the new in-place strategies simultaneously, hence it is fully backwards compatible and imposes no restrictions on the nature of the STM algorithms to be used, nor on their implementation strategies.

Transparency The extension automatically identifies, creates and initializes all the necessary additional metadata fields in objects. No source code changes are required, although some light transformations are applied to the non-transactional bytecode. The new transactional array types — that support metadata at the array cell level — are compatible with the standard arrays, therefore not requiring pre- and post-processing of the arrays when used as arguments in calls to the standard JDK or third-party non-transactional libraries.

Compatibility Our extension is fully backwards compatible and the already existing implementations of STM algorithms are executed with no changes and with zero or negligible performance overhead.

Compliance The extension and bytecode transformations are fully-compliant with the Java specification, hence supported by standard Java compilers and JVMs.

This extension allows to efficiently implement snapshot isolation STM algorithms on top of multi-version techniques. We implemented a snapshot isolation algorithm and evaluated its performance against *opaque* algorithms. We used micro-benchmarks that are safe under SI, as reported in the previous chapter, such as the Linked List.

The Deuce framework assumes a weak atomicity model, i.e., transactions are atomic only with respect to other transactions, and hence, their execution may be interleaved with non-transactional code. Multi-version algorithms store the values of memory blocks in transactional metadata objects (which contain the version lists), and therefore non-transactional memory accesses cannot *see* transactional updates, nor transactional accesses can see non-transactional updates. We tackle this problem by proposing an algorithmic adaptation for multi-version algorithms that allows to support a weak atomicity model for multi-version algorithms with meaningless impact on the performance in general.

This chapter follows with a description of the Deuce framework and its out-place strategy in Section 5.2. Section 5.3 describes properties of the in-place strategy, its implementation, and its limitations as an extension to Deuce. We present an evaluation of the extension's implementation using several metrics in Section 5.4. Section 5.5 describes the implementation of several state-of-the-art STM multi-version algorithms using our proposed extension. In Section 5.6 we show how to adapt the multi-version algorithms to support a weak-atomicity model. Finally, we present a comparison between different single- and multi-version algorithms using standard benchmarks in Section 5.7.

5.2 Deuce and the Out-Place Strategy

Deuce supplies a single `@Atomic` Java annotation, and relies heavily on bytecode instrumentation to provide a transparent transactional interface to application developers, which are unaware of how the STM algorithms are implemented and which strategies they use to store the transactional metadata.

Algorithms such as TL2 [DSS06] or LSA [RFF06] use an out-place strategy by resorting to a very fast hashing function and storing a single lock in each table entry. However, due to performance issues, the mapping table does not avoid hash collisions and thus two memory locations may be mapped to the same table entry, resulting in the false sharing of a lock by two different memory locations. In these algorithms, the false sharing may cause transactions to fail and abort that otherwise would succeed, hurting the system performance but never compromising the correctness.

The out-place strategy suits algorithms where metadata information does not depend on the memory locations, such as locks and timestamps, but not algorithms that need to keep location-dependent metadata information, such as multi-version algorithms. The out-place implementations of these algorithms require a mapping table with collision lists, which significantly degrades performance.

Deuce provides the STM algorithms with a unique identifier for each object field, composed by the reference to the object and the field's logical offset within that object. This unique identifier is then used by the STM algorithms as the key to any map implementation that associates the object's field with the transactional metadata. Likewise for arrays, the unique identifier of an array's cell is composed by the array reference and the index of that cell.

The performance of STM algorithms are known to depend with both the hardware and the transactional workload, and a thorough experimental evaluation is required to assess the optimal combination of the triple hardware–algorithm–workload. Deuce is an extensible STM framework that may be used to address such comparison of different STM algorithms. However, Deuce is biased towards the out-place strategy, allowing very efficient implementations for some algorithms like TL2 and LSA, but hampering some others, like the multi-version oriented STM algorithms.

To support the out-place strategy, Deuce identifies an object's field by the object reference and the field's logical offset. This logical offset is computed at compile time, and for every field f in every class C an extra static field f^o is added to that class, whose value represents the logical offset of f in class C. No extra fields are added for array cells, as the logical offset of each cell corresponds to its index. Within a memory transaction, when there is a read or write memory access to a field f of an object O, or to the array element $A[i]$, the runtime passes the pair (O, f^o) or (A, i) respectively as the argument to the call-back function. The STM algorithm shall not differentiate between field and array accesses. If an algorithm wants to, e.g., associate a lock with a field, it has to store the lock in an external table indexed by the hash value of the pair (O, f^o) or (A, i). STM algorithm implementations must comply with a well defined Java interface, as depicted in Figure 5.1. The methods specified in the interface are the call-back functions that are injected by the instrumentation process in the application code. For each read and write of a field of an object, the methods `onReadAccess` and `onWriteAccess`, are invoked respectively. The method `beforeReadAccess` is called before the actual read of an object's field.

We have extended Deuce to support an efficient in-place strategy, in addition to the already

```
1  public interface Context {
2    void init(int atomicBlockId, String metainf);
3    boolean commit();
4    void rollback();
5
6    void beforeReadAccess(Object obj, long field);
7
8    int onReadAccess(Object obj, int value, long field);
9    // ... onReadAccess for the remaining types
10
11   void onWriteAccess(Object obj, int value, long field);
12   // ... onWriteAccess for the remaining types
13 }
```

Figure 5.1: `Context` interface for implementing an STM algorithm.

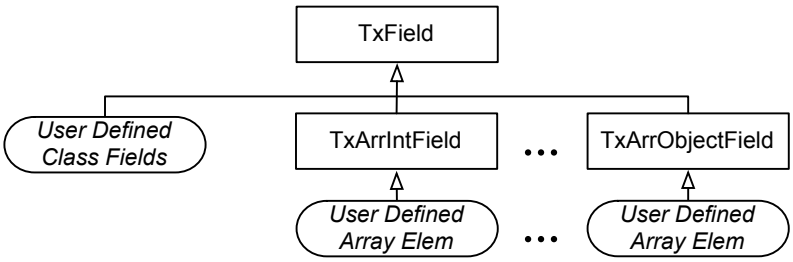


Figure 5.2: Metadata classes hierarchy.

existing out-place strategy, while keeping the same transparent transactional interface to the applications.

5.3 Supporting the In-Place Strategy

In our approach to extend Deuce to support the in-place strategy, we replace the previous pair of arguments to call-back functions (O, f^o) with a new metadata object f^m, whose class is specified by the STM algorithm's programmer. We guarantee that there is a unique metadata object f^m for each field f of each object O, and hence the use of f^m to identify an object's field is equivalent to the pair (O, f^o). The same applies to arrays, where we ensure that there is a unique metadata object a^m for each position of any array A.

5.3.1 Implementation

Although the implementation of the support for in-place metadata objects differs considerably for class fields and array elements, a common interface is used to interact with the STM algorithm implementation. This common interface is supported by a well defined hierarchy of metadata classes, illustrated in Figure 5.2, where the rounded rectangle classes are defined by the STM algorithm developer.

All metadata classes associated with class fields extend directly from the top class `TxField` (see Figure 5.3). The constructor of `TxField` class receives the object reference and the logical offset of the field. All subclasses must call this constructor. For array elements, we created

```
1  public class TxField {
2    public Object ref;
3    public final long offset;
4
5    public TxField(Object ref, long offset) {
6      this.ref = ref;
7      this.offset = offset;
8    }
9  }
```

Figure 5.3: `TxField` class.

```
1  public interface ContextMetadata {
2    void init(int atomicBlockId, String metainf);
3    boolean commit();
4    void rollback();
5
6    void beforeReadAccess(TxField field);
7    int onReadAccess(int value, TxField field);
8    // ... onReadAccess for the remaining types
9
10   void onWriteAccess(int value, TxField field);
11   // ... onWriteAccess for the remaining types
12 }
```

Figure 5.4: `Context` interface for implementing an STM algorithm supporting in-place metadata.

specialized metadata classes for each primitive type in Java, the `TxArr*Field` classes, where `*` ranges over the Java primitive types[1]. All the `TxArr*Field` classes extend from `TxField`, providing the STM algorithm with a simple and uniform interface for call-back functions.

We defined a new interface for the call-back methods (see Figure 5.4). In this new interface, the read and write call-back functions (`onReadAccess` and `onWriteAcess` respectively) receive only the metadata `TxField` object, not the object reference and logical offset of the `Context` interface. This new interface coexists with the original one in Deuce, allowing new STM algorithms to access the in-place metadata while ensuring backward compatibility.

The `TxField` class can be extended by the STM algorithm programmer to include additional information required by the algorithm for, e.g., locks, timestamps, or version lists. The newly defined metadata classes need to be registered in our framework to enable its use by the instrumentation process, using a Java annotation in the class that implements the STM algorithm, as exemplified in Figure 5.5. The programmer may register a different metadata class for each kind of data type, either for class field types or array types. As shown in the example of Figure 5.5, the programmer registers the metadata implementation class `TL2IntField` for the fields of **`int`** type, by assigning the name of the class to the `fieldIntClass` annotation property.

The STM algorithm must implement the `ContextMetadata` interface (Figure 5.4) that includes a call-back function for the read and write operations on each Java type. These functions always receive an instance of the super class `TxField`, but no confusion arises from there, as each

[1] **`int`**, **`long`**, **`float`**, **`double`**, **`short`**, **`char`**, **`byte`**, **`boolean`**, and `Object`.

```
@InPlaceMetadata(
  fieldObjectClass="TL2ObjField",
  fieldIntClass="TL2IntField",
  ...
  arrayObjectClass="TL2ArrObjectField",
  arrayIntClass="TL2ArrIntField",
  ...
)
public class TL2Context implements ContextMetadata {
  ...
}
```

Figure 5.5: Declaration of the STM algorithm specific metadata.

```
class C {
  int a;
  Object b;
}
```

$\Longrightarrow$

```
class C {
  int a;
  Object b;
  final TxField a_metadata;
  final TxField b_metadata;
}
```

Figure 5.6: Example transformation of a class with the in-place strategy.

algorithm knows precisely which metadata subclass was actually used to instantiate the metadata object.

Lets now see where and how the metadata objects are stored, and how they are used on the invocation of the call-back functions. We will explain separately the management of metadata objects for class fields and for array elements.

5.3.1.1 Adding Metadata to Class Fields

During the execution of a transaction, there must be a metadata object f^m for each accessed field f of object O. Ideally, this metadata object f^m is accessible by a single dereference operation from object O, which can be achieved by adding a new metadata field (of the corresponding type) for each field declared in a class C. The general rule for this process can be described as: given a class C that has a set of declared fields $F = \{f_1, \ldots, f_n\}$, for each field $f_i \in F$ we add a new metadata object field f^m_{i+n} to C, such that the class ends with the set of fields $F^m = \{f_1, \ldots, f_n, f^m_{1+n}, \ldots, f^m_{n+n}\}$, where each field f_i is associated with the metadata field f^m_{i+n} for any $i \leq n$. In Figure 5.6 we show a concrete example of the transformation of a class with two fields.

Instance and static fields are expected to have instance and static metadata fields, respectively. Thus, instance metadata fields are initialized in the class constructor, while static metadata fields are initialized in the static initializer (`static { ... }`). This ensures that whenever a new instance of a class is created, the corresponding metadata objects are also new and unique, while static metadata objects are the same in all instances. Since a class can declare multiple constructors that can call each other, using the *telescoping constructor* pattern [Blo08], blindly instantiating the metadata fields in all constructors would be redundant and impose unnecessary stress on the garbage collector. Therefore, the creation and initialization of metadata objects only takes place

in the constructors that do not rely in another constructor to initialize its target.

Opposed to the transformation approach based in the *decorator* pattern, where primitive types must be replaced with their object equivalents (e.g., in Java an **int** field is replaced by an `Integer` object), our transformation approach keeps the primitive type fields untouched, simplifying the interaction with non-transactional code, limiting the code instrumentation and avoiding auto-boxing and its overhead.

5.3.1.2 Adding Metadata to Array Elements

The structure of an array is very strict. Each array cell contains a single value of a well defined type and no other information can be added to those cells. The common approach to overcome this limitation and add some more information to each cell, is to change the original array to an array of objects that wrap the original value and the additional information. This straight-forward transformation has many implications in the application, as code statements accessing the original array or array elements will now have to be rewritten to use the new array type or wrapping class respectively. This problem is even more complex if the new arrays with wrapped elements are to be manipulated by non-instrumented libraries, such as the JDK libraries, which are unaware of the new array types.

While the instrumentation process can replace the original arrays with the new arrays where needed, the straight-forward transformation approach needs to be able to revert back to the original arrays when presented with non-instrumented code. For example, consider that the application code is invoking the non-instrumented method `Arrays.binarySearch(int[], int)` from the Java platform. Throughout the instrumented code **int**`[]` has been replaced by a new type, which we denote as `IntWrapper[]`. As the `binarySearch` method was not instrumented, the array parameter remains of type **int**`[]`, thus one needs to construct a temporary **int**`[]` array with the same state of the `IntWrapper[]` array, which can then be passed as an argument to the `binarySearch` method. From the caller perspective, the non-instrumented method itself is a black box which may have modified some array cells.[2] Hence, unless we were to build some kind of black/white list with such information for *all* non-instrumented methods, the values from the temporary **int**`[]` array have to be copied back to the original `IntWrapper[]` array. All these memory allocation and copies significantly hamper the performance when executing non-instrumented code, which should not be affected due to transactional-related instrumentation. We call the straight-forward approach just described the *naïve solution*.

The solution we propose is also based on changing the type of the array to be manipulated by the instrumented application code, but with minimal impact on the performance of non-instrumented code. We keep all the values in the original array, and have a sibling second array, only manipulated by the instrumented code, that contains the additional information and references to the original array. The type in the declaration of the base array is changed to the type of the corresponding sibling array (`TxArr*Field`), as shown in Figure 5.7. This Figure also illustrates the general structure of the sibling `TxArr*Field` arrays (in this case, a `TxArrIntField` array). Each cell of the sibling array has the metadata information required by the STM algorithm, its own position/index in the array, and a reference to the original array where the data is stored (i.e., where the reads and updates take place). This scheme allows the sibling array to keep a

[2] In this example we used the `binarySearch` method which does not modify the array, but in general we do not know.

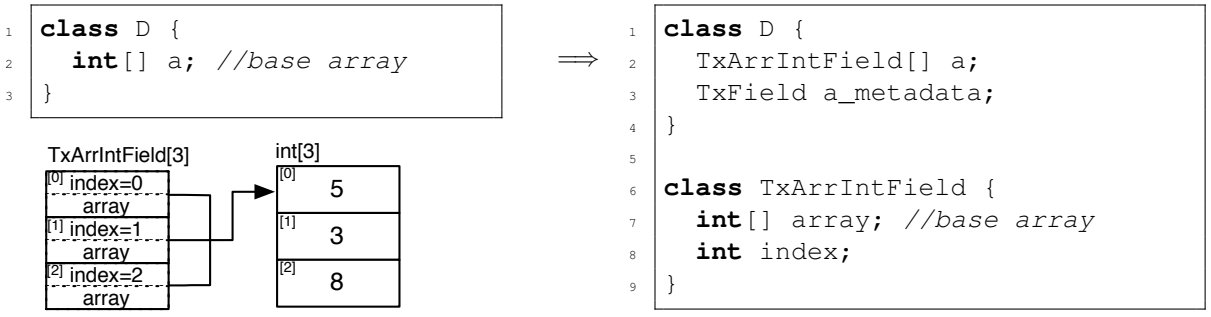


Figure 5.7: Memory structure of a `TxArrIntField` array.

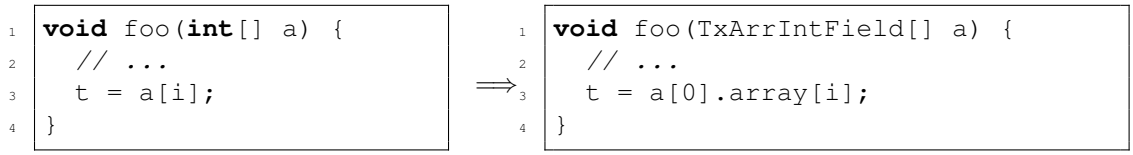


Figure 5.8: Example transformation of array access in the in-place strategy.

metadata object for each element of the original array, while maintaining the original array always updated and compatible with non-transactional legacy code. With this approach for adding metadata support to arrays, the original array can still be retrieved with a minimal overhead by dereferencing twice the sibling `TxArr*Field` array. Since the original array serves as the backing store, no memory allocation or copies need to be performed, even when array elements are changed by non-instrumented code. We call our proposed solution the *efficient solution*.

Non-transactional methods that have arrays as parameters are also instrumented to replace the array type by the corresponding sibling `TxArr*Field`. For non-instrumented methods, relying on the method signature is not enough to know if there is the need to revert to primitive arrays. Take, for example, the `System.arraycopy(Object, int, Object, int, int)` method from the Java platform. The signature refers `Object` but it actually receives arrays as arguments. We identify these situations by inspecting the type of the arguments on a *virtual stack*[3] and if an array is found, despite the method's signature, we revert to primitive arrays. The value of an array element is then obtained by dereferencing the pointer to the original array kept in the sibling, as illustrated in Figure 5.8. When passing an array as argument to an uninstrumented method (e.g., from the JDK library), we can just pass the original array instance. Although the instrumentation of non-transactional code adds an extra dereference operation when accessing an array, we still do avoid the auto-boxing of primitive types, which would impose a much higher overhead.

5.3.1.3 Adding Metadata to Multi-Dimensional Arrays

The special case of multi-dimensional arrays is tackled using the `TxArrObjectField` class, which has a different implementation from the other specialized metadata array classes. This class has an additional field, `nextDim`, which may be null in the case of a unidimensional reference type array, or may hold the reference of the next array dimension by pointing to another

[3]During the instrumentation process we keep the type information of the operand stack.

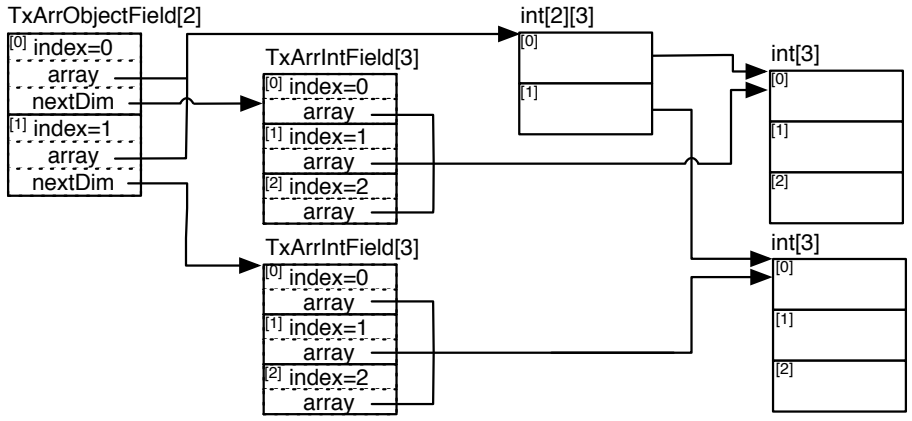


Figure 5.9: Memory structure of a multi-dimensional `TxArrIntField` array.

Table 5.1: Comparison between primitive and transactional arrays.

Arrays	Access n^{th} dimension	Objects	Non-transactional methods
Primitive arrays	n *derefs*	$\sum_{i=1}^{n} l_{i-1}$	—
Instrumented arrays	$2n+1$ *derefs*	$\sum_{i=1}^{n} 2l_{i-1} + (l_i \times l_{i-1})$	2 *derefs*

n (dimensions), l_i (length of i^{th} dimension)

array of type `TxArr*Field`. Once again, the original multi-dimensional array is always up to date and can be safely used by non-transactional code.

Figure 5.9 depicts the memory structure of a bi-dimensional array of integers. Each element of the first dimension of the sibling array has a reference to the original integer matrix. The elements of the second dimension of the sibling array have a reference to the second dimension of the matrix array.

Table 5.1 provides a comparison between the regular primitive arrays, used in the out-place strategy, and our instrumented arrays, used in the in-place strategy. The instrumented arrays follow the strategy described above. For accessing a cell in a n-dimensional array (Table 5.1, second column), in a primitive array it takes n object dereferences, i.e., dereferencing all intermediate dimension arrays and directly accessing the cell. With our array instrumentation it takes $2n+1$ dereferences, introducing an extra dereference per dimension ($2n$) because each cell is now a `TxArr*Field`. Since the original array is used as the backing store, there is an additional dereference of the original array in the last dimension to access the value. Regarding the number of objects that each approach needs for an n-dimensional array (Table 5.1, third column), for simplicity's sake let's assume that all intermediate i^{th}-dimensional arrays have the same length, l_i. Primitive arrays have l_{i-1} objects per dimension, i.e., each dimension's array cell is a reference to another array, except in the last dimension. The instrumented arrays have twice the number of arrays, i.e., $2l_{i-1}$, corresponding to the the original array (which is kept) and the sibling array, plus an extra `TxArr*Field` in every array cell ($l_i \times l_{i-1}$). When an array is to be used by a non-instrumented method (Table 5.1, fourth column), the instrumented arrays require two dereferences to obtain the backing-store primitive array, i.e., dereferencing the sibling array followed by a dereference of a `TxArr*Field` cell, from which the `array` field can be used. These two

dereferences required by our instrumented arrays contrast with the expensive memory allocation and copies necessary for the straight-forward naïve solution, described in 5.3.1.2.

5.3.2 Instrumentation Limitations

Some Java core classes, mostly in the `java.lang` package, are loaded and initialized during the JVM bootstrap. Because these classes are loaded upon JVM startup, they can either be redefined online after the bootstrap, or require an offline, static, instrumentation. Online redefinition of classes has many and strong limitations, and its support is an optional functionality for JVMs [Ora12]. For this reason, instead of online redefinition of bootstrap-loaded classes, Deuce provides an offline instrumentation process.

Most JVMs are very sensitive with regard to the order in which classes are loaded during the bootstrap. If that order is changed due to the execution of instrumented code during the bootstrapping phase (i.e., because instrumented code may depend on certain classes that need to be loaded before the instrumented code can be executed), the JVM may crash [BHM07]. The Deuce online instrumentation injects static fields and their initialization, which would disrupt the class loading order if done on bootstrap-loaded classes. Deuce solves this problem in the offline instrumentation by creating a separate class to hold the fields instead. This is possible because the necessary fields are static.

The instrumentation to support the in-place metadata strategy is more complex, requiring the injection of instance fields and modifying arrays. For this reason, the instrumentation of bootstrap-loaded classes is not supported by our current instrumentation process, as these transformations disrupt the bootstrap class loading order by loading the metadata and transactional array classes.

At the moment there is no support for structural modification of arrays inside non-instrumented code, such as the java runtime library, because the solution for metadata at array element level relies on a sibling array where a structural invariant exists between the sibling array and the original array. If a non-instrumented method modifies the original array, the structural invariant is broken and both structures become different.

5.4 Implementation Assessment

The implementation of the proposed Deuce extension, described in the previous sections, introduces more complexity to the transactional processing when comparing with the original Deuce implementation. This complexity, in the form of additional memory operations and allocations, may slowdown the performance in some cases. In our first step to assess the extension implementation performance, we evaluate the overhead of the new implementation by comparing it with the original Deuce implementation.

In a second step we evaluate the performance speedup of using our extension to implement a multi-version STM algorithm, against an implementation of the same algorithm using the original Deuce interface. We chose a well known multi-version STM algorithm, JVSTM, described in [CRS06], and implemented two versions of the algorithm, one using the original Deuce interface and an out-place strategy (referred to as `jvstm-outplace`), and another using our new interface and extension supporting an in-place strategy (referred to as `jvstm-inplace`).

