ATOM:
Automatic Transaction-Oriented Memoization

Hugo Jorge Beja Rito

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri
Presidente: Doutor José Carlos Martins Delgado
Orientador: Doutor João Manuel Pinheiro Cachopo
Co-Orientador: Doutor António Paulo Teles de Menezes Correia Leitão
Vogais: Doutor João Carlos Serrenho Dias Pereira

Outubro 2009
Acknowledgements

First and foremost, I would like to thank my adviser Professor João Cachopo for all the support he gave me throughout this work. Our discussions helped me define not only the purpose of this dissertation, but also many of the little details of my proposal. I know that without his opinion, feedback, and reviews this dissertation’s completion was not possible.

A very special thanks to the members of the ESW group of INESC-ID for all the fruitful discussions we had at our weekly meetings. Individualizing, I would like to thank Sérgio Fernandes for helping me on the technical details of the JVSTM, Fénix Framework, TPCW benchmark, and for the contribution on the STM Bench7 benchmark code. I would thank Ivo Anjo for all the help he gave me, specially with bytecode manipulation, and Luís Pina for our discussions that helped me clarify my ideas.

I would like to thank Professor António Leitão for helping me to define the best way to present my work.

Finally, I would like to thank my family for all the support they gave me. Specially, I would like to thank my father for driving me to Lisbon and for having the worst work hours in the world, “allowing” me to work from 7:30 to 18:30 every day. Also, I would like to thank my mother for waking up at 6:30 in the morning just to make my breakfast and my brother for reminding me how good it is not having work to do.

This work was partially funded by an FCT research grant in the context of the Pastramy project (PTDC/EIA/72405/2006).

Lisboa, October 22, 2009
Hugo Jorge Beja Rito
Resumo

Nas últimas décadas o paradigma de programação orientada a objectos evoluiu de tal forma que actualmente é a abordagem preferencial para o desenvolvimento de aplicações informáticas. Neste paradigma de programação os objectos mantêm estado interno que influencia o seu comportamento e que, em geral, muda ao longo do tempo.

Esta característica diferenciadora da programação orientada a objectos dificulta a aplicação de técnicas utilizadas nos demais paradigmas de programação. Um destes exemplos é a memoization, uma técnica de optimização bem conhecida, e muito utilizada em programação funcional, para melhorar o desempenho de um programa de uma forma transparente e sem alterar a sua semântica.

A dificuldade de aplicar memoization em programação imperativa orientada a objectos explica-se pelo comportamento de um método poder depender de estado partilhado ou provocar alterações visíveis no estado global do sistema. Algo que não acontece em programação funcional.

Nesta dissertação eu proponho uma memoization estendida que, aproveitando o suporte oferecido pelas memórias transaccionais por software (STM), elimina grande parte das limitações da memoization tradicional. Para tal introduzo o ATOM, o primeiro sistema automático de memoization a satisfazer as necessidades da programação imperativa orientada a objectos. Eu exponho como a memoization pode ser aplicada quase sem custos em sistemas que já utilizem uma STM, e demonstro como esta sinergia pode ser particularmente útil em contextos transaccionais.

Para validar a minha tese aplico memoization na benchmark STMBench7. A versão da benchmark com memoization mostra uma melhoria até 14 vezes no throughput do sistema para um workload dominado por leituras.
Abstract

Over the last decades the object-oriented paradigm evolved to the point that, nowadays, it is the preferred approach for software development. In this programming paradigm, objects maintain internal state that influence the objects’ behaviors and, in general, this state changes over time.

This differentiating characteristics of object-oriented programs difficult the application of techniques commonly used in other programming paradigms. One such example is memoization, a well-known technique for improving the performance of pure functional programs in a transparent way and without changing their semantics.

The difficulty of applying memoization in object-oriented imperative programs results, precisely, from the fact that a method’s behavior may depend on shared state values or produce side-effects. Something that does not what happen in functional contexts.

In this dissertation, I propose an extended memoization approach that builds on the support provided by a Software Transactional Memory (STM) to remove many of the limitations of traditional memoization. I argue that my extended automatic memoization system (ATOM) is the first to suit the needs of object-oriented programming. I show how memoization can be implemented almost for free in systems that use an STM, and present the reasons why this synergy can be particularly useful in transactional contexts.

I validate the usefulness of memoizing object-oriented programs by applying memoization to the STMBench7 benchmark, a standard benchmark developed for evaluating STM implementations. The memoized version of the benchmark shows up to a 14-fold increase in the throughput for a read-dominated workload.
Palavras Chave

Keywords

Memoization
Cache de funções
Programação imperativa orientada a objectos
Memória transaccional por software

Memoization
Function caching
Object-oriented imperative programming
Software transactional memory
# Contents

1 Introduction ................................................. 1
   1.1 Software Development ................................... 1
   1.2 Functional and Non-Functional Requirements ................. 3
      1.2.1 Performance ........................................ 3
   1.3 Object-Oriented Paradigm ................................ 4
   1.4 Thesis Statement ......................................... 4
   1.5 Notation ................................................... 5
   1.6 Outline of the Dissertation ............................... 5