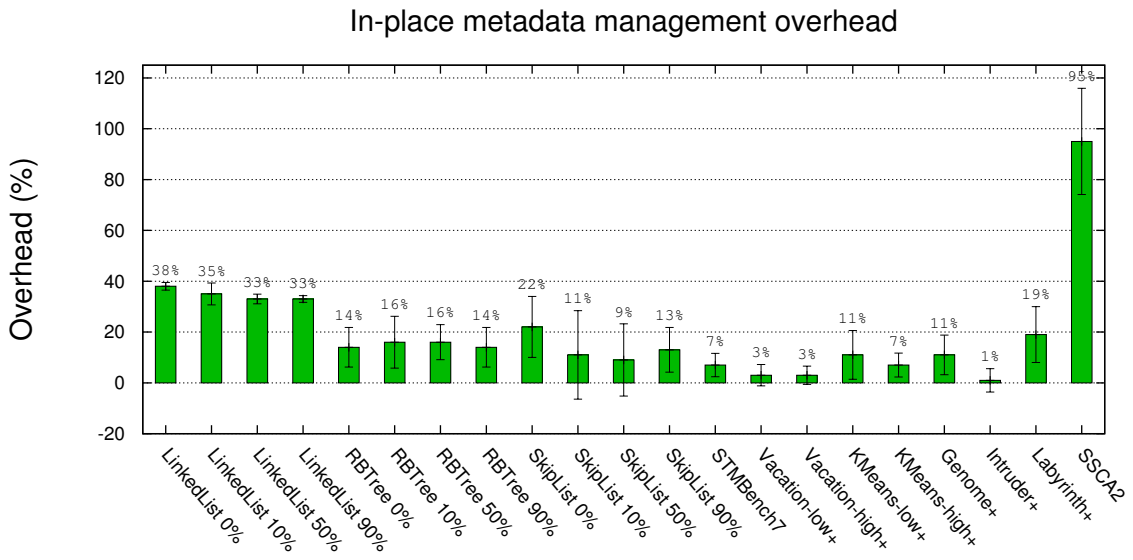


Figure 5.10: Performance overhead measure of the usage of metadata objects relative to out-place TL2.

Both the overhead and speedup evaluations are preformed using several micro- and macro-benchmarks. Micro-benchmarks are composed by the Linked List, Red-Black Tree, and Skip-List data structures. Macro-benchmarks are composed by the STAMP [CMCKO08] benchmark suite and the STMBench7 [GKV07] benchmark. All these benchmarks were executed in our extension of Deuce with in-place metadata with no changes whatsoever, as all the necessary bytecode transformations were performed automatically by our instrumentation process.

The benchmarks were executed on a computer with four AMD Opteron 6272 16-Core processors @ 2.1 GHz with 8×2 MB of L2 cache, 16 MB of L3 cache, and 64 GB of RAM, running Debian Linux 3.2.41 x86_64, and Java 1.7.0_21.

In the following sections we describe in detail, and present the results, of the overhead evaluation as well as the speedup evaluation.

5.4.1 Overhead Evaluation

To evaluate the performance overhead of our extension, we compared the performance of the TL2 algorithm as provided by the original Deuce distribution, with another implementation of TL2 (`tl2-overhead`) using the new interface of our modified Deuce (as described in Figure 5.4 in page 100). The original Deuce interface for callback functions provide a pair with the object reference and the field logical offset. The new interface provides a reference to the field metadata (`TxField`) object. Despite using the in-place metadata feature, the `tl2-overhead` implementation is as much alike as the original as possible, and still uses an external table to map memory references to locks. The main differences between the two versions reside in the additional management of metadata objects (allocation, and array manipulation), and the two additional dereferences on the metadata object to obtain the field's object reference and the field offset, for each read and write operation. By comparing these two very similar implementations, we can make a reasonable estimation of the performance overhead introduced by the management of the metadata object fields and sibling arrays.

Figure 5.10 depicts the average and standard deviation of the performance overhead of `tl2--overhead` implementation with respect to the original Deuce TL2 implementation. The Figure reports on several benchmarks, with executions ranging form 1 to 64 threads. Appendix A.1 presents the detailed results for each benchmark. The overhead of the additional management of metadata objects and sibling arrays is in average about 20%. The benchmarks that use metadata arrays (SkipList, Kmeans, Genome, Labyrinth, SSCA2) have in general a higher overhead than the benchmarks that only use metadata objects for class fields (RBTree, STMBench7, Vacation, Intruder). The micro-benchmarks were all tested in four scenarios: with a read-only workload (0% of updates), and read-write workloads (10%, 50%, and 90% of updates). These micro-benchmarks are composed of small transactions which only perform read and write accesses to shared memory, and thus, the overhead is more visible. The LinkedList benchmark has a high overhead and does not use metadata arrays. This benchmark has long running transactions that perform a very high number of read operations, and our extension requires an external table lookup and an additional object dereference to retrieve the metadata object for each memory read operation.

The STAMP benchmarks, show relatively low overhead with the exception of SSCA2+ benchmark. These benchmarks have medium sized transactions which perform some computations with the data read from the shared memory. The SSCA2+ benchmark only preforms read and write operations over arrays, and may be considered the worst-case scenario for our extension.

The STMBench7 benchmark was executed with a read-dominant workload, without long-traversals, and with structural modifications activated. In this benchmarks transactions are computationally much heavier, which hides the small overhead introduced by the management of in-place metadata.

From this results we can conclude that the extension introduces a small overhead due to the management of in-place metadata, and additionally it allows the efficient implementation of a class of STM algorithms that require a *one-to-one* relation between memory locations and their metadata. Multi-version based algorithms fit into that class, as they associate a list of versions (holding past values) with each memory location.

In the next sections we show the comparison of the performance of the same multi-version algorithm implemented in the original Deuce framework and implemented using our extension.

5.4.2 Implementing a Multi-Versioning Algorithm: JVSTM

The JVSTM algorithm defines the notion of version box (*vbox*), which maintains a pointer to the head of an unbounded list of versions, where each version is composed by a timestamp and the data value. Each version box represents a distinct memory location. The timestamp in each version corresponds to the timestamp of the transaction that created that version, and the head of the version list always points to the most recent version.

During the execution of a transaction, the read and write operations are done in versioned boxes, which hold the data values. For each write operation a new version is created and tagged with the transaction timestamp. For read operations, the version box returns the version with the highest timestamp less than or equal to the transaction's timestamp. A particularity of this algorithm is that read-only transactions never abort, neither do write-only transactions. Only read-write transactions may conflict, thus aborting.

On committing a transaction, a global lock must be acquired to ensure mutual exclusion with all other concurrent transactions. Once the global lock is acquired, the transaction validates the

read-set, and in case of success, creates the new version for each memory location that was written, and finally releases the global lock. To prevent version lists from growing indefinitely, versions that are no more necessary are cleaned up by a *vbox* garbage collector.

To implement the JVSTM algorithm, we need to associate a *vbox* with each field of each object. For the sake of the correctness of the algorithm, this association must guarantee a relation of *one-to-one* between the *vbox* and the object's field. We will detail the implementation of this association for both, the out-place and the in-place strategies.

5.4.2.1 Out-Place Strategy

To implement JVSTM algorithm in the original Deuce framework, which only supports the out-place strategy, the *vboxes* must be stored in an external table[4]. The *vboxes* are indexed by a unique identifier for the object's field, composed by the object reference and the field's logical offset.

Whenever a transaction performs a read or write operation on an object's field, the respective *vbox* must be retrieved from the table. In the case where the *vbox* does not exists, we must create one and add it into the table. These two steps, verifying if a *vbox* is present in the table and creating and inserting a new one if not, must be performed atomically, otherwise we would incur in the case where two different *vboxes* may be created for the same object's field. Once the *vbox* is retrieved from the table, either it is a read operation and we look for the appropriate version using the transaction's timestamp and return the version's value, or it is a write operation and we add an entry to the transaction's write-set.

We use weak references in the table indices to reference the *vbox* objects and not hamper the garbage collector from collecting old objects. Whenever an object is collected our algorithm is notified in order to remove the respective entry from the table.

Despite using a concurrent hash map, this implementation suffers from a high overhead penalty when accessing the table, since it is a point of synchronization for all the transactions running concurrently. This implementation (`jvstm-outplace`) will be used as a base reference when comparing with the implementation of the same JVSTM algorithm using the in-place strategy (`jvstm-inplace`).

5.4.2.2 In-Place Strategy

The in-place version of JVSTM algorithm makes use of the metadata classes to hold the same information as the *vbox* in the out-place variant. This will allow direct access to the version list whenever a transaction is reading or writing.

We extend the *vbox* class from the `TxField` class as shown in Figure 5.11.

The actual implementation creates a `VBox` class for each Java type in order to prevent the boxing and unboxing of primitive types. When the constructor is executed, a new version with timestamp zero is created, containing the current value of the field identified by object `ref` and logical offset `offset`. The value is retrieved using the private method `read()`.

The code to create these `VBox` objects during the execution of the application is inserted automatically by our bytecode instrumentation process. The lifetime of an instance of the class `VBox` is the same as the lifetime of the object `ref`. When the garbage collector decides to collect the object `ref`, all metadata objects of class `VBox` associated with each field of the object `ref`, are also collected.

[4]We opted to use a concurrent hash table from the `java.util.concurrent` package.

```
1  public class VBox extends TxField {
2    protected VBoxBody body;
3
4    public VBox(Object ref, long offset) {
5      super(ref, offset);
6      body = new VBoxBody(read(), 0, null);
7    }
8
9    // ... methods to access and commit versions
10 }
```

Figure 5.11: `VBox` in-place implementation.

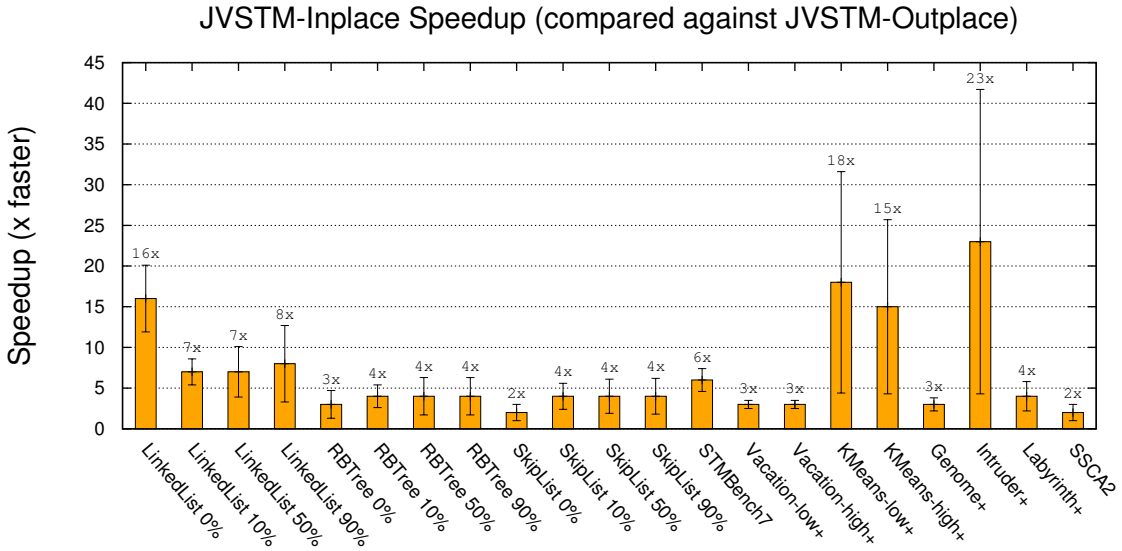


Figure 5.12: In-place over Out-place strategy speedup: the case of JVSTM.

Our comparison evaluation shows that the direct access to the version list allowed by the in-place strategy will greatly benefit the performance of the algorithm. We present the comparison results in the next section by presenting the speedup of the in-place version with respect to the out-place version.

5.4.3 Speedup Evaluation

From the evaluation of the in-place management overhead, we concluded that this strategy is a viable option for implementing algorithms biased to in-place transactional metadata. Hence, we implemented and evaluated two versions of the JVSTM algorithm as proposed in [CRS06], one in the original Deuce using the native out-place strategy (`jvstm-outplace`), and another in the extended Deuce using our in-place strategy (`jvstm-inplace`), as described in the previous Section 5.4.2.

Figure 5.12 depicts the average speedup of our two implementations of the JVSTM algorithm: one In-Place (`jvstm-inplace`) and another Out-Place (`jvstm-outplace`). We used the same set of benchmarks and configuration that was used for the overhead evaluation in Section 5.4.1.

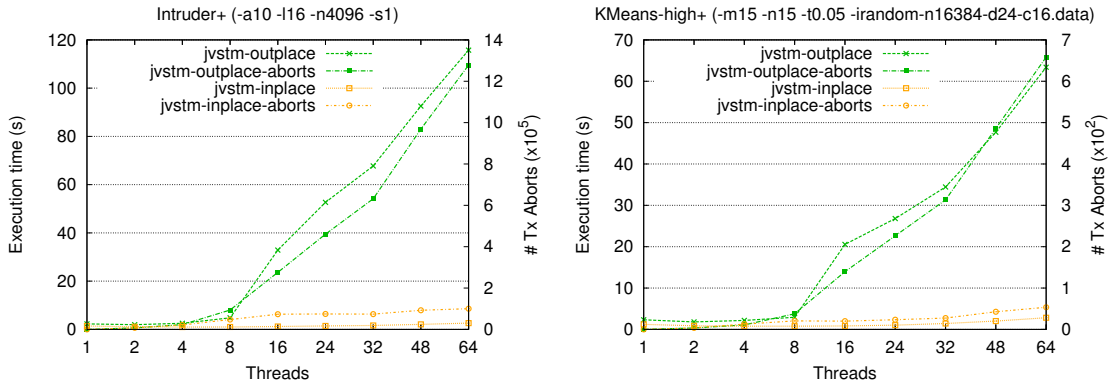


Figure 5.13: Performance and transaction aborts of JVSTM-Inplace/Outplace for the Intruder and KMeans benchmarks.

In Appendix A.2 we also present the results in detail for each benchmark. The in-place version of the JVSTM algorithm is in average 7 times faster than its dual out-place version.

The speedup observed for the micro-benchmarks, where transactions are small and contention is low, shows that the multi-versioning algorithms greatly benefit from our in-place support. In the case of the STAMP benchmarks, where transactions are submitted to workloads of intensive contention, the in-place version is much faster than the out-place approach as it avoids completely the use of a shared external table, which becomes a serious bottleneck in the presence of high contention. In the special case of KMeans and Intruder benchmarks, the overhead of managing a shared external table drastically increases the probability of transaction aborts as depicted in Figure 5.13, which in turn makes the transactional throughput to decrease. The STMBench7 macro-benchmark has many long-running transactions and the overall throughput for both algorithms is relatively low. Even so, the in-place algorithm is in average $6\times$ faster.

5.4.4 Memory Consumption Evaluation

To assess the impact of the in-place strategy in memory usage, we measured the memory consumption of the algorithms we described and used before, namely `tl2`, `tl2-overhead`, `jvstm-outplace` and `jvstm-inplace`. The comparison of the two `tl2` variants shall give an insight about the additional memory overhead imposed by the use of in-place metadata. Please remember that the `tl2-overhead` variant uses in-place metadata just to reference the locks, associated with each object's field, stored in an external table. Hence, the `tl2-overhead` should use the same amount of memory as the `tl2` variant plus the memory consumed by the metadata objects. The comparison of the two `jvstm` variants assess the additional memory benefits of using the in-place metadata strategy, besides the performance improvement. The `jvstm-outplace` variant needs to store the version lists in a shared external table which also consumes memory, and the garbage collection of these version lists is done manually using weak references and reference queues which may originate a greater memory footprint.

Figure 5.14 depicts the relative maximum consumed memory for each pair of algorithms. The result of `tl2-overhead` variant is relative to `tl2`, and the result of `jvstm-inplace` is relative to `jvstm-outplace`. The results correspond to how much more or much less memory is consumed by each algorithm relative to its counter part. We measured the average and standard deviation of the maximum consumed memory for each benchmark, which were executed in the

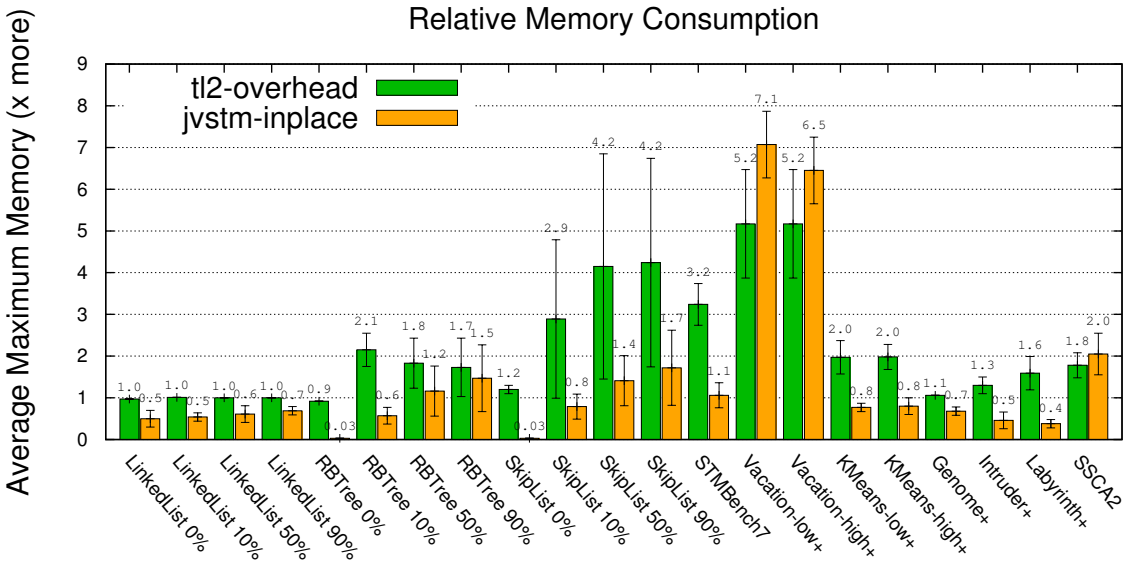


Figure 5.14: Relative memory consumption of TL2-Overhead and JVSTM-Inplace

same environment and configuration as in the previous evaluations. We will first discuss the comparison of the `tl2` variants and then the `jvstm` variants.

TL2 The use of in-place metadata in TL2 adds an extra object for each existing field of an object, and in the case of the arrays it more than duplicates the number of objects. The `tl2-overhead` results depicted in Figure 5.14 show this behavior. In the LinkedList example the consumed memory is roughly the same, as each node of the list only has one non-final field, the `next` field. In the case of the Red-Black Tree, each node has five non-final fields, and for this case the `tl2-overhead` variant consumes in average $1.6\times$ more memory. The Skip List benchmark, uses arrays to store forward pointers. Each node has an array of objects, and in this case the `tl2-overhead` consumes in average almost $3\times$ more memory. These micro-benchmarks results show that the additional cost in memory usage introduced by the in-place strategy is small when compared with the performance benefits as reported in the previous section.

In the STMBench7 benchmark, which performs very long operations, on a very big data structure, the use of in-place metadata objects only duplicates the memory consumption.

The STAMP benchmarks, use a mix of objects and array objects workload. Nevertheless, the average of memory consumption is about $2.5\times$ more than the `tl2` variant. The Vacation benchmark reports a higher amount of consumed memory. This is due to the use of several red-black trees and also the use of arrays.

JVSTM Opposed to the `jvstm-outplace` variant, the `jvstm-inplace` variant does not need an external table to store the vboxes. Instead, each vbox is made into a metadata object and is stored near to its respective object's field. Moreover, the vboxes are garbage collected automatically by the JVM when the objects are no longer reachable. These differences to the `jvstm-outplace` variant are sufficient to get in general a lower memory footprint than the `jvstm-outplace` as shown by the results in Figure 5.14. The exception is the Vacation benchmark where `jvstm-inplace` consumes between 6 and $7\times$ more memory. This strange result is explained by the way the vboxes are initialized in each variant. The Vacation benchmark creates

several red-black trees with a large number of nodes in the beginning of the execution. In the `jvstm-inplace` variant, the vboxes are instantiated when the node is created, and hence, when the transactional work starts all vboxes associated with the data structures are already instantiated. Contrarily, the `jvstm-outplace` only creates the vbox when the node is accessed, and thus, if some nodes are never accessed then the respective vboxes are not created as well, saving some memory. This is what happens in the Vacation benchmark. The `jvstm-outplace` variant does not create all the vboxes that are created by the `jvstm-inplace` variant, and therefore, the `jvstm-inplace` variant gets a higher memory footprint than the `jvstm-outplace` variant.

5.5 Use Case: Multi-version Algorithm Implementation

Our main purpose for extending Deuce with support for in-place metadata was to allow the efficient implementation of a class of STM algorithms that require a *one-to-one* relation between memory locations and their metadata. Multi-version based algorithms fit into that class, as they associate a list of versions (holding past values) with each memory location. With the support for in-place metadata we can implement and compare the state-of-the-art multi-version algorithms, both between themselves and with single-version algorithms.

To support this fact, we implemented two state-of-the-art multi-version algorithms: SMV [PBLK11] and JVSTM-LockFree [FC11]. These algorithms are significantly different, although both are MV-*permissive* [PFK10]. They differ on the progress guarantees, e.g., JVSTM-LockFree implements a commit algorithm that is lock-free, while SMV uses write-set locking, and also differ on the technique used to garbage collect unnecessary versions, where JVSTM-LockFree uses a custom parallel garbage collector, while SMV resorts to the JVM garbage collector by using weak-references.

We also implemented a new multi-version algorithm, based in TL2 (referred to as `mvstm`), which has a bounded number of versions for each memory location and, at commit time, it locks each memory location of the write-set to preform the write-back tentative values. This algorithm is not MV-*permissive* as read-only transactions may abort due to an unavailable version or even because the respective memory location is locked by other transaction that is committing.

In the following sections we describe the implementation details of each of the above algorithms.

5.5.1 SMV – Selective Multi-versioning STM

The SMV algorithm described in [PBLK11] is an MV-*permissive* multi-version algorithm, which uses the JVM garbage collector to automatically collect unreachable versions. The implementation of this algorithm in our extension of Deuce was based on the original source code released by the authors[5]. The original algorithm is object-based, opposite to Deuce, and our extension, which only supports word-based STMs, and hence we adapted the SMV algorithm to work as a word-base STM.

The transactional metadata required by SMV can be depicted in Figure 5.15. This is a direct adaptation of the `SMVAdapterLight` class provided by the original source code. Also, we used the same source code that implements the behavior of read- and update-transactions with minimal changes. We did this by implementing our extension's interface `ContextMetadata` as an

[5] `http://tx.technion.ac.il/~dima39/sourcecode/SMVLib-29-06-11.zip`

```
public class SMVObjAdapter extends TxField {
  public volatile Object latest;
  public int creatorTxnId;
  public final AtomicInteger version = new AtomicInteger(1);
  public volatile WeakReference<VersionHolder> prev =
        new WeakReference<VersionHolder>(null);

  // ... public methods
}
```

Figure 5.15: SMV transactional metadata class.

```
public class VBoxAdapter extends TxField {
  protected VBox<Object> vbox;

  // ... public methods
}
```

Figure 5.16: JVSTM-LockFree transactional metadata class.

adapter of the original source code, each transactional operation (read, write, commit, abort) is forward to the original implementation.

The change from an object-based to a word-based approach only required minimal changes on the read and write procedures. In the case of a read operation, instead of returning an object, is returned a field's value. And in the case of a write operation, instead of cloning the object to be written and storing in the transaction's write-set, the tentative value of a field is stored in the write-set.

The overall adaptation of the original source code to our framework was very easy and fast, which proves the flexibility of our support for implementing different STM algorithms.

5.5.2 JVSTM Lock Free

The JVSTM-LockFree [FC11] is an adaptation of the original JVSTM algorithm [CRS06], which enhances the commit procedure using a lock-free algorithm, instead of using a global lock, and also improves the garbage collector algorithm by the use of a parallel collecting approach. Once again, we based our implementation in the original source code[6].

We created a metadata object containing a reference to a vbox, as implemented originally by the JVSTM-LockFree algorithm. We show the object metadata implementation in Figure 5.16.

The context class was implemented as an adapter to the original implementation of the read-only and update transactions. Actually, we used the JVSTM-LockFree implementation as an external library (JAR file), and the Deuce context class only forwards the transactional calls to the external library. This approach was possible because there was no need to make any changes to the JVSTM-LockFree algorithm, for it to work in our framework extension.

[6] https://github.com/inesc-id-esw/jvstm

5.5.3 MVSTM – A New Multi-Version Algorithm

We developed and implemented a new multi-version algorithm (MVSTM) using the in-place metadata support and inspired in TL2. It defines a maximum size for the list of versions, imposing a bound in the number of versions for each memory location. At commit time, MVSTM uses a lock per memory location listed in the write-set.

The structure for each version is the same as in JVSTM. Each version is composed by a timestamp, which corresponds to the timestamp of the transaction that committed the version, and the data value. Each metadata object has a pointer to the head of a version list with a fixed size. Whenever a transaction commits a new version, and the maximum size of the version list is reached, we discard half of the older versions. This decision allows to limit the memory used by the algorithm and avoid complex garbage collection algorithms to remove old versions. The drawback of this approach is that read-only operations can now abort because they may try to read a version that was already removed. Moreover, read-only transactions will also abort when trying to read an object's field that is being currently updated by a concurrent commit operation. Thus, this multi-version STM algorithm is not MV-*permissive*.

The commit operation is similar to the TL2 algorithm. Read-only transactions may commit without any additional validation procedure, whilst read-write transactions need to lock the write-set entries and then validate their read-set. In the case of a successful validation of the read-set, the transaction applies the write-set by creating a new version for each entry in the write-set, and finally unlocks the write-set locks. This locking scheme allows two transactions to commit concurrently if their write-sets are disjoint.

Although the algorithm does not guarantee MV-*permissiveness*, it has a very simple implementation which may benefit the performance of short and medium sized transactions, and reduce the abort rate when compared to other algorithms such as TL2.

MVSTM-SI – Snapshot Isolation Version The efficient support for multi-version algorithms introduced by our extension to Deuce framework allows to efficiently implement snapshot isolation based STM algorithms. We decided to implement a snapshot isolation algorithm based on the implementation of the MVSTM algorithm. The main benefits of using snapshot isolation in a transactional memory implementation is that we do not need to track any read accesses, i.e., we do not need to store a read-set nor to verify the read-set validity at commit time.

The implementation of the MVSTM-SI algorithm required a minimal set of changes to the implementation of MVSTM algorithm. The MVSTM algorithm was changed to keep only the write-set, and at commit time, instead of validating the read-set, it now validates the write-set to check for write-write conflicts. Moreover, as in snapshot isolation transactions must always read from the snapshot valid at the start of the transaction, read-write transactions always read from the version list, as the read-only transactions.

5.6 Supporting the Weak Atomicity Model

Multi-version algorithms read and write the data values from and into the list of versions. This implies that all accesses to fields in shared objects must be done inside a memory transaction, and thus multi-version algorithms require a *strong atomicity* model [BLM05].

Deuce does not provide a strong atomicity model as memory accesses done outside of transactions are not instrumented, and hence it is possible to have non-transactional accesses to fields of objects that were also accessed inside memory transactions. This hinders the usage of multi-version algorithms in Deuce. One approach to address this problem is to rewrite the existing benchmarks to wrap all accesses to shared objects inside an atomic method, but such code changes are always a cumbersome and error prone process. We addressed this problem by adapting the multi-version algorithms to support the *weak atomicity* model.

When using a weak atomicity model with a multi-version scheme, updates made by non-transactional code to object fields are not seen by transactional code and, on the other way around, updates made by transactional code are not seen by non-transactional code. The key idea for our solution is to store the value of the latest version in the object's field instead of in the node at the head of the version list. When a transaction needs to read a field of an object, it requests the version corresponding to the transaction timestamp. If it receives the head version, then it reads the value directly from the object's field, otherwise it reads the value from the version node.

The problem with this approach is how to guarantee the atomicity of the commit of a new version, because now we have two steps: adding a new version node to the head of the list and updating the field's value. These two steps must be atomic with respect to the other concurrent transactions. Our solution is to create a temporary new version with an infinite timestamp, making it unreachable for other concurrent transactions, until we update the value and then change the timestamp to its proper value.

The algorithmic adaptation that we propose is not intended to support a workload of intertwined non-transactional and transactional accesses, but rather a phased workload where non-transactional code does not execute concurrently with transactional code. Many of the transactional benchmarks we used exhibit such a phased workload, because the data structures are initialized in the program startup using non-transactional code. After this initialization, the transactional code can now operate over the data previously installed by non-transactional code. After the transactional processing, non-transactional code may also post-process the data, such as in a case of a validation procedure.

5.6.1 Read Access Adaptation

In a multi-version scheme, read-only transactions always search for a correct version to return its value. Each version container holds the timestamp (or version number) and the respective value. When the transaction finds the correct version, it returns the value contained in the version.

To support non-transactional accesses mixed with a multi-version scheme, the latest value of an object's field is stored in-place, and therefore the head version might not have the correct value because of a previous non-transactional update. The read procedure of a multi-version transaction must be adapted to reflect the new location of the latest value. When a transaction queries for a version, and receives the head version, corresponding to the latest value, it has to return the value directly from the object's field. The pseudo-code of this adaptation is presented below, where the additional operations are denoted in underline.

1. $\underline{val := \mathsf{read}()}$

2. $ver := \mathsf{find\_version}()$

3. return $\begin{cases} \underline{val} & \underline{\text{if is\_head\_version}(ver)} \\ ver.val & \text{otherwise} \end{cases}$

The read() function returns the value from the object's field, the find_version function retrieves the corresponding version according to the transaction timestamp, and the is_head_version function asserts if version ver is the head version. This small change introduces the additional shared memory access performed in step 1. The correctness of this adaptation can only be assessed with the explanation of the commit adaptation, which guarantees that whenever the is_head_version function returns true the value val is correct.

5.6.2 Commit Adaptation

The commit operation is typical composed by a validation phase and write-back phase. In the write-back phase, for each new value present in the write-set, a new version is created and is stored as the head version. The write-back phase must be atomic, and this can be achieved using a global lock (JVSTM), a write-set entry locking (SMV, MVSTM), or even a lock-free algorithm (JVSTM-LockFree).

Our adaptation only makes changes to the write-back phase. In each iteration of the write-back phase, a new version is installed as the head version of the version list associated with the object's field being written. The version contains the commit timestamp, which defines the commit ordering, and the new value. Additionally, to support the weak-atomicity model, we also need to write the new value directly to the object's field. The problem that arises with this additional operation is that concurrent transactions need to see the update on the version list, and the update of the object's value as a single operation. The key idea to solve this problem is to create a version with a temporary infinite timestamp, which will prevent concurrent transactions from accessing the head version, and consequently the object's field value.

Below we present the pseudo-code of the adaptation to the commit of a new version, where t_c is the timestamp of the transaction that is performing the commit, t_∞ is the highest timestamp, val is the value to be written, and ver_h is the pointer to the head version. For the sake of simplicity, we assume that these steps execute in mutual exclusion with respect to other concurrent commits (in Section 5.6.3.3 we explain how to apply these steps to a lock-free context as in the JVSTM-LockFree algorithm).

1. $\underline{ver_h.value := \text{read}()}$
2. $ver_n := \text{create\_version}(new\_val, \underline{t_\infty}, ver_h)$
3. $ver_h := ver_n$
4. $\underline{\text{write}(new\_val)}$
5. $\underline{ver_h.timestamp := t_c}$

Once again, the additional changes are denoted in underline. The first step is to update the value of the head version with the current value of the object's field. This update is safe because until this point transactions that retrieve the head version read the value directly from the object's field, as described in the previous section. Then we create a new version with an infinite timestamp and the new value to be written in the object's field, and the pointer to the current head version. In the third step, we make the new version ver_n the current head version and it becomes

visible to all concurrent transactions. This version will never be accessed by any concurrent transaction because of the infinite timestamp. Then we can safely update the object's field value in the fourth step because no concurrent transaction gets the head version (the head version still has an infinite timestamp up to this point). In the last step we change the timestamp of the current head version to its proper value making accessible to concurrent transactions.

The adaptation of the commit operation introduces three new shared memory accesses, where two of them are write accesses. Thus, this adaptation is expected to slightly lower the throughput of the multi-version algorithm. We applied this adaptation to the multi-version algorithms that we described previously, and compared the performance of both versions of each. In the next section we report the experience of adapting each algorithm.

5.6.3 MV-Algorithms Adaptation

We use the algorithmic changes described in the previous section to adapt the four multi-version algorithms under study (JVSTM, SMV, JVSTM-LockFree and MVSTM), enabling the execution of all benchmarks available in the Deuce framework with no modification. In this section, we present the adaptation details as well as the performance comparison with the original algorithm.

To evaluate the original STM algorithms (without the weak-atomicity adaptation) we had to modify some benchmarks so that all accesses to shared memory are done inside of memory transactions. These modifications include mainly wrapping the initialization of data structures, and verification procedures, inside of memory transactions. We modified the Linked List, Red-Black Tree, and Skip List micro-benchmarks, and also the STMBench7 macro-benchmark. In the case of the STMBench7 we had to disable the invariant checks because otherwise it would take hours to perform the checks. No modifications to the benchmarks were necessary when testing the algorithms adapted to support weak-atomicity. When executing the non-modified version of STMBench7 with the adapted versions of the STM algorithm, the invariant checks take less than one minute to execute.

5.6.3.1 JVSTM and MVSTM

The JVSTM and MVSTM algorithms preform the commit operation in mutual exclusion with other concurrent committing transactions. The adaptation of these algorithms to support a weak-atomicity model is straightforward. The changes that we presented in the previous section to modify the read and commit operation can be applied directly to both implementations. Moreover, the Deuce framework already provides the memory value when a read access is issued (see Figure 5.4 in page 100), which simplifies the first step of the read procedure described in Section 5.6.1.

Figures 5.17 and 5.18 depict the performance comparison between the original and adapted versions of JVSTM and MVSTM respectively. The comparison is done by showing the relative performance of the adapted version over the original version.

Both adapted versions of JVSTM and MVSTM show a performance very similar to the original versions. Sometimes, the adapted version can even outperform the original version. This is due to the specificity of the Deuce framework that already provides the memory value for each read access callback. In the case of the adapted version, most of the times that value is used, opposed to the original version where the value is always obtained by dereferencing a version container.

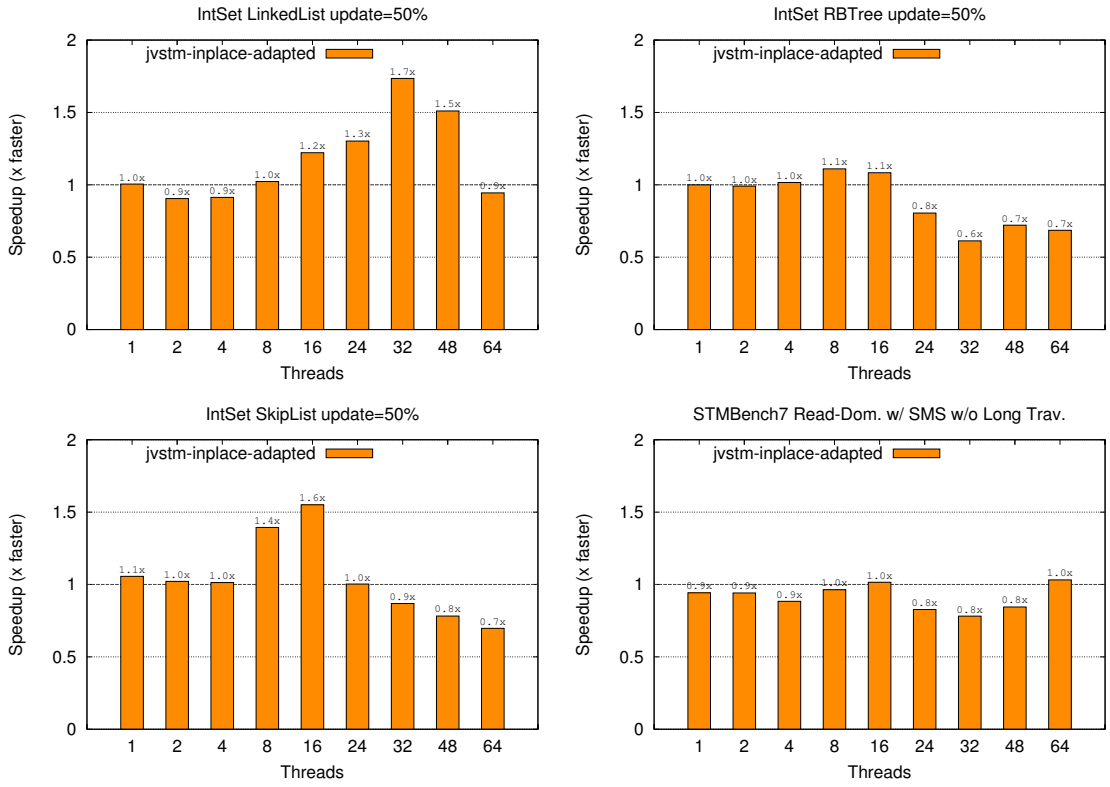


Figure 5.17: Performance comparison between original JVSTM and adapted JVSTM.

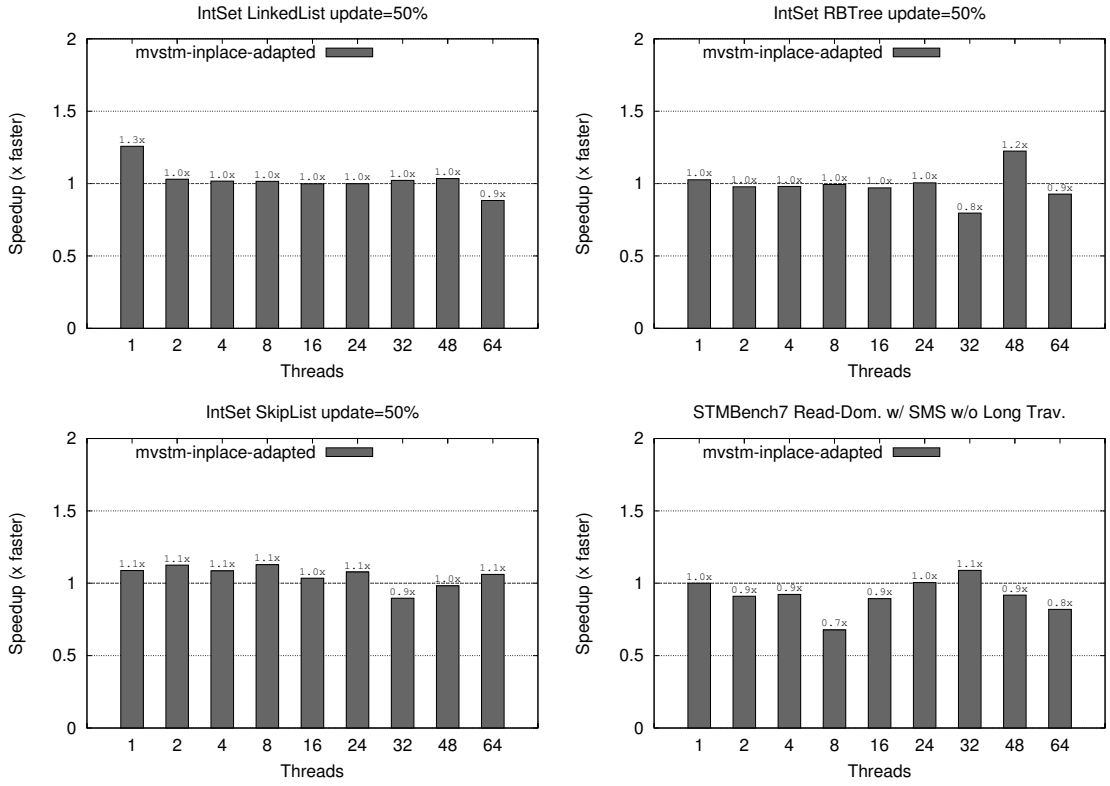


Figure 5.18: Performance comparison between original MVSTM and adapted MVSTM.

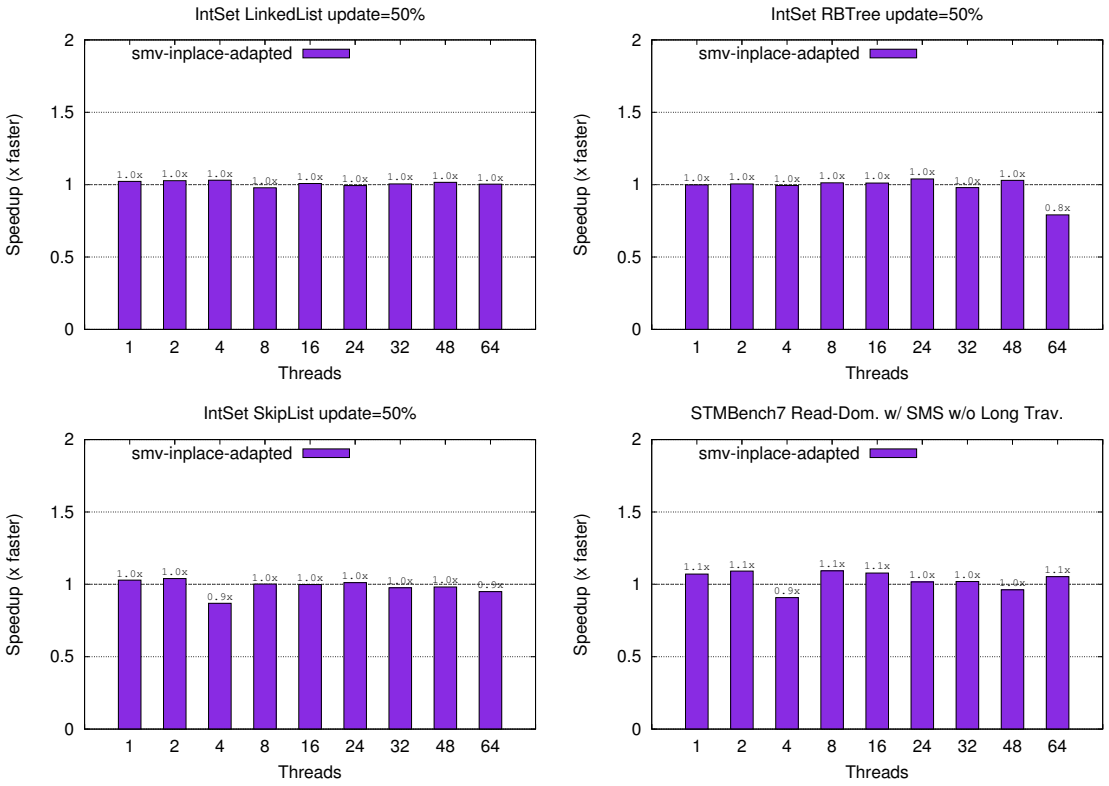


Figure 5.19: Performance comparison between original SMV and adapted SMV.

5.6.3.2 SMV

The SMV algorithm defines a different memory layout for the version list. In SMV, the value of the latest version is stored outside of the version list, which reassembles our adaptation proposal of storing the latest value directly on the memory location. To apply the support for a weak-atomicity model, we simply moved the value of the latest version from an auxiliary variable (used in SMV original implementation) directly to the associated memory location.

This modification has consequences in the commit operation, which must also be adapted to atomically update the latest version information and the memory location value. The first step in the SMV commit operation is to move the latest value and timestamp to a newly created version container and add it to the head of the version list. We change this step by using the latest value stored in memory. In the last step of the SMV commit operation the variable containing the latest value is updated with the new tentative value. We changed this step by writing the tentative value directly to memory.

The changes made to the SMV algorithm are minimal and thus we expect that the performance differences between the two versions to be also minimal. The results depicted in Figure 5.19 confirm our expectations, showing minimal differences between the original version and adapted version.

5.6.3.3 JVSTM-LockFree

The JVSTM-LockFree implements a lock free commit operation. The assumption to apply the adaptation for the commit procedure, presented in Section 5.6.2, is that the commit should be

```
1 public void commit(Object newValue, int txNumber) {
2   Version currHead = this.head;
3   Version existingVersion = currHead.getVersion(txNumber);
4
5   if (existingBody.version < txNumber) {
6     Version newVer = new Version(newValue, txNumber, currHead);
7     compare_and_swap(this.head, currHead, newVer);
8   }
9 }
```

Figure 5.20: JVSTM-LockFree original commit operation.

done in mutual exclusion. This assumption is true for the previous algorithms but not for the JVSTM-LockFree. In this algorithm, the commit of a single version can be done by more than one thread at the same time by resorting to atomic primitives such as compare-and-swap.

The adaptation of the read procedure is straightforward as in the JVSTM algorithm. The adaptation of the commit procedure is rather complex and requires additional atomic operations to ensure the correctness of the algorithm. Figure 5.20 depicts a simplified version of the original commit. The method `commit` preforms a compare-and-swap to install the new version. Other threads may be executing the same method for the same vbox, but only one of them will install the new version. Further details on how the JVSTM-LockFree commit algorithm works can be found in [FC11].

Figure 5.21 depicts the adapted version of the JVSTM-LockFree commit algorithm to support a weak-atomicity model. The algorithm has roughly three times more operations than the original version. We explain this adapted version by describing how each step of the adaptation described in Section 5.6.2 related to the code listed in the Figure.

The first step $ver_h.value := \mathsf{read}()$ is preformed by lines 5 and 7-9. The update of the head version's value (line 8) is done inside a conditional statement because other concurrent thread may had already preformed the same update. The creation of a new version in the second step $ver_n := \mathsf{create\_version}(new\_val, t_\infty, ver_h)$ is preformed in line 10. The publication of the new version in the third step $ver_h := ver_n$ is preformed in lines 11-19. In this step we preform a compare-and-swap, as in the original algorithm, to publicize the new version, but if other concurrent thread already publicize the new version, then we need to get a pointer to the new version. This is done in lines 14 to 18. Using this pointer we can preform the final fourth and fifth steps $\mathsf{write}(new\_val)$ and $ver_h.timestamp := t_c$, which are done in lines 20-23. The writing of the new value directly to memory (line 21) is done using a compare-and-swap atomic operation to prevent lost updates. The update of the version number (line 23) is safe because we always have a pointer to the correct version container. These last two steps are also preformed in lines 28-31, in the case when a thread attempting to commit finds out, in line 6, that other concurrent thread already publicized the new version, and therefore it helps finishing the commit. Another source of overhead is caused by a limitation of the compare-and-swap operation, which can only be preformed for reference and integer types. Thus, for other primitive type such as **`float`**, or **`byte`**, the compare-and-swap operations preformed in lines 21 and 29, must be substituted by some mutual exclusion block. Fortunately the use of compare-and-swap non-supported types in the benchmarks is rare.

The introduced complexity in the commit algorithm will impose a strong performance penalty in workloads that generate a high rate of commits, typical in small-sized transactions, and also in transactions that generate large write-sets. Figure 5.22 presents the results of comparing the

```
public void commit(Object newValue, int txNumber) {
  Version currHead = this.head;
  Version existingVersion = currHead.getVersion(txNumber);

  Object latest = read(memory_location);
  if (existingVersion == currHead && existingVersion.version < txNumber) {
    if (this.head == existingVersion) {
      currHead.value = latest;
    }
    Version newVer = new Version(newValue, Integer.MAX_VALUE, currHead);
    if (compare_and_swap(this.head, currHead, newVer)) {
      existingVersion = newVer
    } else {
      existingVersion = this.head;
      Version tmpVer = existingVersion.getVersion(txNumber);
      if (tmpVer.version == txNumber) {
          existingVersion = tmpVer;
      }
    }
    if (existingVersion.version == Integer.MAX_VALUE) {
      compare_and_swap(memory_location, latest, newValue);
    }
    existingVersion.version = txNumber;
  }
  else {
    if (existingVersion.version < txNumber) {
      existingVersion = currHead;
      if (existingVersion.version == Integer.MAX_VALUE) {
        compare_and_swap(memory_location, latest, newValue);
      }
      existingVersion.version = txNumber;
    }
  }
}
```

Figure 5.21: JVSTM-LockFree adapted commit operation.

adapted version over the original version of JVSTM-LockFree.

In the case of the LinkedList micro-benchmark, the transactions generate small write-sets (the add and remove operations only write to a single object), and typically the commit rate is low due to the long duration of the lookup of a node, which is linear with the size of the list. As so, the adapted version outperforms the original version, due to the read accesses that use value directly from memory and are immediately provided by the Deuce framework.

In the case of the SkipList and RBTree micro-benchmarks, the adapted commit overhead is more notorious when the contention increases with the number of threads. These benchmarks generate a high rate of commit operations, although still with small write-sets per transaction.

In the STMBench7 benchmark, known to generate very large read- and write-sets, the adapted version can only achieve half the performance of the original version. The results confirm our performance expectations, and also confirm that the overhead introduced by adapting a multi-version algorithm to support a weak-atomicity model is almost nil for algorithms that preform the commit of versions in mutual exclusion, and has a considerable cost otherwise.

5.7 Performance Comparison of STM Algorithms

In this chapter we presented an extension of the Deuce framework to support the efficient implementation of STM algorithms that require a one-to-one relation between memory locations and

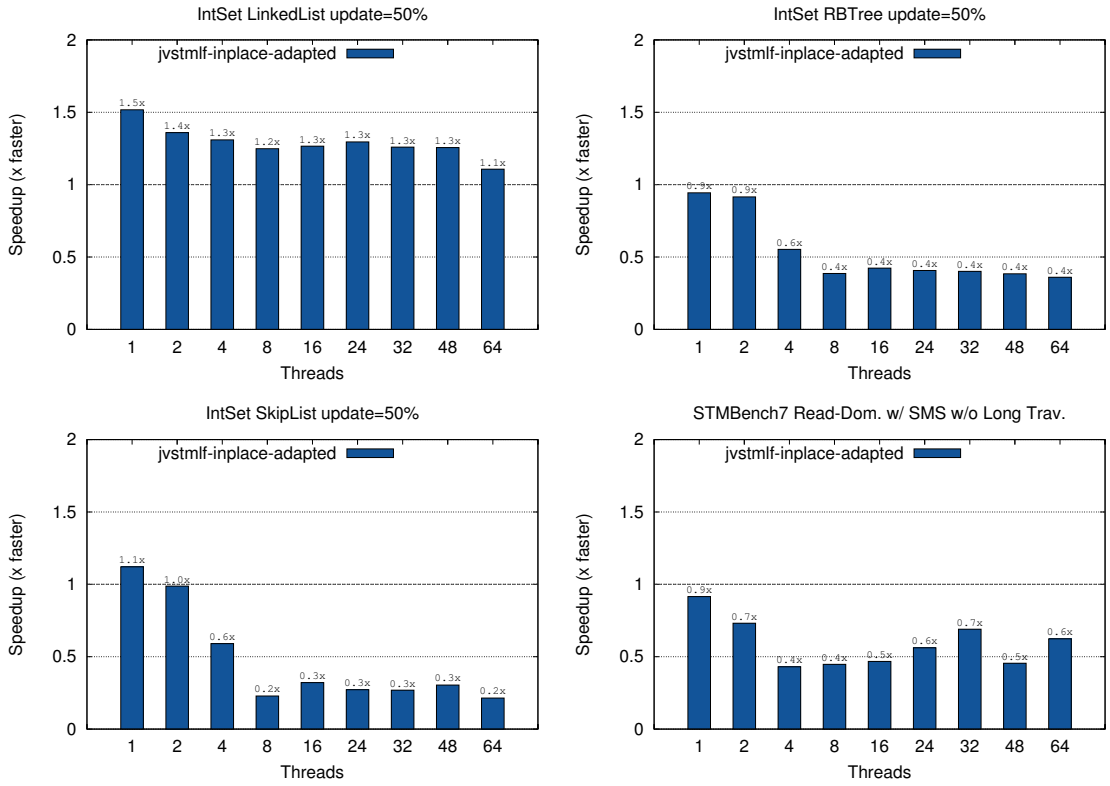


Figure 5.22: Performance comparison between original JVSTM-LockFree and adapted JVSTM-LockFree.

transactional metadata, being multi-version algorithms an instance of this class of algorithms. We evaluated the extension considering the implications in both performance and memory consumption. The results were very satisfactory and thus we implemented two state-of-the-art multi-version algorithms (SMV and JVSTM-LockFree), and implemented a new multi-version algorithm (MVSTM) with a very simple design.

Given this support for very different classes of STM algorithms, we may now aiming at a fair comparison of their performance, i.e., compare the algorithms implemented in the same framework and with the same benchmarks. In this section we show the direct comparison between several out-place and in-place STM algorithms. The list of STM algorithms chosen for comparison are TL2, JVSTM, JVSTM-LockFree, SMV, and MVSTM. In the case of TL2 we use two versions: the out-place version (TL2-Outplace) which is distributed with Deuce, and an in-place version (TL2-Inplace) which we implemented in our extension. The in-place version moves the locks from the external lock table to the transactional metadata, and completely avoids the false-sharing on locks.

In the case of multi-version algorithms our measurements were conducted under two settings. The first setup consisted on executing the (unmodified) benchmarks combined with the weak-atomicity-adapted multi-version algorithms. In the second setup, we executed a modified version of the micro-benchmarks and STMBench7 combined with the original multi-version algorithms that do not support weak-atomicity. In the comparison results, we will only use the best of the results of the original and the adapted versions of each multi-version algorithm.

We also compare the snapshot isolation algorithm MVSTM-SI against other *opaque* algorithms

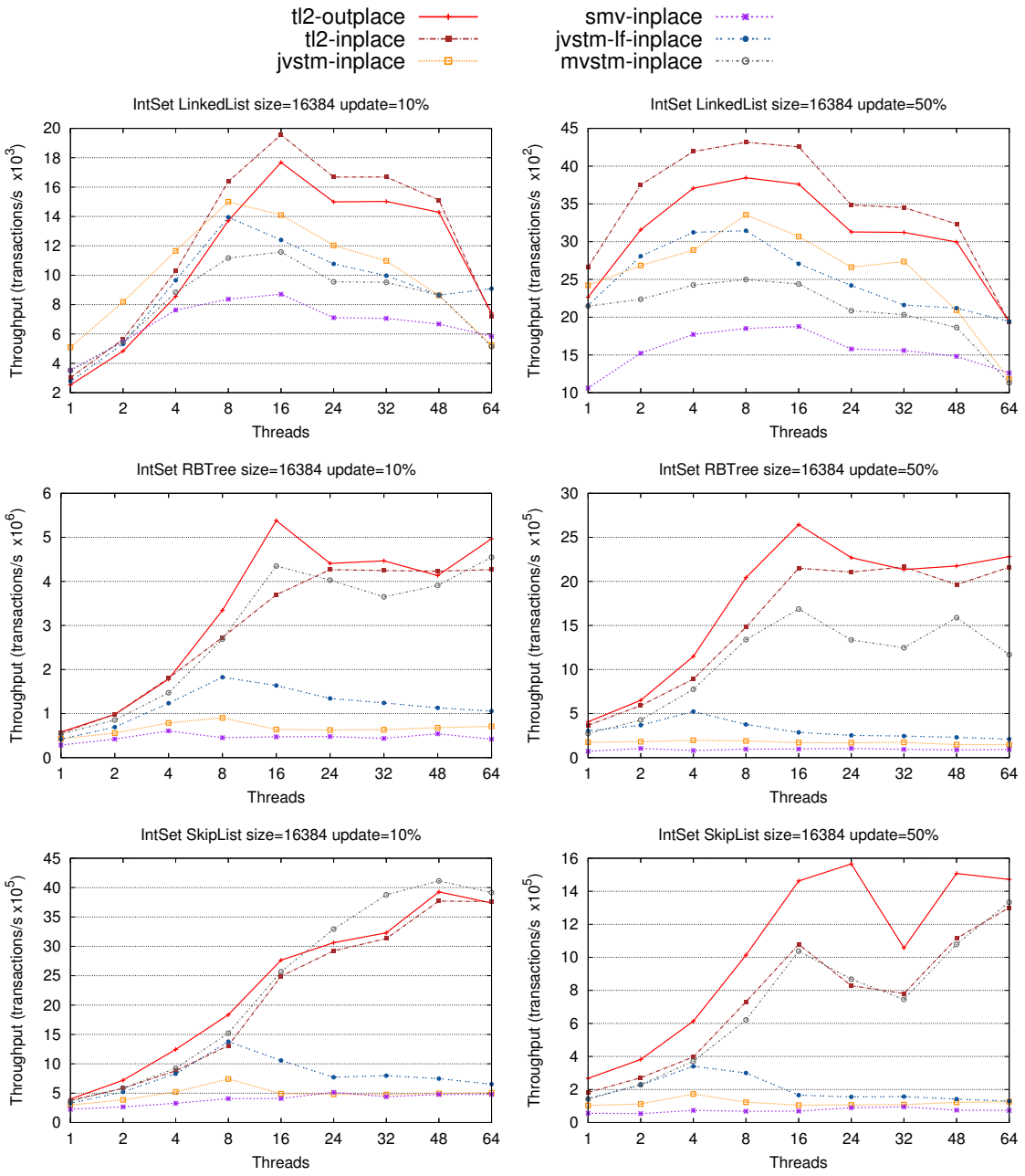


Figure 5.23: Micro-benchmarks comparison.

whenever we know that the benchmarks execute safely under snapshot isolation, which is the case of the Linked List and Skip List benchmarks.

As in the extension evaluation, the benchmarks were executed on a computer with four AMD Opteron 6272 16-Core processors @2.1 GHz with 8×2 MB of L2 cache, 16 MB of L3 cache, and 64 GB of RAM, running Debian Linux 3.2.41 x86_64, and Java 1.7.0_21.

Figure 5.23 shows the results of the execution of the micro-benchmarks Linked List, Red-Black Tree, and Skip List. The Linked List benchmark is characterized by transactions with large read-sets and by a high abort rate. In this benchmark the algorithms do not scale well with the increase in the number of threads. The single-version algorithms TL2-Outplace and TL2-Inplace exhibit better performance. These algorithms have very efficient implementations and the read

accesses are very lightweight. Additionally, in the case of read-only transactions, each read access is checked for consistency but the transaction can safely commit without further verification. To support multiple versions per memory location, the multi-version algorithms add a high number of extra computations when reading a value from a memory location, with the benefit of avoiding spurious transaction aborts and hence avoid the re-execution of transactions. Although, in the micro-benchmarks this possible benefit is not observed.

In the Red-Black Tree and Skip List benchmarks, transactions are very small and fast, and have a low conflict probability, except in the Red-Black Tree when tree rotations are preformed. These benchmarks hide even more the advantages of multi-version algorithms when compared with single-version algorithms. A surprising result of these benchmarks is the performance achieved by the MVSTM algorithm which can compete with the TL2 versions. The MVSTM algorithm has a very lightweight implementation trading permissiveness properties by performance, which works well in these kinds of workloads.

Another unexpected result is the poor performance of SMV algorithm when compared with other multi-version algorithms. We investigated the causes for this behavior, and the problem resides on the mechanism for garbage collection of unnecessary versions. SMV implements a mechanism for storing the list of versions using Java weak-references that allows the JVM garbage collection to collect the unnecessary versions, instead of using an additional component to preform this version cleaning. While in theory this appears to be an efficient design choice, in practice it does not work as expected. In the micro-benchmarks where the workload generates millions of transactions per second, the read-write transactions are also creating a very large number of versions per second, and since SMV uses weak-references to store versions, the JVM garbage collector has trouble to keep up the cleaning of so many versions. What happens in reality is that during the benchmark execution the garbage collector is always working and is hindering the real performance of the SMV algorithm, and also it consumes more memory than other multi-version algorithms.

The comparison results for the STAMP benchmarking suite are depicted in Figure 5.24. In these results the y-axis represents execution time and therefore lower values are better. The benchmarks in this suite exhibit very different workloads, some of them even generate such high contention that hinders the scaling for all of the tested algorithms. The benchmarks KMeans, Genome, and Intruder, exposes the corner cases of the *adapted* JVSTM-LockFree algorithm, which must execute some updates to the memory location inside of a mutual exclusion block as described in Section 5.6.3.3, and hence its performance is strongly penalized. We believe that the original JVSTM-LockFree algorithm would perform much better than the adapted version in these particular benchmarks.

The TL2 based algorithms overall exhibit a very good performance, as well as MVSTM, which in most cases can compete with the TL2 algorithms. In the Labyrinth benchmark the multi-version algorithm JVSTM-LockFree presents a very good result. This algorithm has a low abort rate when compared with the other algorithms, which allows it to not waist so much work in transaction restarts. In the SSCA2 benchmark all the in-place algorithms suffer from the high overhead of transactional metadata management shown in Figure 5.10 of Section 5.4.1.

In Figure 5.25 we show the results for the STMBench7 benchmarks. This benchmark generate CPU-intensive transactions with large read-sets and write-sets. This benchmarks allows to exploit the benefits of multi-version algorithms which can avoid spurious aborts and thus achieve better performance than single-version algorithms. The JVSTM-Lockfree algorithm achieves a good

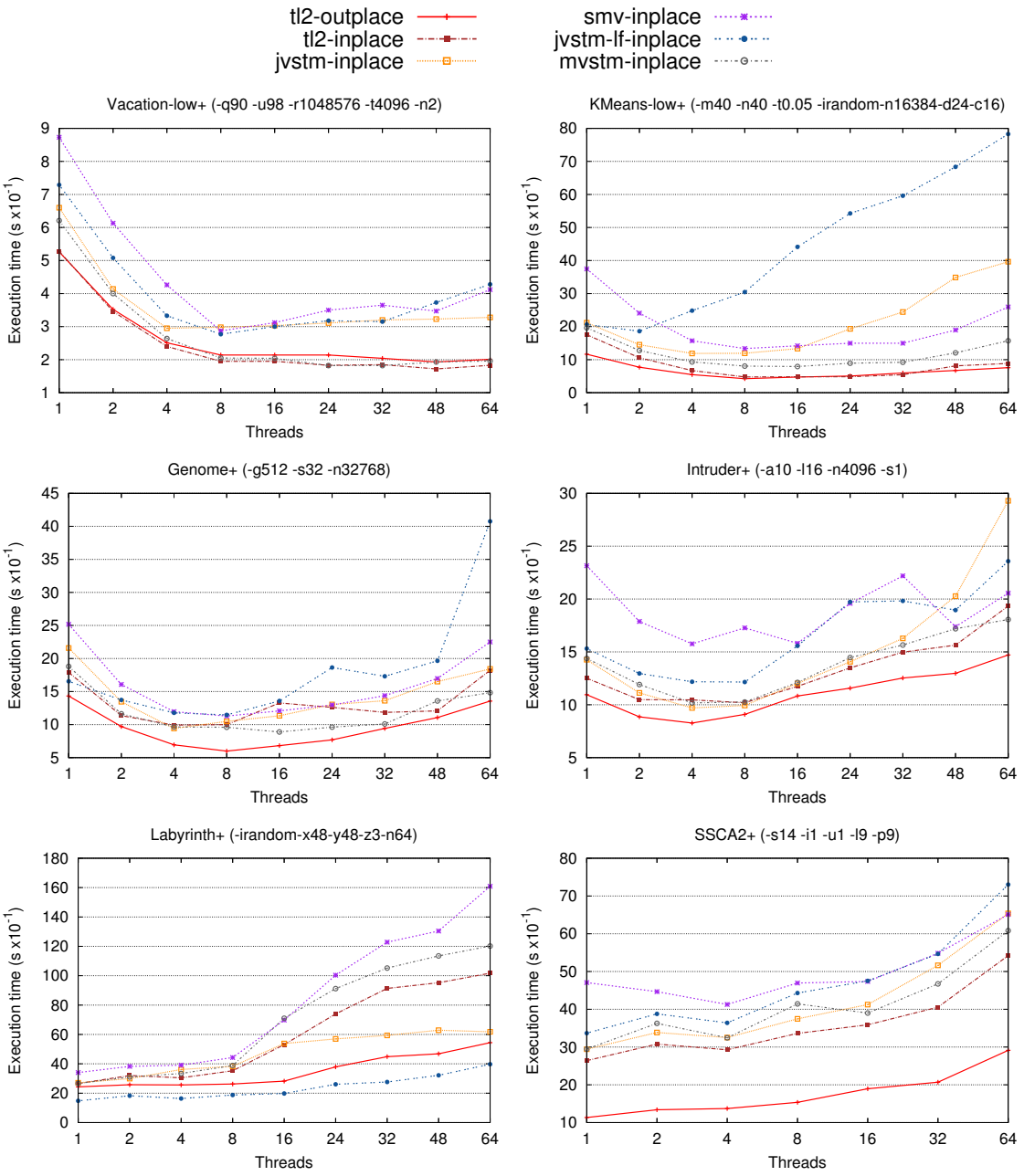


Figure 5.24: STAMP benchmarks comparison.

performance, higher than the remaining algorithms, confirming the advantages of using an MV-*permissive* algorithm in this kind of workload.

In this benchmark, there is a significant performance difference between the out-place and in-place versions of TL2 algorithm. The out-place version does not even scale with the number of threads. The reason of this behavior may be due to cache locality issues. The in-place version is much more cache-friendly than the out-place version. The in-place version has a high probability of having the metadata in the same cache line as the memory location. This does not happen in the out-place version, and in the special case of STMBench7, where transactions perform a large number of reads and writes, the out-place version must read many entries from the external lock table, which may not fit in the cache and requiring much more page transfers from main memory

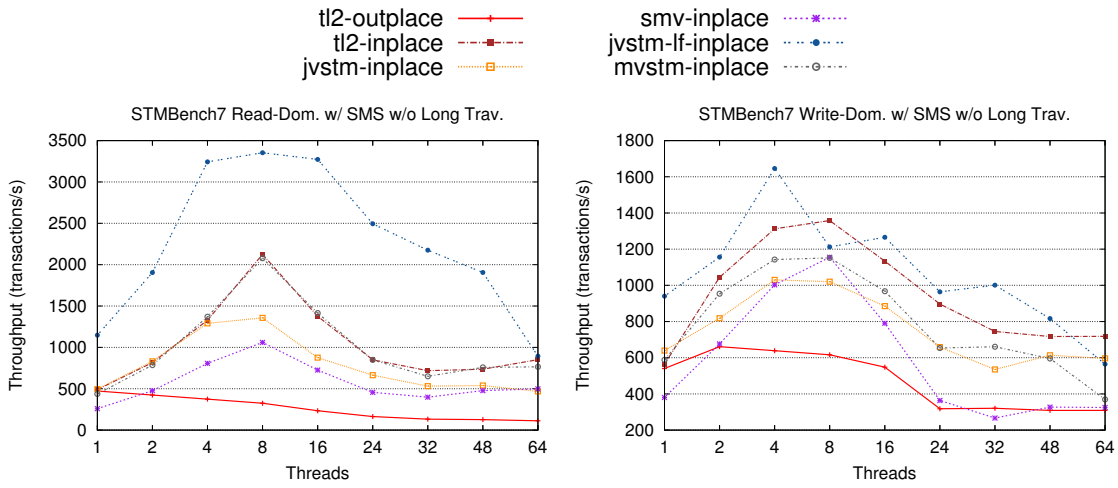


Figure 5.25: STMBench7 comparison.

to the cache.

In the write-dominated workload of STMBench7, all algorithms have similar performance with the exception of TL2-Outplace. Although almost all transactions are read-write, the multi-version algorithms can still compete with the single-version TL2-Inplace algorithm, and JVSTM-LockFree almost always exhibit the best performance.

We evaluated the snapshot isolation algorithm MVSTM-SI with the two benchmarks known to be safe under snapshot isolation, and the results are depicted in Figure 5.26. In the Linked List benchmark, opposed to all opaque algorithms, the snapshot isolation based algorithm can scale with the number of threads. The Linked List benchmark is the extreme case where the avoidance of read-write conflicts is more effective. For the Skip List benchmark, the MVSTM-SI algorithm preforms similarly with its dual opaque version. In this benchmark, the opaque versions have low abort rates, and therefore, the snapshot isolation based algorithm does not have space to perform better than the other algorithms. Also, it is important to note that the MVSTM-SI algorithm does not preform worse than the opaque version, which can induce us to believe that using snapshot isolation will always benefit performance and never degrade it.

5.8 Concluding Remarks

To the best of our knowledge, the extension of Deuce as described in this chapter creates the first Java STM framework providing a performance-wise balanced support of both in-place and out-place strategies. This is achieved by a transformation process of the program bytecode that adds new metadata objects for each class field, and that includes a customized solution for N-dimensional arrays that is fully backwards compatible with primitive type arrays.

We evaluated our system by measuring the overhead introduced by our new in-place strategy with respect to the original Deuce implementation. Although we can observe a light slowdown in our new implementation of arrays, we would like to reinforce that our solution has no limitations whatsoever concerning the type of the array elements, the number of its dimensions, fits equally to algorithms biased towards in-place or out-place strategies, and all bytecode transformations are done automatically requiring no changes to the source code.

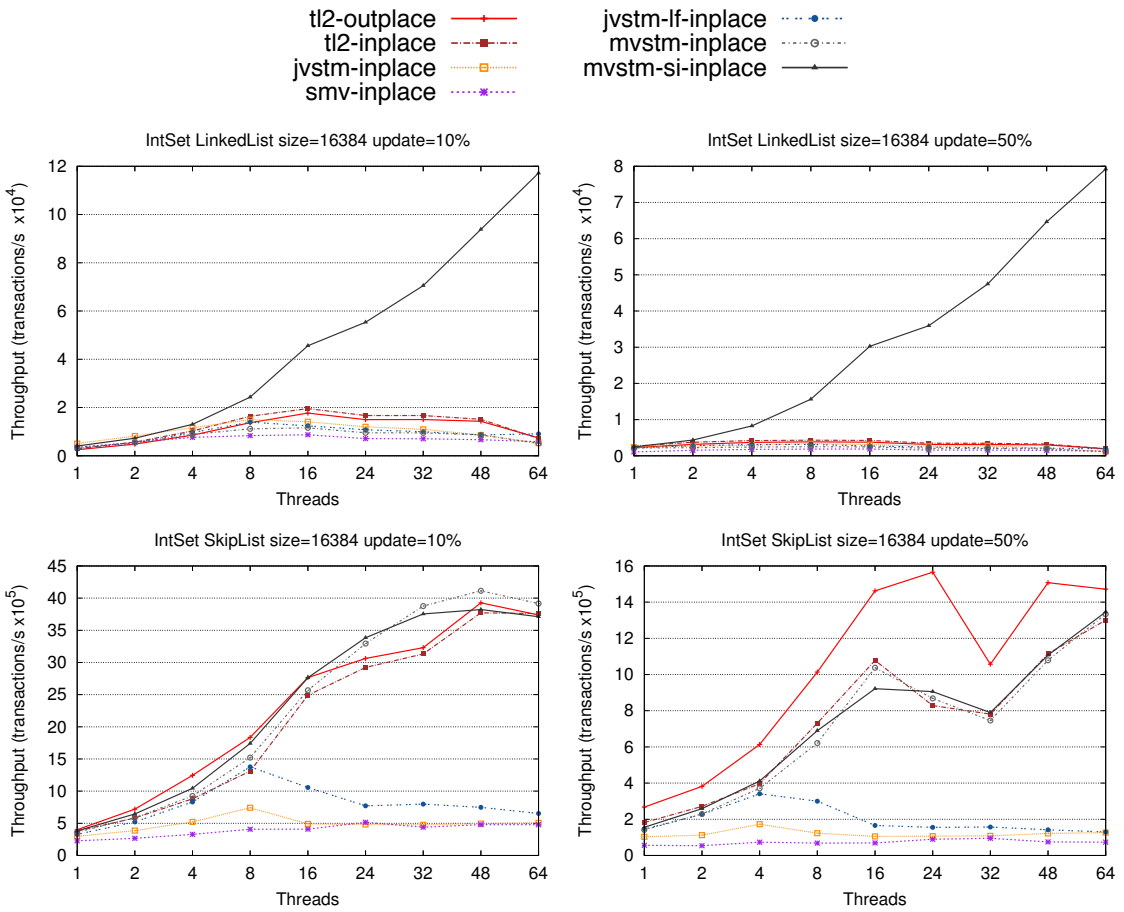


Figure 5.26: Snapshot Isolation algorithms comparison.

We also evaluated the effectiveness of the new in-place interface by comparing the performance of a well known multi-version STM algorithm implemented using the original out-place interface, resorting to an external mapping table, and implemented using the in-place interface. The results show that, by using the in-place strategy, multi-version algorithms can now be fairly compared with other STM algorithms such as TL2, which was not possible when using the original Deuce framework.

Using this new infrastructure we implemented two state-of-the-art multi-version algorithms SMV and JVSTM-LockFree and made the first performance comparison between the two. Also, we were able to efficiently implement snapshot isolation based algorithms on top of existing multi-version algorithms.

Finally, we proposed an algorithmic adaptation for multi-version algorithms to support the weak-atomicity model as provided in the Deuce framework. We reported the experience of adapting several state-of-the-art multi-version algorithms and evaluate their performance. In general, multi-version algorithms can be adapted to support the weak-atomicity model without a performance penalty, except the case of the algorithms that implement a lock-free commit operation.

Publications The contents of this chapter were partially published in:

- [DVL12] **Efficient support for in-place metadata in transactional memory**. Ricardo J. Dias, Tiago M. Vale, and João M. Lourenço. In proceedings of Euro-Par 2012, August 2012.
- [DVL13] **Efficient support for in-place metadata in java software transactional memory**. Ricardo J. Dias, Tiago M. Vale, and João M. Lourenço. Concurrency and Computation: Practice and Experience, 2013.

6

Conclusions and Future Work

Although optimization techniques such as structuring a program using finer-grain transactions or using relaxed isolation runtimes have the potential to increase the parallelism of transactional memory programs, these techniques also introduce serious correctness problems, which may hinder the application functionality and manifest themselves as incorrect results or runtime errors.

To prove our thesis, presented in the first chapter of this dissertation, we developed two static analysis techniques that allow to maintain the correctness of transactional memory programs despite the employment of finer-grain transactions, and a transactional memory runtime based on snapshot isolation to increase parallelism.

In particular we proposed a scalable and precise static analysis to detect atomicity violations caused by the use of finer-grain transactions. We developed a novel approach to detect high-level data races and stale-value errors that relies on the notion of *causal dependencies* to improve the precision over previous detection techniques. Moreover, we were able to unify the detection of both high-level data races and stale-value errors within the same theoretical framework, using the graph of causal dependencies. These static analysis algorithms were implemented in a tool called MoTH, which identifies atomicity violations in Java bytecode programs. Our detection analysis still remains unsound. Nevertheless, as our experimental results confirm, the design decisions that we made allowed to maintain the scalability of our approach while maintaining a very good precision level. The next challenge of this work will be to develop a sound static analysis without losing scalability and precision with respect to false positives. Also, the integration of this tool with existing IDEs, to detect misplacements of atomic blocks, would allow the increase of productivity in the software development cycle of concurrent programs.

The use of a relaxed isolation level, such as snapshot isolation, has the potential to increase parallelism of transactional systems at the cost of losing opacity. Snapshot isolation allows the occurrence of serialization anomalies known as write-skews. To solve this problem, we proposed a static verification procedure to certify that transactional memory programs, executing under snapshot isolation, are free from write-skew anomalies. This verification procedure grounds on a

state-of-the-art shape analysis technique based on separation logic and extend it with heap path expressions, which represent abstract memory locations. Our analysis technique can compute an approximation of the read- and write-sets of each transaction, which then can be applied to detect the possibility of the occurrence of write-skews at execution time. Our algorithm is sound and hence suffers from over-reporting but not from under-reporting, i.e., all the write skews in the program are detected but some false warnings may also be generated. We implemented the verification algorithm in a tool called StarTM which can be applied to Java bytecode programs.

The proposed verification procedure, although being the first published approach to statically detect write-skews in transactional memory programs, has still many limitations in the nature of the programs to be analyzed, such as dealing with large-sized programs, with arrays and with cyclic data structures. Our approach is a first step towards a new research topic of detecting serialization anomalies in transactional memory programs and much more can be done. For instance, more efficient abstract memory representations could limit the impact of the state explosion and improve the scalability of the analysis, and a modular static analysis algorithm can also be developed enabling the verification of large-sized programs. Another research direction would be to employ sound dynamic analysis techniques to solve the same problem.

This work also contributed to the development of a generic and extensible STM framework to support the efficient implementation of several STM algorithms. In particular, our framework supports the efficient implementation of both single- and multi-version algorithms. We extended the Deuce framework to support in-place metadata, i.e., the co-location of transactional metadata near the object fields instead of in a shared external mapping table. The extension provided a successfully runtime infrastructure to the efficient implementation of multi-version algorithms, allowing for the first time a fair comparison of single- and multi-version algorithms in the same framework and using exactly the same benchmarking programs.

The technique that we developed to co-locate metadata near object fields is very effective for class fields, but has a non-negligible time and space overhead for array elements. The support for in-place metadata in array objects, using only bytecode instrumentation, is a difficult task because of the restricted memory structure of the arrays. It would be interesting to evaluate an implementation of our approach of in-place metadata at the virtual machine level, which we believe would allow a more efficient implementation for array objects at the cost of portability.

Our proposed extension can also be enhanced to transparently support distributed STM algorithms. This goal would probably require the generalization of the STM algorithms interface to include additional callbacks to support inter-node synchronization. Moreover, it is required a modular architecture to specify a global algorithm to coordinate the different nodes of the system, and a centralized algorithm to coordinate the threads within each node. Another research direction following this work is the development of static analysis techniques to reduce the over-instrumentation inherent to these extensible and transparent frameworks, narrowing the gap between a programmer tailored source-code program and and automatically instrumented version.

Finally, all the developed techniques presented in this dissertation can be assembled into a single framework, providing compile-time and runtime support in the form of a software transactional memory stack, to Java applications.

Bibliography

[AWZ88] B. Alpern, M. N. Wegman, and F. K. Zadeck. "Detecting Equality of Variables in Programs". In: *Proc. of the 15th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*. POPL '88. San Diego, California, United States: ACM, 1988, pp. 1–11. ISBN: 0-89791-252-7. DOI: http://doi.acm.org/10.1145/73560.73561.

[AHB03] C. Artho, K. Havelund, and A. Biere. "High-level data races". In: *Software Testing, Verification and Reliability* 13.4 (Dec. 2003), pp. 207–227. ISSN: 0960-0833. DOI: 10.1002/stvr.281.

[AHB04] C. Artho, K. Havelund, and A. Biere. "Using block-local atomicity to detect stale-value concurrency errors". In: *Automated Technology for Verification and Analysis* (2004), pp. 150–164.

[BT07] C. Barrett and C. Tinelli. "CVC3". In: *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*. Ed. by W. Damm and H. Hermanns. Vol. 4590. LNCS. Springer-Verlag, 2007, pp. 298–302.

[BBA08] N. E. Beckman, K. Bierhoff, and J. Aldrich. "Verifying Correct Usage of Atomic Blocks and Typestate". In: *SIGPLAN Not.* 43.10 (2008), pp. 227–244. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/1449955.1449783.

[BCCDOWY07] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. "Shape analysis for composite data structures". In: *Proceedings of the 19th international conference on Computer aided verification*. CAV'07. Berlin, Germany: Springer-Verlag, 2007, pp. 178–192. ISBN: 978-3-540-73367-6.

[BCO05] J. Berdine, C. Calcagno, and P. W. O'Hearn. "Symbolic execution with separation logic". In: *Proceedings of the Third Asian conference on Programming Languages and Systems*. APLAS'05. Tsukuba, Japan: Springer-Verlag, 2005, pp. 52–68. ISBN: 3-540-29735-9, 978-3-540-29735-2. DOI: 10.1007/11575467_5. URL: http://dx.doi.org/10.1007/11575467_5.

[BBGMOO95] H. Berenson, P. Bernstein, J. N. Gray, J. Melton, E. O'Neil, and P. O'Neil. "A critique of ANSI SQL isolation levels". In: *SIGMOD '95: Proceedings of the 1995*

ACM SIGMOD international conference on Management of data. San Jose, California, United States: ACM, 1995, pp. 1–10. ISBN: 0-89791-731-6. DOI: 10.1145/223784.223785.

[BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1987. ISBN: 0-201-10715-5.

[BHM07] W. Binder, J. Hulaas, and P. Moret. "Advanced Java bytecode instrumentation". In: *Proceedings of the International Symposium on Principles and Practice of Programming in Java (PPPJ)*. 2007, pp. 135–144.

[Blo08] J. Bloch. *Effective Java (2nd Edition)*. Addison-Wesley, 2008.

[BLM05] C. Blundell, E. C. Lewis, and M. M. K. Martin. "Deconstructing Transactions: The Subtleties of Atomicity". In: *Fourth Annual Workshop on Duplicating, Deconstructing, and Debunking*. Publisher unkownn, 2005.

[BBC08] J. Brotherston, R. Bornat, and C. Calcagno. "Cyclic proofs of program termination in separation logic". In: *Proc. of the 35th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. POPL '08. San Francisco, California, USA: ACM, 2008, pp. 101–112. ISBN: 978-1-59593-689-9. DOI: 10.1145/1328438.1328453.

[BK10] J. Brotherston and M. Kanovich. "Undecidability of Propositional Separation Logic and Its Neighbours". In: *Proceedings of the 2010 25th Annual IEEE Symposium on Logic in Computer Science*. LICS '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 130–139. ISBN: 978-0-7695-4114-3. DOI: 10.1109/LICS.2010.24.

[BL04] M. Burrows and K. Leino. "Finding stale-value errors in concurrent programs". In: *Concurrency and Computation: Practice and Experience* 16.12 (2004), pp. 1161–1172.

[CRS06] J. Cachopo and A. Rito-Silva. "Versioned boxes as the basis for memory transactions". In: *Sci. Comput. Program.* 63.2 (2006), pp. 172–185. ISSN: 0167-6423. DOI: http://dx.doi.org/10.1016/j.scico.2006.05.009.

[CD11] C. Calcagno and D. Distefano. "Infer: an automatic program verifier for memory safety of C programs". In: *Proceedings of the Third international conference on NASA Formal methods*. NFM'11. Pasadena, CA: Springer-Verlag, 2011, pp. 459–465. ISBN: 978-3-642-20397-8.

[CDOY09] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. "Compositional shape analysis by means of bi-abduction". In: *Proc. of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '09. Savannah, GA, USA: ACM, 2009, pp. 289–300. ISBN: 978-1-60558-379-2.

[CDOY07] C. Calcagno, D. Distefano, P. O'Hearn, and H. Yang. "Footprint Analysis: A Shape Analysis That Discovers Preconditions". In: *Static Analysis*. Ed. by H. Nielson and G. Filé. Vol. 4634. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, pp. 402–418. ISBN: 978-3-540-74060-5.

[CMCKO08] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. "STAMP: Stanford Transactional Applications for Multi-Processing". In: *IISWC '08: Proc. IEEE Int. Symp. on Workload Characterization*. 2008.

[CLLOSS02] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. "Efficient and precise datarace detection for multithreaded object-oriented programs". In: *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*. PLDI '02. Berlin, Germany: ACM, 2002, pp. 258–269. ISBN: 1-58113-463-0. DOI: 10.1145/512529.512560. URL: http://doi.acm.org/10.1145/512529.512560.

[Cor08] A. Cortesi. "Widening Operators for Abstract Interpretation". In: *Software Engineering and Formal Methods*. 2008, pp. 31–40. DOI: http://dx.doi.org/10.1109/SEFM.2008.20.

[Cou01] P. Cousot. "Abstract Interpretation Based Formal Methods and Future Challenges". In: *Informatics*. 2001, pp. 138–156. DOI: http://dx.doi.org/10.1007/3-540-44577-3_10.

[CC77] P. Cousot and R. Cousot. "Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. POPL '77. Los Angeles, California: ACM, 1977, pp. 238–252. DOI: 10.1145/512950.512973. URL: http://doi.acm.org/10.1145/512950.512973.

[DV12] R. Demeyer and W. Vanhoof. "A Framework for Verifying the Application-Level Race-Freeness of Concurrent Programs". In: *22nd Workshop on Logic-based Programming Environments (WLPE 2012)*. 2012, p. 10.

[DDSL12] R. J. Dias, D. Distefano, J. C. Seco, and J. M. Lourenço. "Verification of Snapshot Isolation in Transactional Memory Java Programs". In: *ECOOP 2012 – Object-Oriented Programming*. Ed. by J. Noble. Vol. 7313. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2012, pp. 640–664. ISBN: 978-3-642-31056-0. URL: http://dx.doi.org/10.1007/978-3-642-31057-7_28.

[DLP11] R. J. Dias, J. M. Lourenço, and N. M. Preguiça. "Efficient and Correct Transactional Memory Programs Combining Snapshot Isolation and Static Analysis". In: *Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar'11)*. HotPar'11. http://asc.di.fct.unl.pt/~nmp/pubs/hotpar-2011.pdf. Usenix Association, May 2011.

[DPL12] R. J. Dias, V. Pessanha, and J. M. Lourenço. "Precise Detection of Atomicity Violations". In: *Haifa Verification Conference (HVC 2012)*. Lecture Notes in Computer Science. Springer-Verlag, Nov. 2012.

[DVL12] R. J. Dias, T. M. Vale, and J. M. Lourenço. "Efficient Support for In-Place Metadata in Transactional Memory". In: *Euro-Par 2012 Parallel Processing*. Ed. by C. Kaklamanis, T. Papatheodorou, and P. Spirakis. Vol. 7484. Lecture Notes in

Computer Science. Springer Berlin / Heidelberg, 2012, pp. 589–600. ISBN: 978-3-642-32819-0. URL: http://dx.doi.org/10.1007/978-3-642-32820-6_59.

[DVL13] R. J. Dias, T. M. Vale, and J. M. Lourenço. "Efficient support for in-place metadata in Java software transactional memory (submitted)". In: *Concurrency and Computation: Practice and Experience* (2013).

[DSS06] D. Dice, O. Shalev, and N. Shavit. "Transactional Locking II". In: *Distributed Computing*. Vol. 4167. Stockholm, Sweden: Springer Berlin / Heidelberg, 2006, pp. 194–208.

[DOY06] D. Distefano, P. W. O'Hearn, and H. Yang. "A Local Shape Analysis Based on Separation Logic". In: *Tools and Algorithms for the Construction and Analysis of 12th International Conference (TACAS 2006)*. Lecture Notes in Computer Science. Springer, 2006, pp. 287–302.

[DPJ08] D. Distefano and M. J. Parkinson J. "jStar: towards practical verification for java". In: *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*. OOPSLA '08. Nashville, TN, USA: ACM, 2008, pp. 213–226. ISBN: 978-1-60558-215-3. DOI: 10.1145/1449764.1449782.

[EGLT76] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. "The notions of consistency and predicate locks in a database system". In: *Commun. ACM* 19.11 (1976), pp. 624–633. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/360363.360369.

[FLOOS05] A. Fekete, D. Liarokapis, E. O'Neil, P. O'Neil, and D. Shasha. "Making snapshot isolation serializable". In: *ACM Trans. Database Syst.* 30.2 (2005), pp. 492–528. ISSN: 0362-5915. DOI: 10.1145/1071610.1071615.

[FC11] S. M. Fernandes and J. a. Cachopo. "Lock-free and scalable multi-version software transactional memory". In: *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*. PPoPP '11. San Antonio, TX, USA: ACM, 2011, pp. 179–188. ISBN: 978-1-4503-0119-0. DOI: 10.1145/1941553.1941579.

[FF04] C. Flanagan and S. N. Freund. "Atomizer: a dynamic atomicity checker for multithreaded programs". In: *SIGPLAN Not.* 39.1 (Jan. 2004), pp. 256–267. ISSN: 0362-1340. DOI: 10.1145/982962.964023.

[FF10] C. Flanagan and S. N. Freund. "FastTrack: efficient and precise dynamic race detection". In: *Commun. ACM* 53.11 (Nov. 2010), pp. 93–101. ISSN: 0001-0782. DOI: 10.1145/1839676.1839699.

[FFY08] C. Flanagan, S. N. Freund, and J. Yi. "Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs". In: *SIGPLAN Not.* 43.6 (June 2008), pp. 293–303. ISSN: 0362-1340. DOI: 10.1145/1379022.1375618.

[FQ03] C. Flanagan and S. Qadeer. "Types for atomicity". In: *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*. New Orleans, Louisiana, USA: ACM, 2003, pp. 1–12. ISBN: 1-58113-649-8. DOI: http://doi.acm.org/10.1145/604174.604176.

[FH07] K. Fraser and T. Harris. "Concurrent programming without locks". In: *ACM Trans. Comput. Syst.* 25.2 (2007), p. 5. ISSN: 0734-2071. DOI: http://doi.acm.org/10.1145/1233307.1233309.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994, p. 416. ISBN: 0201633612.

[GBC06] A. Gotsman, J. Berdine, and B. Cook. "Interprocedural shape analysis with separated heap abstractions". In: *Proceedings of the 13th international conference on Static Analysis*. SAS'06. Seoul, Korea: Springer-Verlag, 2006, pp. 240–260. ISBN: 3-540-37756-5, 978-3-540-37756-6. DOI: 10.1007/11823230_16.

[GR92] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1992. ISBN: 1558601902.

[GK08] R. Guerraoui and M. Kapalka. "On the correctness of transactional memory". In: *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. Salt Lake City, UT, USA: ACM, 2008, pp. 175–184. ISBN: 978-1-59593-795-7. DOI: http://doi.acm.org/10.1145/1345206.1345233.

[GKV07] R. Guerraoui, M. Kapalka, and J. Vitek. "STMBench7: a benchmark for software transactional memory". In: *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. EuroSys '07. Lisbon, Portugal: ACM, 2007, pp. 315–324. ISBN: 978-1-59593-636-3. DOI: 10.1145/1272996.1273029.

[HLM06] M. Herlihy, V. Luchangco, and M. Moir. "A Flexible Framework for Implementing Software Transactional Memory". In: *Proc. 21st conference on Object-Oriented Programming Systems, Languages, and Applications*. Portland, Oregon, USA: ACM, 2006, pp. 253–262. ISBN: 1-59593-348-4. DOI: http://doi.acm.org/10.1145/1167473.1167495.

[HLMWNS03] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. "Software transactional memory for dynamic-sized data structures". In: *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*. Boston, Massachusetts: ACM, 2003, pp. 92–101. ISBN: 1-58113-708-7. DOI: http://doi.acm.org/10.1145/872035.872048.

[HM93] M. Herlihy and J. E. B. Moss. "Transactional memory: architectural support for lock-free data structures". In: *ISCA '93: Proceedings of the 20th annual international symposium on Computer architecture*. San Diego, California, United States: ACM, 1993, pp. 289–300. ISBN: 0-8186-3810-9. DOI: http://doi.acm.org/10.1145/165123.165164.

[HW90] M. P. Herlihy and J. M. Wing. "Linearizability: a correctness condition for concurrent objects". In: *ACM Trans. Program. Lang. Syst.* 12.3 (1990), pp. 463–492. ISSN: 0164-0925. DOI: http://doi.acm.org/10.1145/78969.78972.

[Hoa69] C. A. R. Hoare. "An axiomatic basis for computer programming". In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: http://doi.acm.org/10.1145/363235.363259.

[Ibm] *IBM HRL — Concurrency Testing Repository*.

[JFRS07] S. Jorwekar, A. Fekete, K. Ramamritham, and S. Sudarshan. "Automating the detection of snapshot isolation anomalies". In: *VLDB '07: Proceedings of the 33rd international conference on Very large data bases*. Vienna, Austria: VLDB Endowment, 2007, pp. 1263–1274.

[KSF10] G. Korland, N. Shavit, and P. Felber. "Noninvasive Concurrency with Java STM". In: *Proc. MultiProg 2010: Programmability Issues for Heterogeneous Multicores*. 2010.

[Lip75] R. J. Lipton. "Reduction: a method of proving properties of parallel programs". In: *Commun. ACM* 18.12 (Dec. 1975), pp. 717–721. ISSN: 0001-0782. DOI: 10.1145/361227.361234.

[Lom77] D. B. Lomet. "Process structuring, synchronization, and recovery using atomic actions". In: *SIGPLAN Not.* 12.3 (1977), pp. 128–137. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/390017.808319.

[LSTD11] J. Lourenço, D. Sousa, B. Teixeira, and R. Dias. "Detecting concurrency anomalies in transactional memory programs". In: *Computer Science and Information Systems/ComSIS* 8.2 (2011), pp. 533–548.

[LPSZ08] S. Lu, S. Park, E. Seo, and Y. Zhou. "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics". In: *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*. ASPLOS XIII. Seattle, WA, USA: ACM, 2008, pp. 329–339. ISBN: 978-1-59593-958-6. DOI: 10.1145/1346281.1346323. URL: http://doi.acm.org/10.1145/1346281.1346323.

[MBSATHSW08] V. Menon, S. Balensiefer, T. Shpeisman, A.-R. Adl-Tabatabai, R. L. Hudson, B. Saha, and A. Welc. "Single global lock semantics in a weakly atomic STM". In: *SIGPLAN Not.* 43.5 (2008), pp. 15–26. ISSN: 0362-1340. DOI: http://doi.acm.org/10.1145/1402227.1402235.

[MHFA13] J. Mund, R. Huuck, A. Fehnker, and C. Artho. "The Quest for Precision: A Layered Approach for Data Race Detection in Static Analysis". In: *Automated Technology for Verification and Analysis*. Ed. by D. Hung and M. Ogawa. Vol. 8172. Lecture Notes in Computer Science. Springer International Publishing, 2013, pp. 516–525. ISBN: 978-3-319-02443-1. DOI: 10.1007/978-3-319-02444-8_45. URL: http://dx.doi.org/10.1007/978-3-319-02444-8_45.

[Ora12] Oracle. *java.lang.instrument.Instrument*. http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html. 2012.

[PBLK11] D. Perelman, A. Byshevsky, O. Litmanovich, and I. Keidar. "SMV: selective multi-versioning STM". In: *Proceedings of the 25th international conference on Distributed computing*. DISC'11. Rome, Italy: Springer-Verlag, 2011, pp. 125–140. ISBN: 978-3-642-24099-7. URL: http://dl.acm.org/citation.cfm?id=2075029.2075041.

[PFK10] D. Perelman, R. Fan, and I. Keidar. "On maintaining multiple versions in STM". In: *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*. PODC '10. Zurich, Switzerland: ACM, 2010, pp. 16–25. ISBN: 978-1-60558-888-9. DOI: 10.1145/1835698.1835704. URL: http://doi.acm.org/10.1145/1835698.1835704.

[Pes11] V. Pessanha. "Verificação Prática de Anomalias em Programas de Memória Transaccional". MA thesis. Lisbon, Portugal: Universidade Nova de Lisboa, 2011.

[PDLFS11] V. Pessanha, R. J. Dias, J. M. Lourenço, E. Farchi, and D. Sousa. "Practical Verification of Transactional Memory Programs". In: *Proceedings of PADTAD 2011: Workshop on Parallel and Distributed Testing, Analysis and Debugging*. ACM Electronic Library, July 2011.

[PRV10] P. Prabhu, G. Ramalingam, and K. Vaswani. "Safe programmable speculative parallelism". In: *Proceedings of the 2010 ACM SIGPLAN conference on Programming language design and implementation*. PLDI '10. Toronto, Ontario, Canada: ACM, 2010, pp. 50–61. ISBN: 978-1-4503-0019-3. DOI: 10.1145/1806596.1806603. URL: http://doi.acm.org/10.1145/1806596.1806603.

[PMPM11] D. Prountzos, R. Manevich, K. Pingali, and K. S. McKinley. "A shape analysis for optimizing parallel graph programs". In: *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '11. Austin, Texas, USA: ACM, 2011, pp. 159–172. ISBN: 978-1-4503-0490-0. DOI: 10.1145/1926385.1926405. URL: http://doi.acm.org/10.1145/1926385.1926405.

[RHSLGC99] Raja Vallée-Rai, L. Hendren, V. Sundaresan, P. Lam, E. Gagnon, and P. Co. "Soot - a Java Optimization Framework". In: *Proceedings of CASCON 1999*. 1999, pp. 125–135. URL: http://www.sable.mcgill.ca/publications.

[RCG09] M. Raza, C. Calcagno, and P. Gardner. "Automatic Parallelization with Separation Logic". In: *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. ESOP '09. York, UK: Springer-Verlag, 2009, pp. 348–362. ISBN: 978-3-642-00589-3. DOI: 10.1007/978-3-642-00590-9_25. URL: http://dx.doi.org/10.1007/978-3-642-00590-9_25.

[Rey02] J. C. Reynolds. "Separation Logic: A Logic for Shared Mutable Data Structures". In: *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*. LICS '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 55–74. ISBN: 0-7695-1483-9. URL: http://portal.acm.org/citation.cfm?id=645683.664578.

[RB08] T. Riegel and D. B. D. Brum. "Making Object-Based STM Practical in Unmanaged Environments". In: *Proc. of the 3rd Workshop on Transactional Computing*. 2008.

[RFF06] T. Riegel, C. Fetzer, and P. Felber. "Snapshot Isolation for Software Transactional Memory". In: *TRANSACT'06: First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*. Ottawa, Canada, June 2006.

[SRW96] M. Sagiv, T. Reps, and R. Wilhelm. "Solving shape-analysis problems in languages with destructive updating". In: *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. St. Petersburg Beach, Florida, United States: ACM, 1996, pp. 16–31. ISBN: 0-89791-769-3. DOI: 10.1145/237721.237725.

[SRW98] M. Sagiv, T. Reps, and R. Wilhelm. "Solving shape-analysis problems in languages with destructive updating". In: *ACM Trans. Program. Lang. Syst.* 20.1 (1998), pp. 1–50. ISSN: 0164-0925. DOI: 10.1145/271510.271517.

[SRW99] M. Sagiv, T. Reps, and R. Wilhelm. "Parametric shape analysis via 3-valued logic". In: *POPL '99: Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. San Antonio, Texas, United States: ACM, 1999, pp. 105–118. ISBN: 1-58113-095-3. DOI: 10.1145/292540.292552.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. "Parametric shape analysis via 3-valued logic". In: *ACM Trans. Program. Lang. Syst.* 24.3 (2002), pp. 217–298. ISSN: 0164-0925. DOI: 10.1145/514188.514190.

[SBNSA97] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. "Eraser: a dynamic data race detector for multithreaded programs". In: *ACM Trans. Comput. Syst.* 15.4 (1997), pp. 391–411. ISSN: 0734-2071. DOI: 10.1145/265924.265927. URL: http://doi.acm.org/10.1145/265924.265927.

[Sco06] M. L. Scott. "Sequential Specification of Transactional Memory Semantics". In: *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*. 2006.

[SBASVY11] O. Shacham, N. Bronson, A. Aiken, M. Sagiv, M. Vechev, and E. Yahav. "Testing atomicity of composed concurrent operations". In: *SIGPLAN Not.* 46.10 (Oct. 2011), pp. 51–64. ISSN: 0362-1340. DOI: 10.1145/2076021.2048073.

[ST95] N. Shavit and D. Touitou. "Software transactional memory". In: *PODC '95: Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*. Ottowa, Ontario, Canada: ACM, 1995, pp. 204–213. ISBN: 0-89791-710-3. DOI: http://doi.acm.org/10.1145/224964.224987.

[SR05] A. Sălcianu and M. Rinard. "Purity and side effect analysis for java programs". In: *Proceedings of the 6th international conference on Verification, Model Checking, and Abstract Interpretation*. VMCAI'05. Paris, France: Springer-Verlag, 2005, pp. 199–215. ISBN: 3-540-24297-X, 978-3-540-24297-0. DOI: 10.1007/978-3-540-30579-8_14. URL: http://dx.doi.org/10.1007/978-3-540-30579-8_14.

[TLFDS10] B. C. Teixeira, J. M. Lourenço, E. Farchi, R. J. Dias, and D. Sousa. "Detection of Transactional Memory Anomalies using Static Analysis". In: *Proceedings of the International Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*. Ed. by S. U. João Lourenço Eitan Farchi. ACM Electronic Library, July 2010, pp. 26–36.

[Tra10] Transaction Processing Performance Council. *TPC-C Benchmark, Revision 5.11*. 2010.

[VRCGHLS99] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. "Soot - a Java bytecode optimization framework". In: *Proc. of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. CASCON '99. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13–.

[VTD06] M. Vaziri, F. Tip, and J. Dolby. "Associating synchronization constraints with data in an object-oriented language". In: *Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. POPL '06. Charleston, South Carolina, USA: ACM, 2006, pp. 334–345. ISBN: 1-59593-027-2. DOI: 10.1145/1111037.1111067. URL: http://doi.acm.org/10.1145/1111037.1111067.

[VPG04] C. Von Praun and T. Gross. "Static detection of atomicity violations in object-oriented programs". In: *Journal of Object Technology* 3.6 (2004), pp. 103–122.

[vG03] C. von Praun and T. R. Gross. "Static Detection of Atomicity Violations in Object-Oriented Programs". In: *Journal of Object Technology*. 2003, p. 2004.

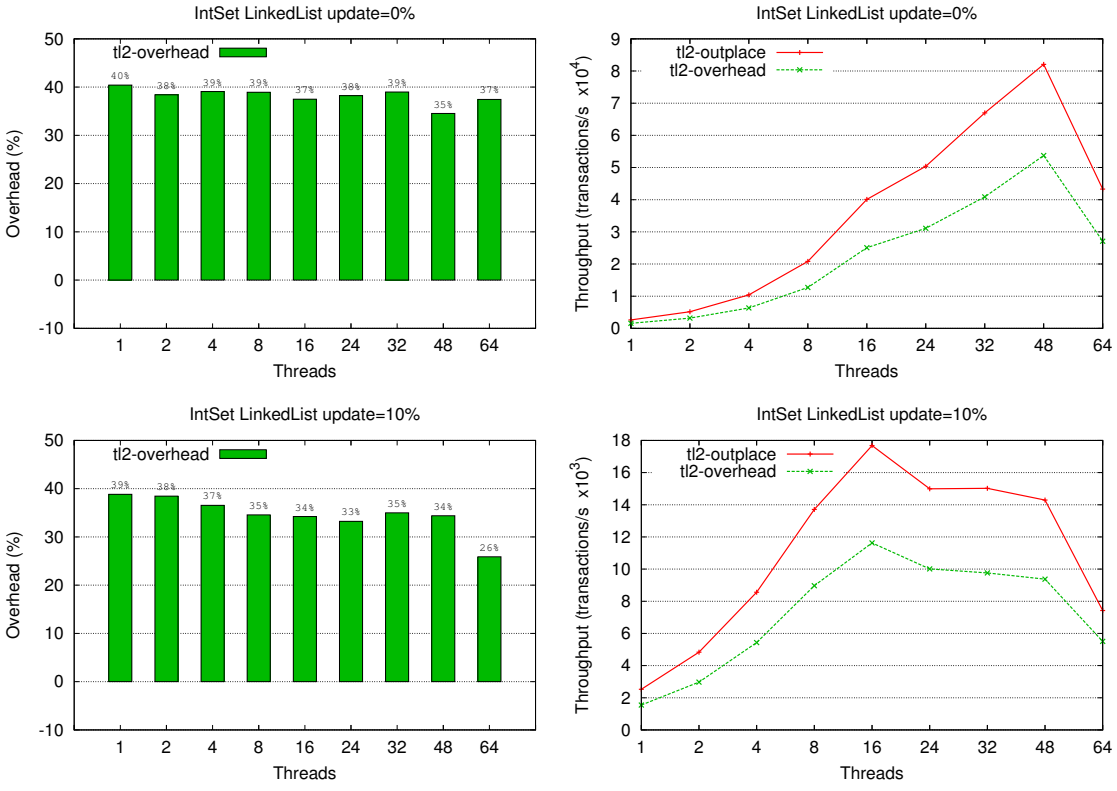
[WS03] L. Wang and S. D. Stoller. "Run-Time Analysis for Atomicity". In: *Electronic Notes in Theoretical Computer Science* 89.2 (2003), pp. 191–209. ISSN: 15710661. DOI: 10.1016/S1571-0661(04)81049-1.

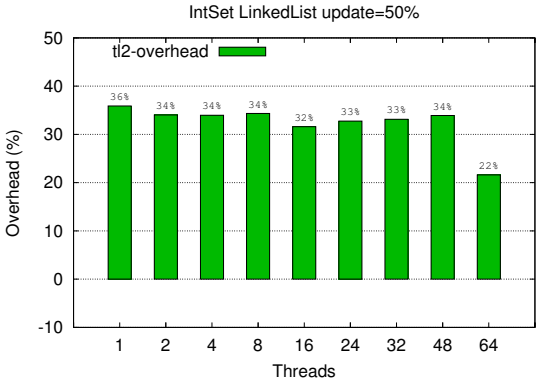
[YLBCCDO08] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. "Scalable Shape Analysis for Systems Code". In: *Proceedings of the 20th international conference on Computer Aided Verification*. CAV '08. Princeton, NJ, USA: Springer-Verlag, 2008, pp. 385–398. ISBN: 978-3-540-70543-7. DOI: 10.1007/978-3-540-70545-1_36.

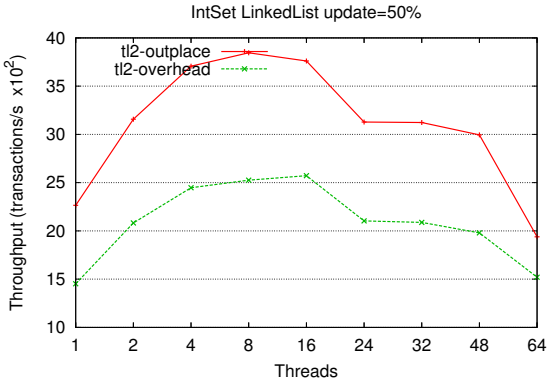
Detailed Execution Results

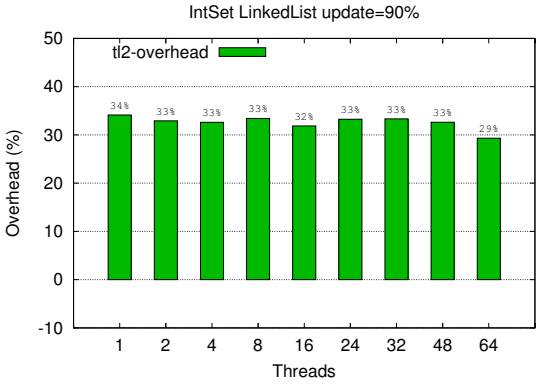
A.1 In-place Metadata Overhead

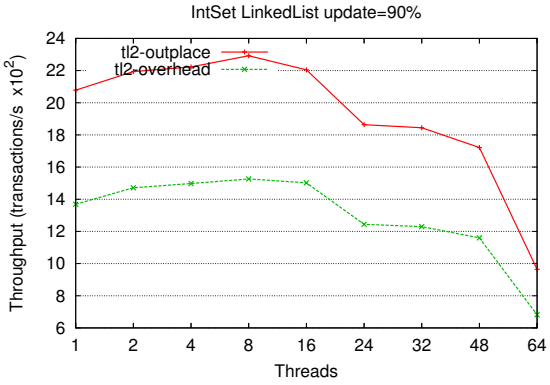
In this appendix we present the detailed results of Section 5.4.1 from comparing the TL2 algorithm, as implemented in the original Deuce framework, and the exact same TL2 algorithm but implemented using the in-place extension.

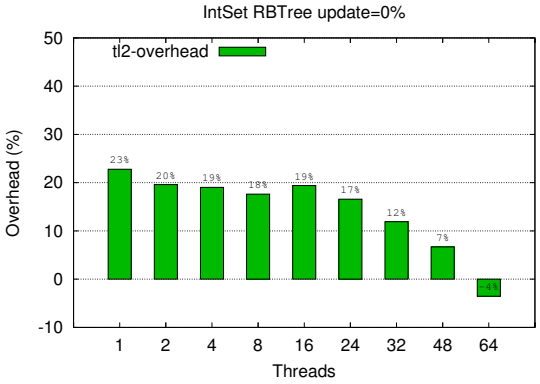


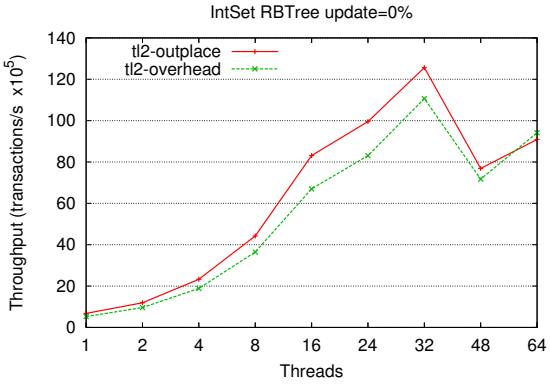

IntSet LinkedList update=50%
tl2-overhead
Overhead (%)
Threads
36%
34%
34%
34%
32%
33%
33%
34%
22%

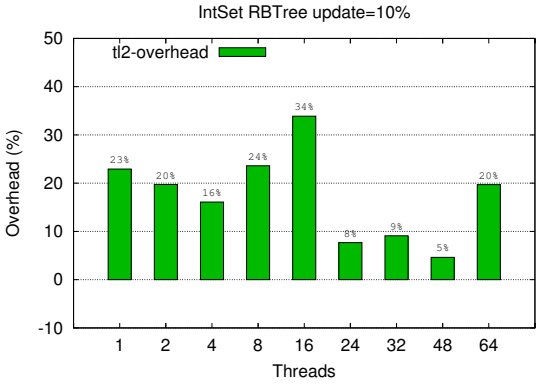

IntSet LinkedList update=50%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^2)
Threads

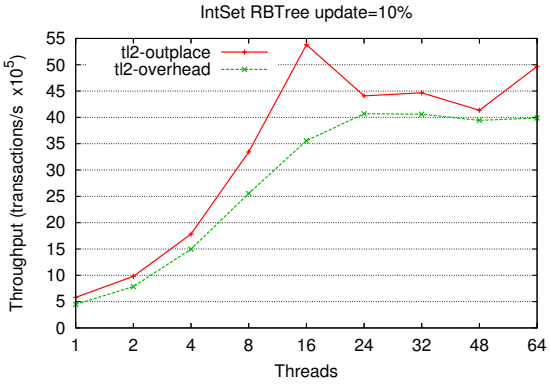

IntSet LinkedList update=90%
tl2-overhead
Overhead (%)
Threads
34%
33%
33%
33%
32%
33%
33%
33%
29%

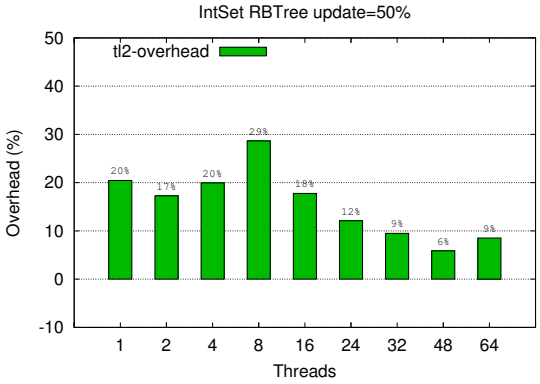

IntSet LinkedList update=90%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^2)
Threads

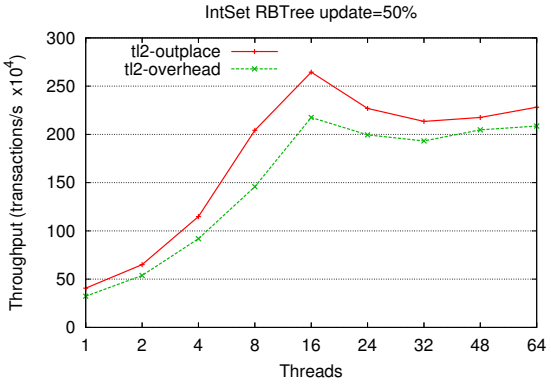

IntSet RBTree update=0%
tl2-overhead
Overhead (%)
Threads
23%
20%
19%
18%
19%
17%
12%
7%
-4%

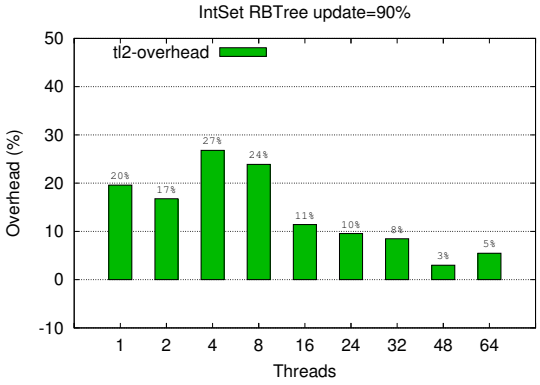

IntSet RBTree update=0%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^5)
Threads

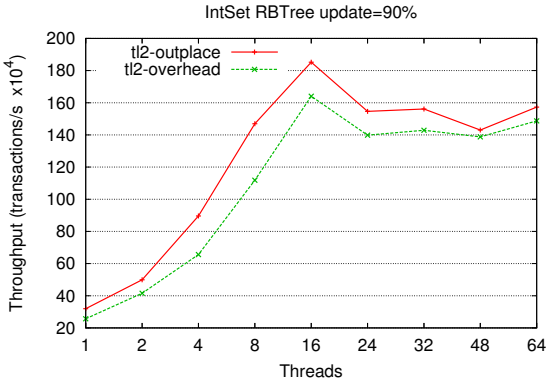

IntSet RBTree update=10%
tl2-overhead
Overhead (%)
Threads
23%
20%
16%
24%
34%
8%
9%
5%
20%

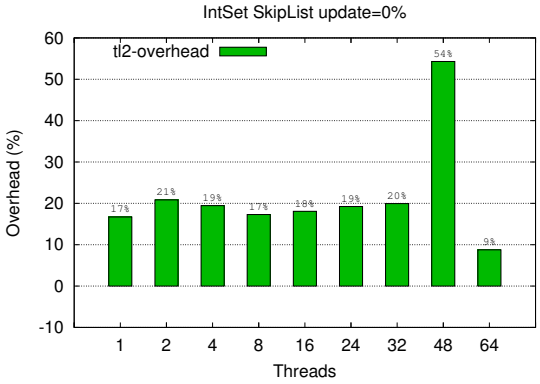

IntSet RBTree update=10%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^5)
Threads

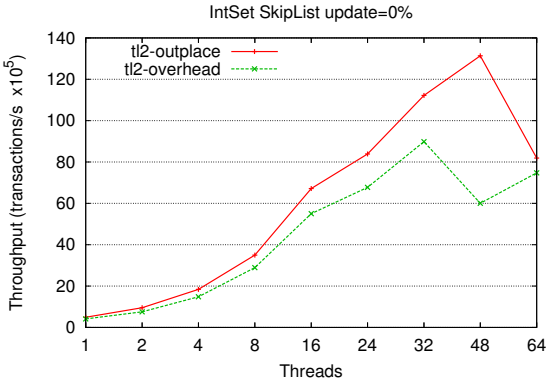

IntSet RBTree update=50%
tl2-overhead
Overhead (%)
Threads
20%
17%
20%
29%
18%
12%
9%
6%
9%

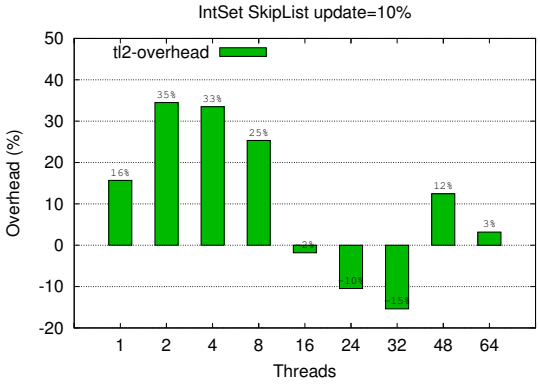

IntSet RBTree update=50%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^4)
Threads

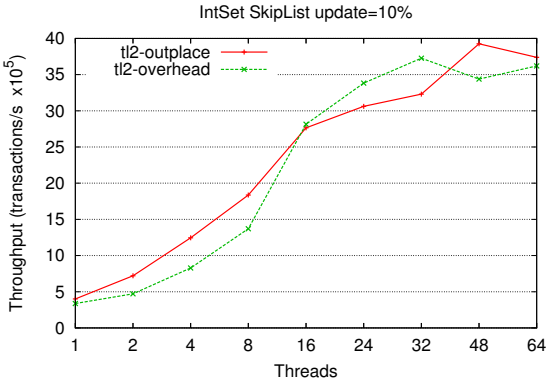

IntSet RBTree update=90%
tl2-overhead
Overhead (%)
Threads
20%
17%
27%
24%
11%
10%
8%
3%
5%

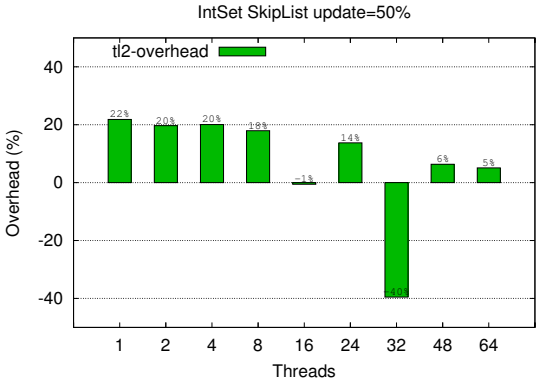

IntSet RBTree update=90%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^4)
Threads

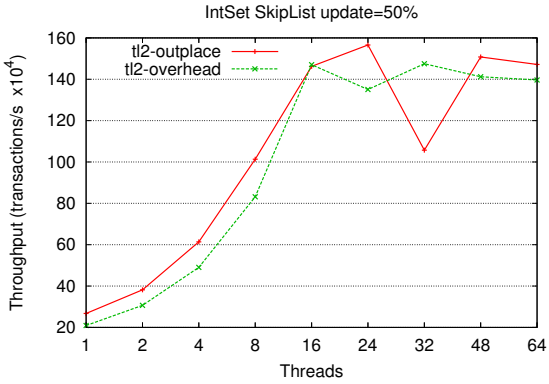

IntSet SkipList update=0%
tl2-overhead
Overhead (%)
Threads
17%
21%
19%
17%
18%
19%
20%
54%
9%

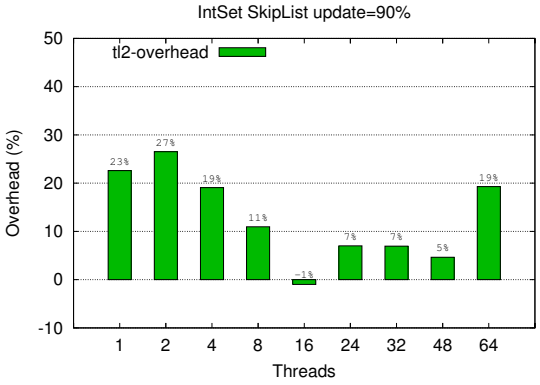

IntSet SkipList update=0%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^5)
Threads

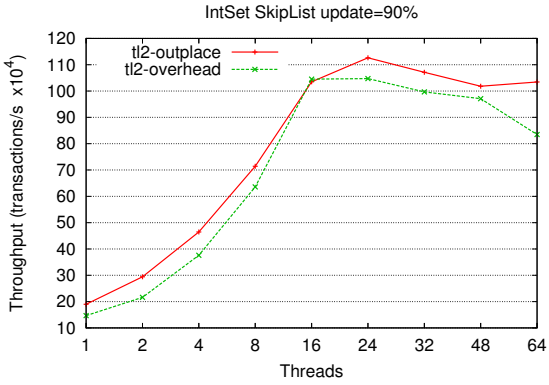

IntSet SkipList update=10%
tl2-overhead
Overhead (%)
Threads
16%
35%
33%
25%
12%
3%

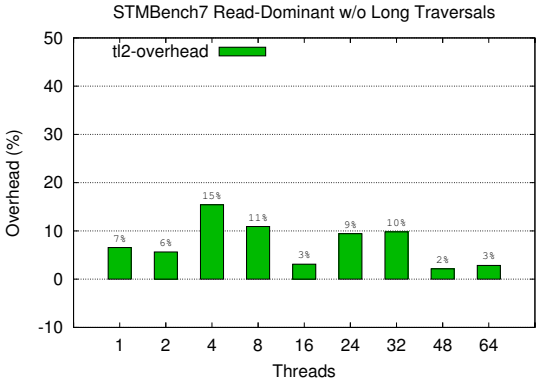

IntSet SkipList update=10%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^5)
Threads

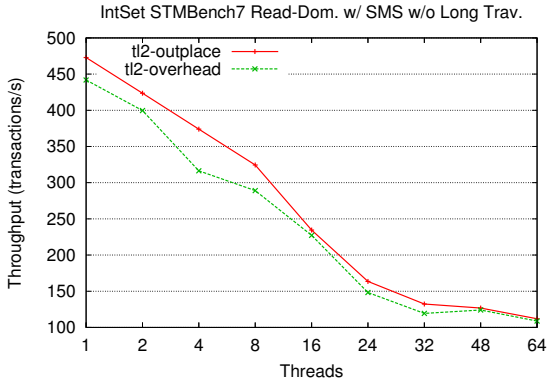

IntSet SkipList update=50%
tl2-overhead
Overhead (%)
Threads
22%
20%
20%
18%
-1%
14%
-40%
6%
5%

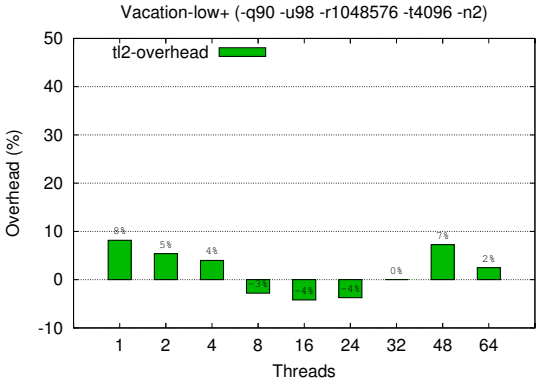

IntSet SkipList update=50%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^4)
Threads

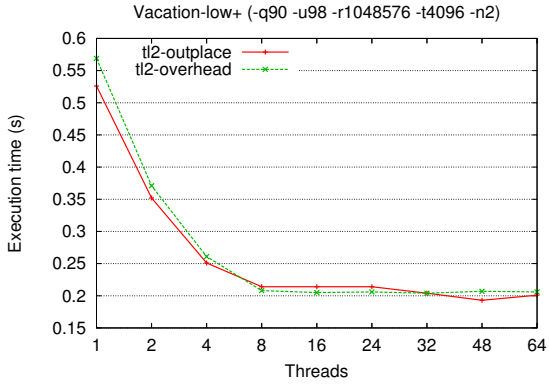

IntSet SkipList update=90%
tl2-overhead
Overhead (%)
Threads
23%
27%
19%
11%
-1%
7%
7%
5%
19%

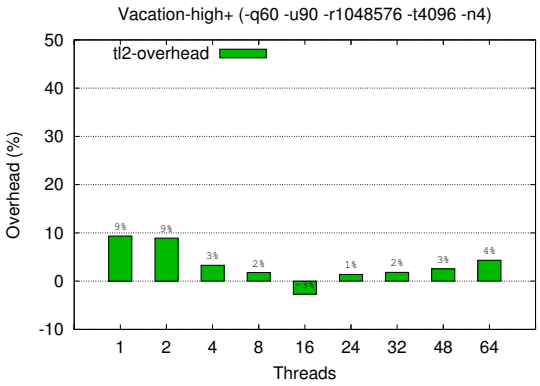

IntSet SkipList update=90%
tl2-outplace
tl2-overhead
Throughput (transactions/s x10^4)
Threads

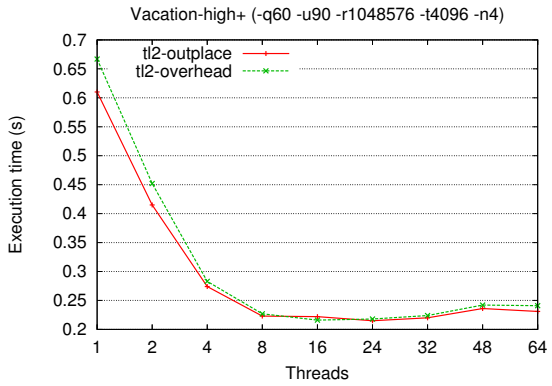

STMBench7 Read-Dominant w/o Long Traversals
tl2-overhead
Overhead (%)
Threads
7%
6%
15%
11%
3%
9%
10%
2%
3%

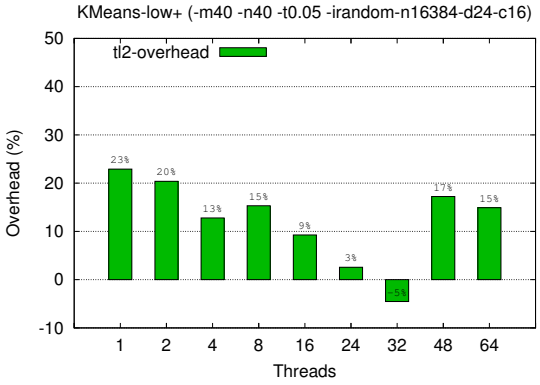

IntSet STMBench7 Read-Dom. w/ SMS w/o Long Trav.
tl2-outplace
tl2-overhead
Throughput (transactions/s)
Threads

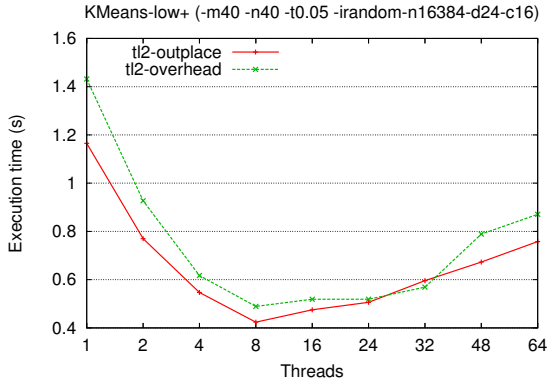

Vacation-low+ (-q90 -u98 -r1048576 -t4096 -n2)
tl2-overhead
Overhead (%)
Threads
8%
5%
4%
0%
7%
2%

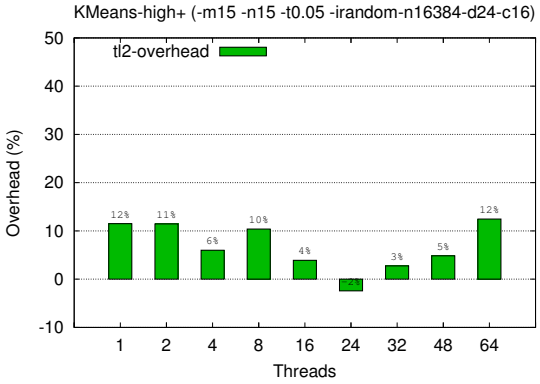

Vacation-low+ (-q90 -u98 -r1048576 -t4096 -n2)
tl2-outplace
tl2-overhead
Execution time (s)
Threads

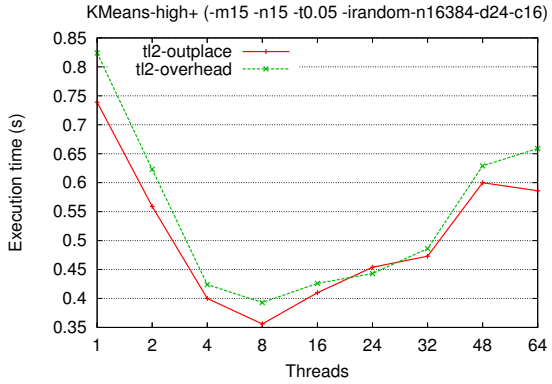

Vacation-high+ (-q60 -u90 -r1048576 -t4096 -n4)
tl2-overhead
Overhead (%)
Threads
9%
9%
3%
2%
-3%
1%
2%
3%
4%

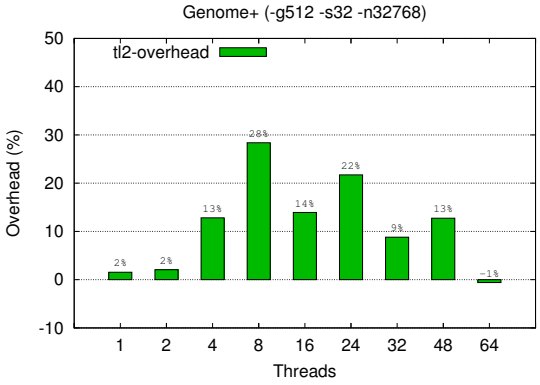

Vacation-high+ (-q60 -u90 -r1048576 -t4096 -n4)
tl2-outplace
tl2-overhead
Execution time (s)
Threads

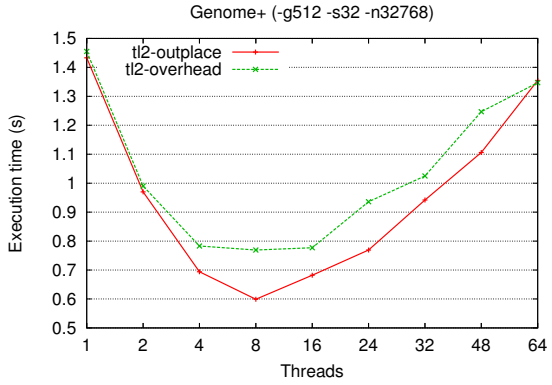

KMeans-low+ (-m40 -n40 -t0.05 -irandom-n16384-d24-c16)
tl2-overhead
Overhead (%)
Threads
23%
20%
13%
15%
9%
3%
-5%
17%
15%

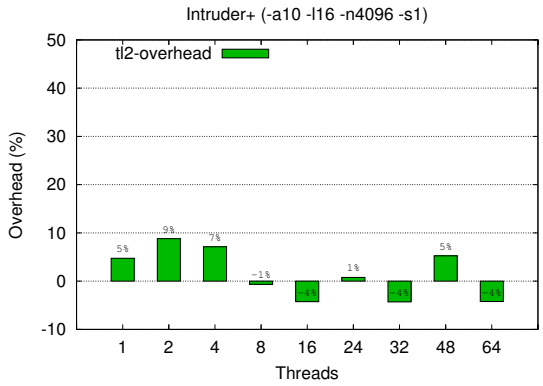

KMeans-low+ (-m40 -n40 -t0.05 -irandom-n16384-d24-c16)
tl2-outplace
tl2-overhead
Execution time (s)
Threads

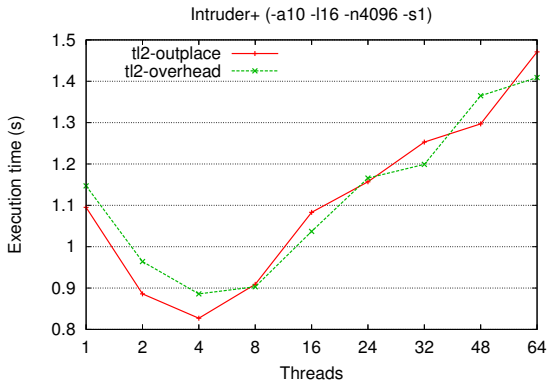

KMeans-high+ (-m15 -n15 -t0.05 -irandom-n16384-d24-c16)
tl2-overhead
Overhead (%)
Threads
12%
11%
6%
10%
4%
-2%
3%
5%
12%

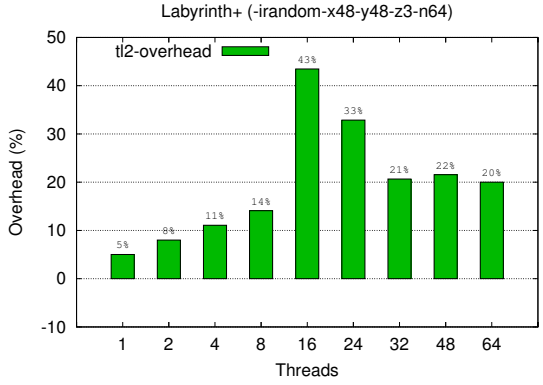

KMeans-high+ (-m15 -n15 -t0.05 -irandom-n16384-d24-c16)
tl2-outplace
tl2-overhead
Execution time (s)
Threads

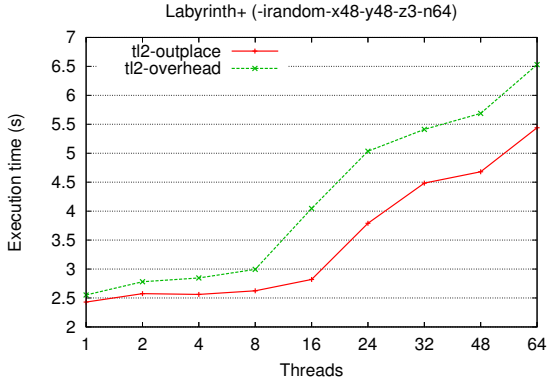

Genome+ (-g512 -s32 -n32768)
tl2-overhead
Overhead (%)
Threads
2%
2%
13%
28%
14%
22%
9%
13%
-1%

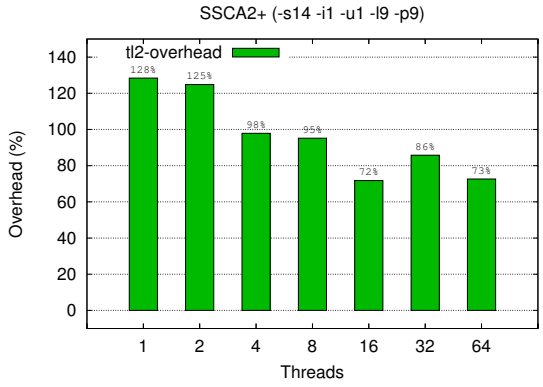

Genome+ (-g512 -s32 -n32768)
tl2-outplace
tl2-overhead
Execution time (s)
Threads

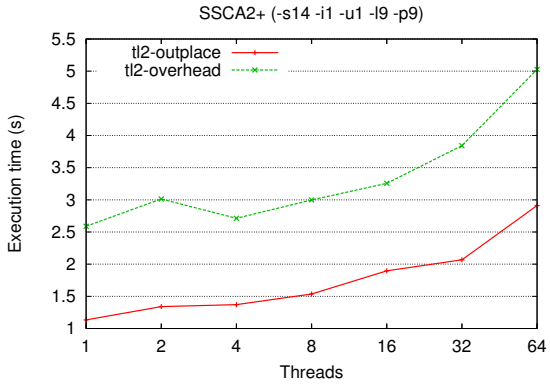






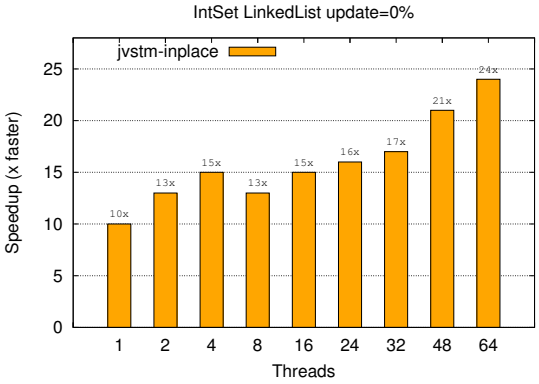


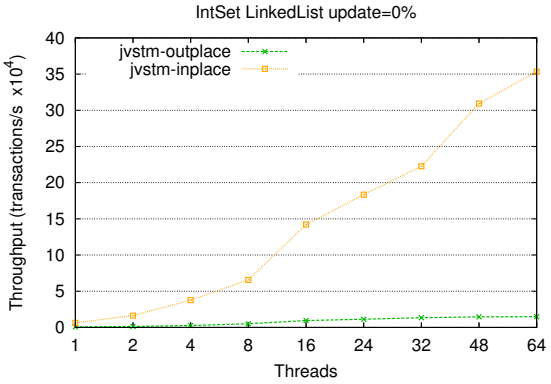


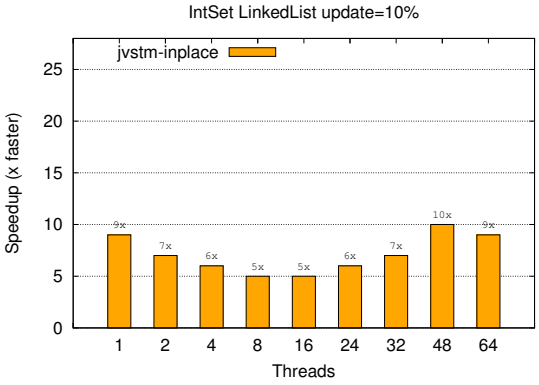


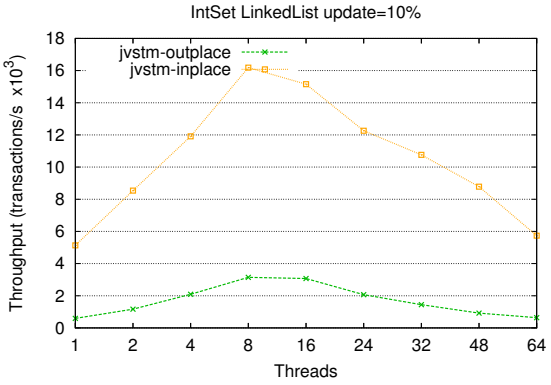
A.2 JVSTM-Inplace Speedup

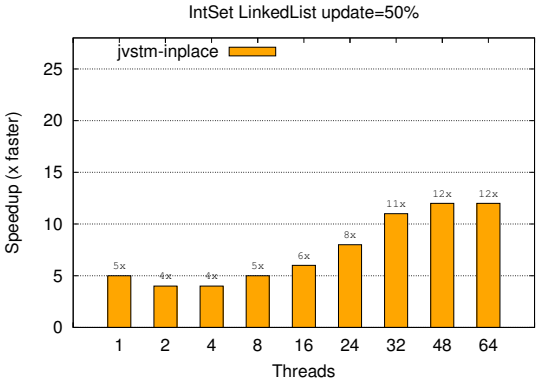
In this appendix we present the detailed results of Section 5.4.3 from comparing the JVSTM algorithm, as implemented in the original Deuce framework, and the JVSTM algorithm implemented using the in-place extension.

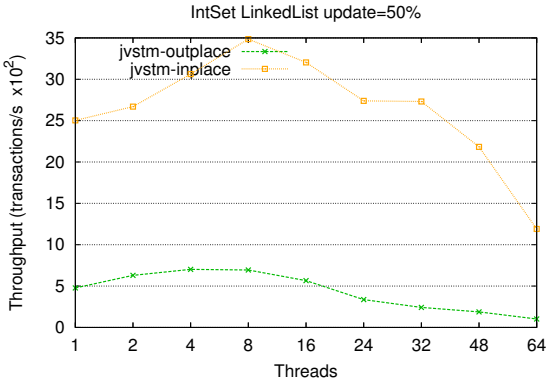

IntSet LinkedList update=0%
jvstm-inplace
Speedup (x faster)
Threads
10x
13x
15x
13x
15x
16x
17x
21x
24x

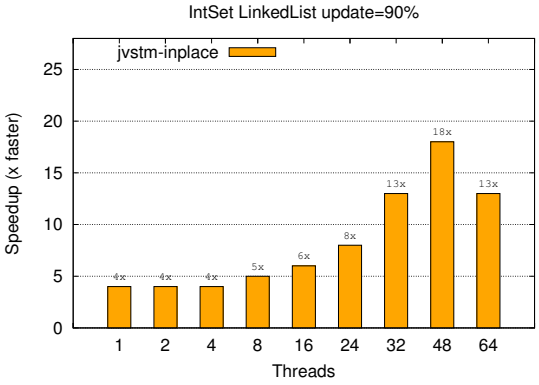

IntSet LinkedList update=0%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^4)
Threads

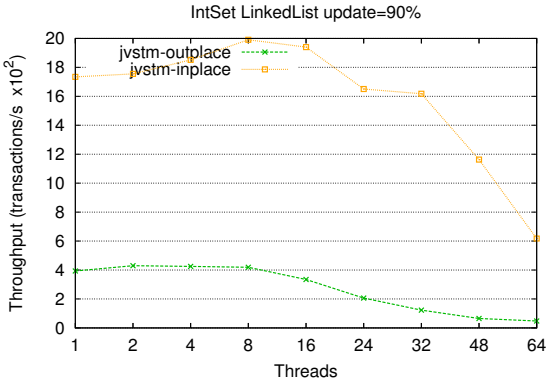

IntSet LinkedList update=10%
jvstm-inplace
Speedup (x faster)
Threads
9x
7x
6x
5x
5x
6x
7x
10x
9x

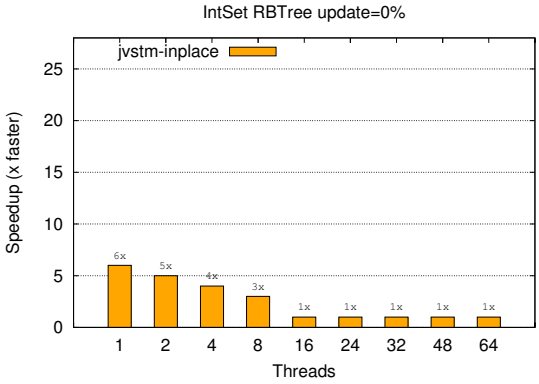

IntSet LinkedList update=10%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^3)
Threads

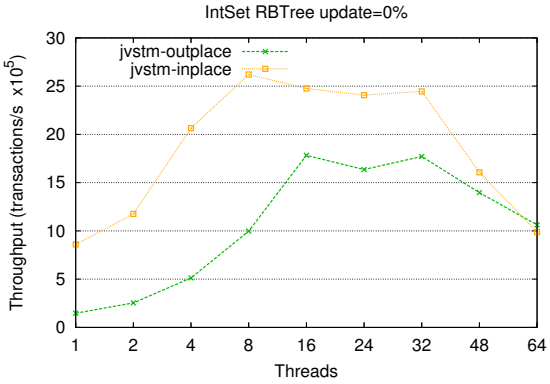

IntSet LinkedList update=50%
jvstm-inplace
Speedup (x faster)
Threads
5x
4x
4x
5x
6x
8x
11x
12x
12x

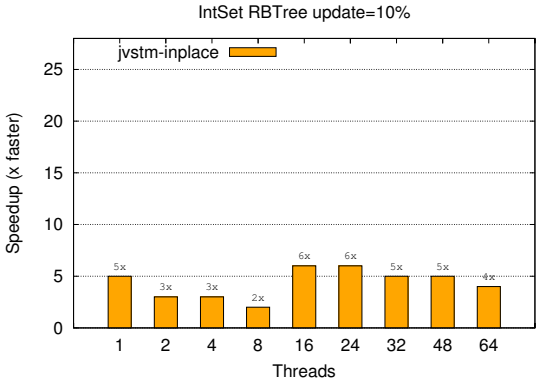

IntSet LinkedList update=50%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^2)
Threads

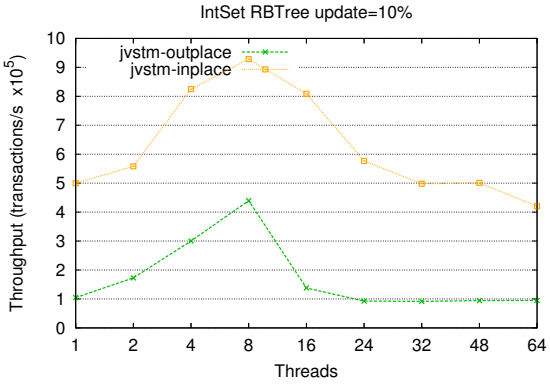

IntSet LinkedList update=90%
jvstm-inplace
Speedup (x faster)
Threads
4x
4x
4x
5x
6x
8x
13x
18x
13x

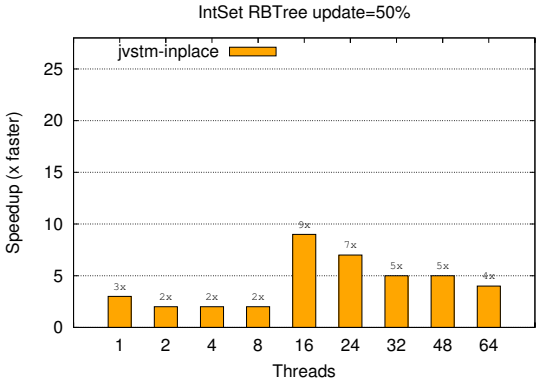

IntSet LinkedList update=90%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^2)
Threads

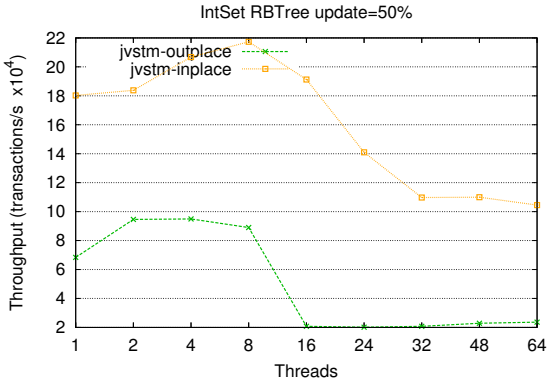

IntSet RBTree update=0%
jvstm-inplace
Speedup (x faster)
Threads
6x
5x
4x
3x
1x
1x
1x
1x
1x

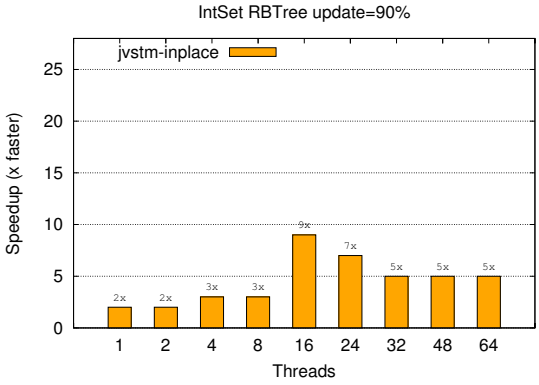

IntSet RBTree update=0%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^5)
Threads

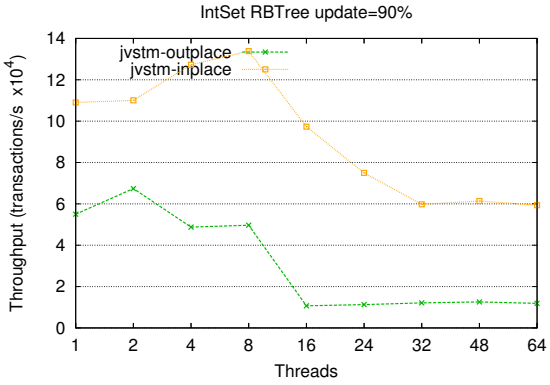

IntSet RBTree update=10%
jvstm-inplace
Speedup (x faster)
Threads
5x
3x
3x
2x
6x
6x
5x
5x
4x

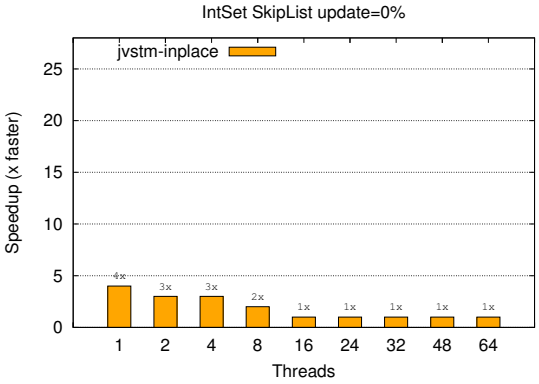

IntSet RBTree update=10%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^5)
Threads

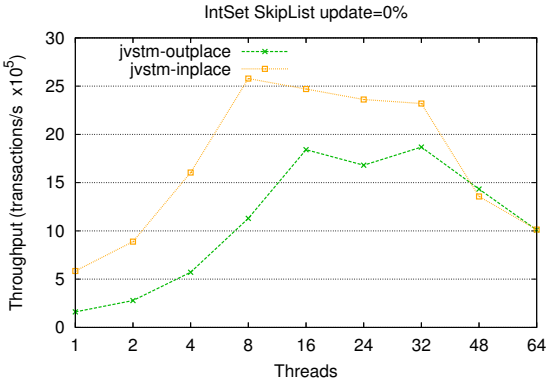

IntSet RBTree update=50%
jvstm-inplace
Speedup (x faster)
Threads
3x
2x
2x
2x
9x
7x
5x
5x
4x

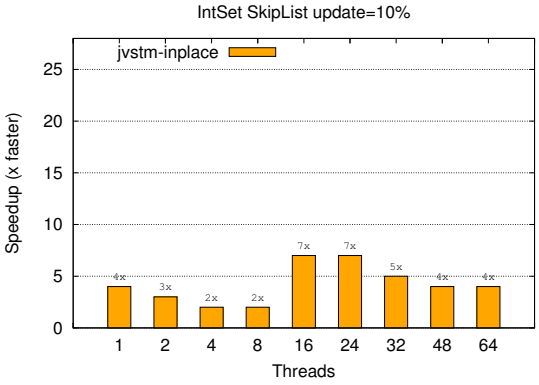

IntSet RBTree update=50%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^4)
Threads

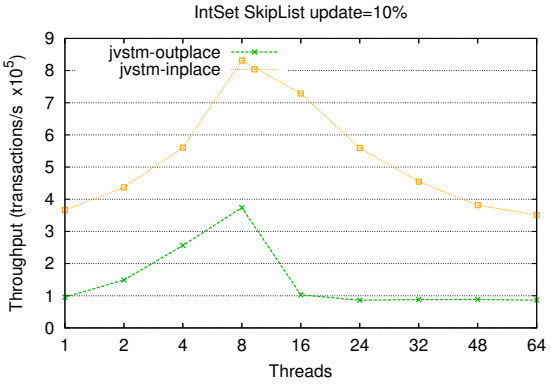

IntSet RBTree update=90%
jvstm-inplace
Speedup (x faster)
Threads
2x
2x
3x
3x
9x
7x
5x
5x
5x

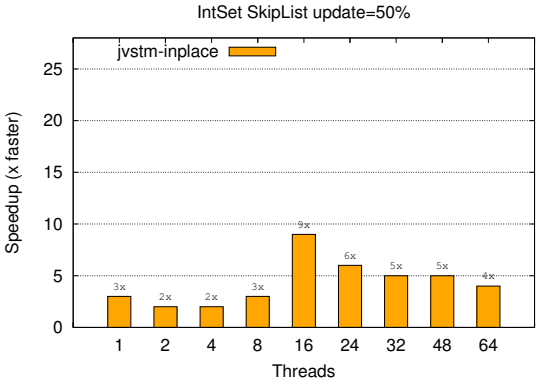

IntSet RBTree update=90%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^4)
Threads

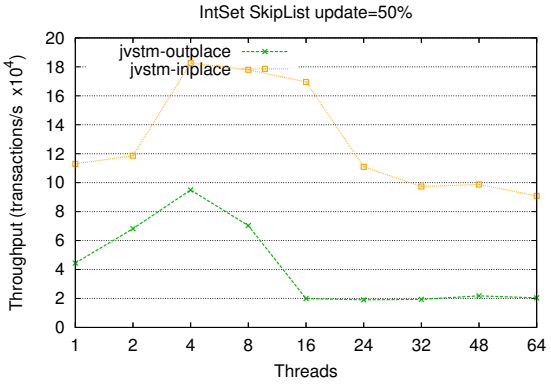

IntSet SkipList update=0%
jvstm-inplace
Speedup (x faster)
Threads
4x
3x
3x
2x
1x
1x
1x
1x
1x

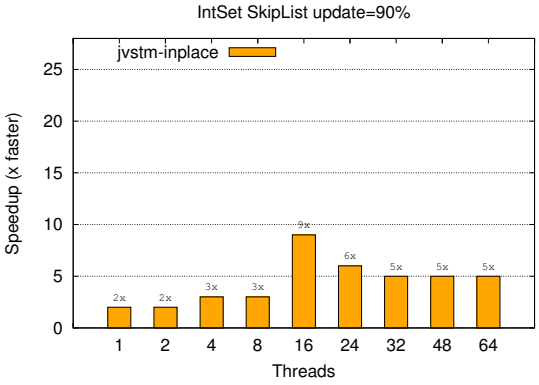

IntSet SkipList update=0%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^5)
Threads

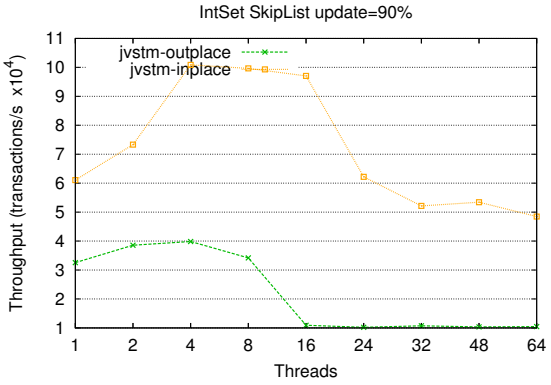

IntSet SkipList update=10%
jvstm-inplace
Speedup (x faster)
Threads
4x
3x
2x
2x
7x
7x
5x
4x
4x

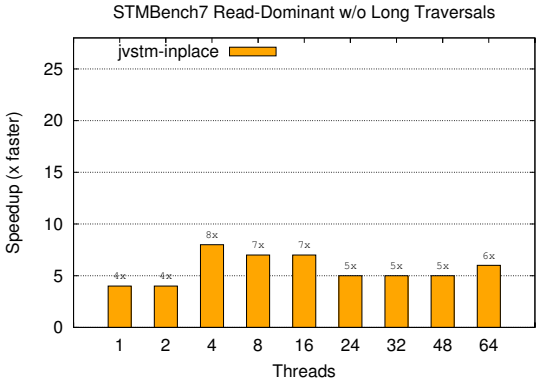

IntSet SkipList update=10%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^5)
Threads

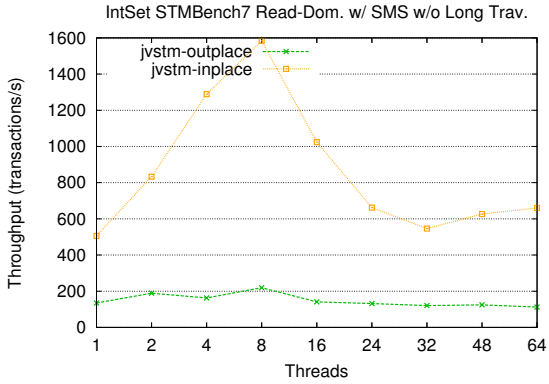

IntSet SkipList update=50%
jvstm-inplace
Speedup (x faster)
Threads
3x
2x
2x
3x
9x
6x
5x
5x
4x

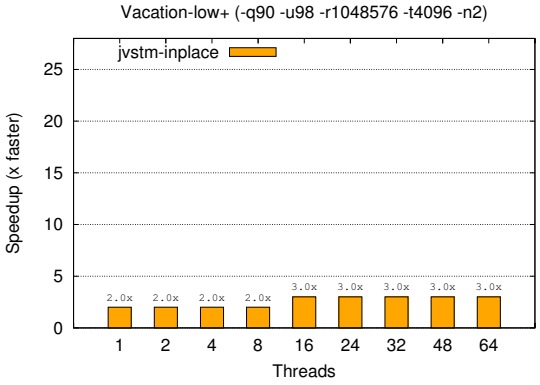

IntSet SkipList update=50%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^4)
Threads

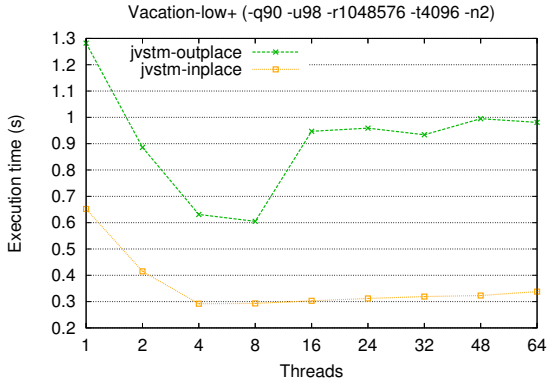

IntSet SkipList update=90%
jvstm-inplace
Speedup (x faster)
Threads
2x
2x
3x
3x
9x
6x
5x
5x
5x

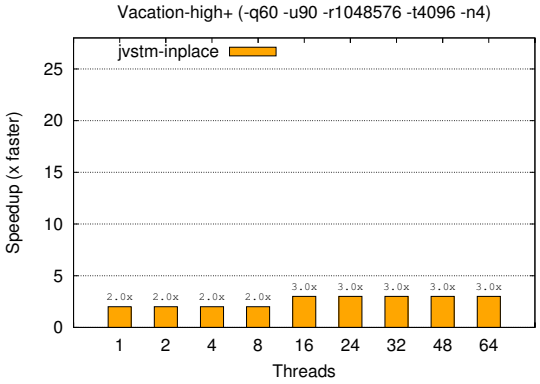

IntSet SkipList update=90%
jvstm-outplace
jvstm-inplace
Throughput (transactions/s x10^4)
Threads

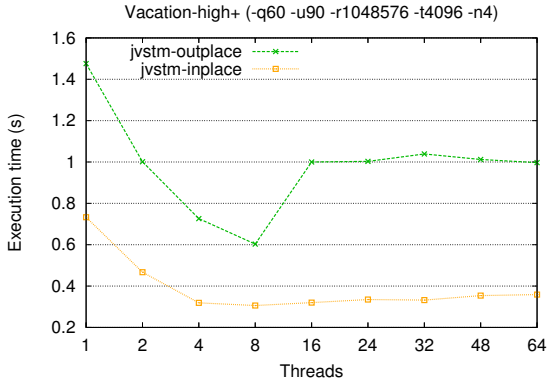

STMBench7 Read-Dominant w/o Long Traversals
jvstm-inplace
Speedup (x faster)
Threads
4x
4x
8x
7x
7x
5x
5x
5x
6x

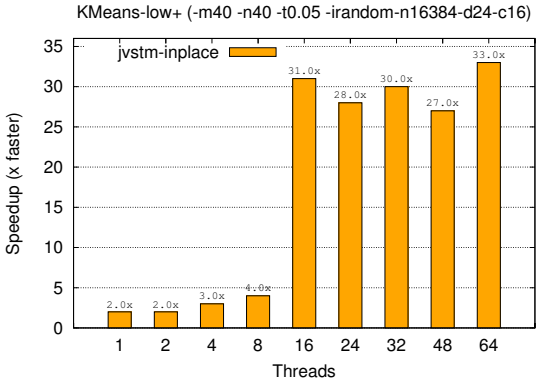

IntSet STMBench7 Read-Dom. w/ SMS w/o Long Trav.
jvstm-outplace
jvstm-inplace
Throughput (transactions/s)
Threads

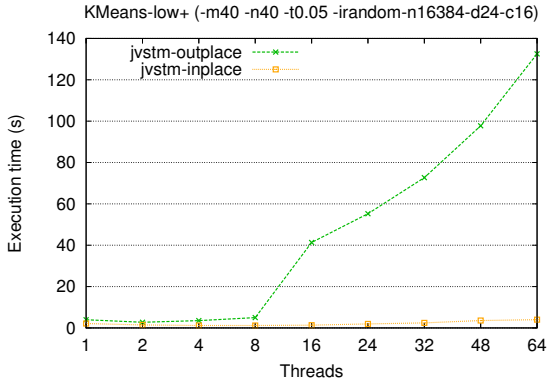

Vacation-low+ (-q90 -u98 -r1048576 -t4096 -n2)
jvstm-inplace
Speedup (x faster)
Threads
2.0x
2.0x
2.0x
2.0x
3.0x
3.0x
3.0x
3.0x
3.0x

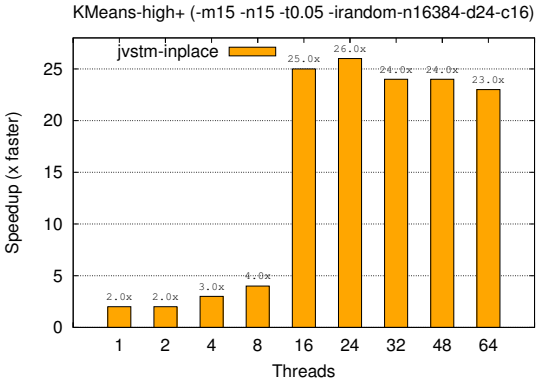

Vacation-low+ (-q90 -u98 -r1048576 -t4096 -n2)
jvstm-outplace
jvstm-inplace
Execution time (s)
Threads

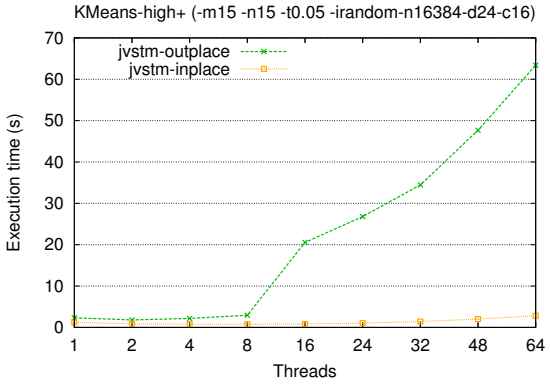

Vacation-high+ (-q60 -u90 -r1048576 -t4096 -n4)
jvstm-inplace
Speedup (x faster)
Threads
2.0x
2.0x
2.0x
2.0x
3.0x
3.0x
3.0x
3.0x
3.0x

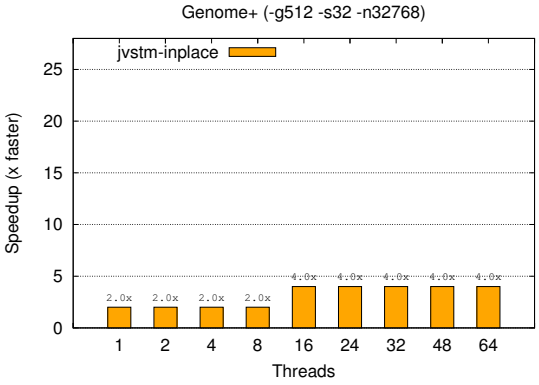

Vacation-high+ (-q60 -u90 -r1048576 -t4096 -n4)
jvstm-outplace
jvstm-inplace
Execution time (s)
Threads

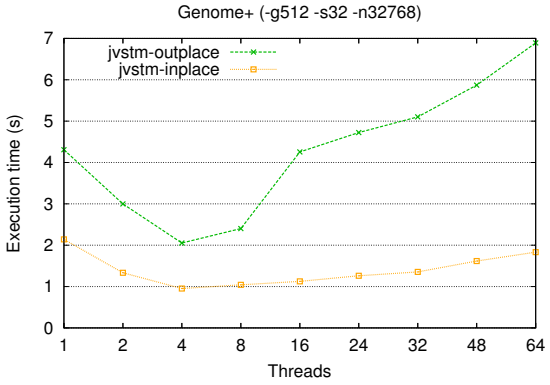

KMeans-low+ (-m40 -n40 -t0.05 -irandom-n16384-d24-c16)
jvstm-inplace
Speedup (x faster)
Threads
2.0x
2.0x
3.0x
4.0x
31.0x
28.0x
30.0x
27.0x
33.0x

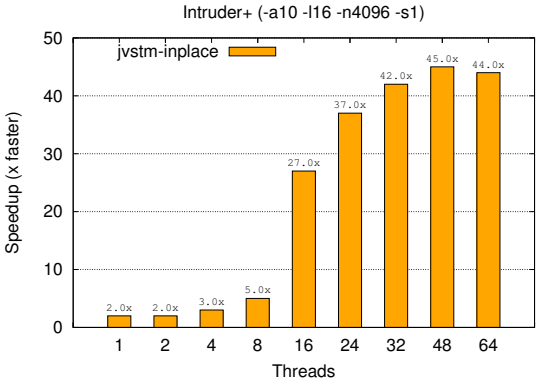

KMeans-low+ (-m40 -n40 -t0.05 -irandom-n16384-d24-c16)
jvstm-outplace
jvstm-inplace
Execution time (s)
Threads

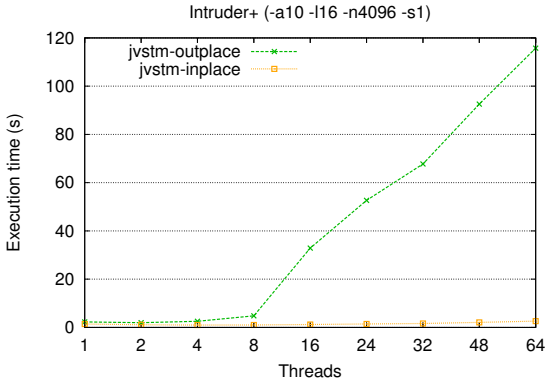

KMeans-high+ (-m15 -n15 -t0.05 -irandom-n16384-d24-c16)
jvstm-inplace
Speedup (x faster)
Threads
2.0x
2.0x
3.0x
4.0x
25.0x
26.0x
24.0x
24.0x
23.0x

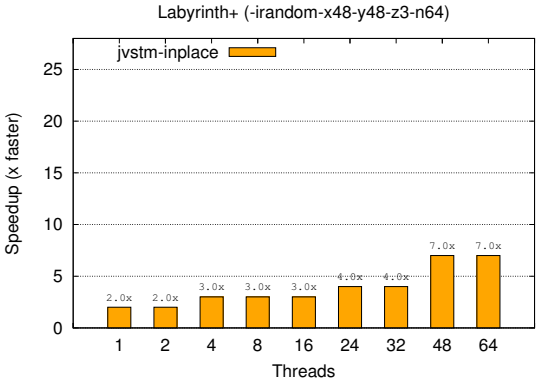

KMeans-high+ (-m15 -n15 -t0.05 -irandom-n16384-d24-c16)
jvstm-outplace
jvstm-inplace
Execution time (s)
Threads

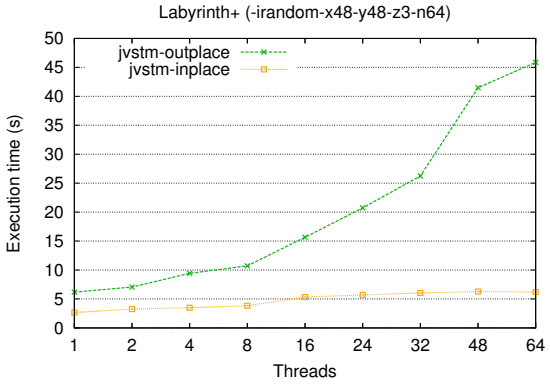

Genome+ (-g512 -s32 -n32768)
jvstm-inplace
Speedup (x faster)
Threads
2.0x
2.0x
2.0x
2.0x
4.0x
4.0x
4.0x
4.0x
4.0x

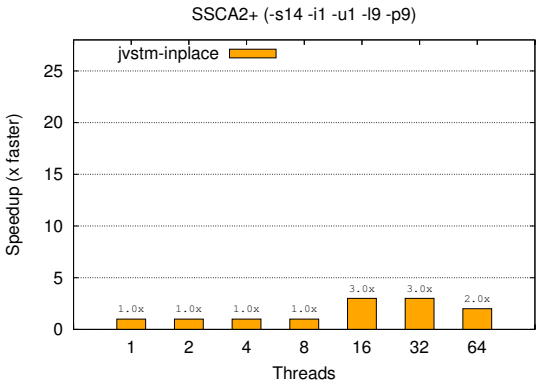

Genome+ (-g512 -s32 -n32768)
jvstm-outplace
jvstm-inplace
Execution time (s)
Threads

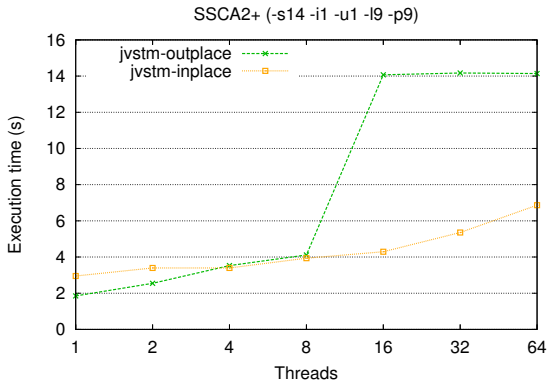

Intruder+ (-a10 -l16 -n4096 -s1)
jvstm-inplace
Speedup (x faster)
Threads
2.0x
2.0x
3.0x
5.0x
27.0x
37.0x
42.0x
45.0x
44.0x


Intruder+ (-a10 -l16 -n4096 -s1)
jvstm-outplace
jvstm-inplace
Execution time (s)
Threads


Labyrinth+ (-irandom-x48-y48-z3-n64)
jvstm-inplace
Speedup (x faster)
Threads
2.0x
2.0x
3.0x
3.0x
3.0x
4.0x
4.0x
7.0x
7.0x


Labyrinth+ (-irandom-x48-y48-z3-n64)
jvstm-outplace
jvstm-inplace
Execution time (s)
Threads


SSCA2+ (-s14 -i1 -u1 -l9 -p9)
jvstm-inplace
Speedup (x faster)
25
20
15
10
5
0
1.0x
1.0x
1.0x
1.0x
3.0x
3.0x
2.0x
1
2
4
8
16
32
64
Threads


SSCA2+ (-s14 -i1 -u1 -l9 -p9)
jvstm-outplace
jvstm-inplace
Execution time (s)
16
14
12
10
8
6
4
2
0
1
2
4
8
16
32
64
Threads