2 Motivation and Problem Statement .......................... 7
   2.1 Memoization .............................................. 7
   2.2 Example of an Application in the Banking Domain ......... 11
      2.2.1 Banking Application Functionality ................. 11
   2.3 The Pitfalls of Applying Memoization in Object-Oriented Imperative Programs ....... 12
      2.3.1 Shared State Reads ................................. 12
      2.3.2 Side-Effects ........................................ 13
   2.4 Problem Statement ........................................ 15
   2.5 Summary .................................................. 16

3 Related Work ................................................. 19
   3.1 Incremental Computation ................................ 19
   3.2 Dynamic Programming Style ............................... 21
   3.3 Automatic Memoization Tools ............................. 22
3.4 Memory Management ......................................................... 22
3.5 Precise Dependencies ..................................................... 23
3.6 Identifying Side-Effects ................................................... 24
   3.6.1 Programmer-based Solutions ........................................ 25
   3.6.2 Static Purity Analysis ................................................. 26
   3.6.3 Dynamic Purity Analysis ............................................. 26
3.7 Shared State Reads ....................................................... 27
3.8 Summary ................................................................. 28

4 Solution: The ATOM System .............................................. 31
   4.1 Software Transactional Memory ....................................... 31
      4.1.1 Introduction to Software Transactional Memory .............. 32
      4.1.2 The Java Versioned STM ......................................... 32
   4.2 Extending the Applicability of Memoization ....................... 33
      4.2.1 Finding the Relevant State .................................... 34
      4.2.2 Identifying Side-Effects ....................................... 36
      4.2.3 Proposed Solution ............................................... 37
   4.3 The ATOM System ...................................................... 39
      4.3.1 The Rationale for the Memoization System ..................... 39
      4.3.2 The ATOM API .................................................... 40
      4.3.3 The MemoCache Class ............................................. 41
      4.3.4 Implementation of the Memoization Cache ..................... 42
      4.3.5 Granularity of the Read Set ................................... 46
      4.3.6 Memory Management in the MemoCache ....................... 46
      4.3.7 Extending JVSTM Transactions ................................ 47
   4.4 The Memoization Advisory System ................................ 50
   4.5 Memoizing Transactions .............................................. 51
   4.6 Summary ............................................................... 53
5 Validation

5.1 Case study: the STMBench7 benchmark

5.1.1 The STMBench7 Data Structure

5.1.2 The STMBench7 Operations

5.1.3 The STMBench7 Execution Setting

5.1.4 The STMBench7 Throughput Results: Multi-Threaded

5.1.5 The STMBench7 Throughput Results: Single-Threaded

5.2 Summary

6 Conclusions

6.1 Main Contributions

6.2 Limitations

6.2.1 Local Instances of Transactional Objects

6.2.2 Input/Output

6.3 Future Research

6.3.1 Read-Set Invalidation Strategy

6.3.2 Runtime Choice of Methods to Memoize

6.3.3 Transaction Reexecution
List of Figures

2.1 Computation-tree of $\text{fib}(5)$ ................................. 8

2.2 Side-by-side comparison of Fibonacci function’s execution flow for both the unmemoized and memoized version. ................................. 10

2.3 Implementation-level class diagram for the banking application’s domain model. ........ 12

4.1 Graphical representation of a versioned box. .................................................. 33

4.2 UML object diagram showing the state of the banking system after a client opens two distinct accounts with initial balance of 400 and 600, respectively. ................................. 35

4.3 Read-set of the method $\text{getTotalBalance}$ if executed with the system state depicted in Figure 4.2. .................................................. 35

4.4 State of the banking system after the bank’s only client opens a third account with initial balance of 500. .................................................. 36

4.5 Read set of the method $\text{getTotalBalance}$ if executed with the system state depicted in Figure 4.4. .................................................. 36

4.6 Execution flow of a memoized method using my memoization proposal. ................ 38

4.7 Organization of the method cache .................................................. 43

4.8 The hierarchy of transactions of the JVSTM and the ATOM. .................................. 48

5.1 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-dominated workload with all long read-write traversals and structural modifications disabled. .................................................. 60

5.2 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-dominated workload with all long traversals disabled. ................. 60

5.3 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-dominated workload with all long traversals and structural modifications disabled. .................................................. 61
5.4 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-write workload with all long read-write traversals and structural modifications disabled. .................................................. 63

5.5 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-write workload with all long traversals disabled. ................. 63

5.6 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-write workload with all long traversals and structural modifications disabled. .................................................. 64

5.7 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a write-dominated workload with all long read-write traversals and structural modifications disabled. .................................................. 65

5.8 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a write-dominated workload with all long traversals disabled. ....... 66

5.9 Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a write-dominated workload with all long traversals and structural modifications disabled. .................................................. 66
List of Tables

5.1 Output of the memo advisory tool for the STMBench7 benchmark with 14 memoized operations. ......................................................... 58
5.2 The results of the STMBench7 benchmark with all long read-write traversals and structural modifications disabled and a read-dominated workload. ........................................ 59
5.3 The results of the STMBench7 benchmark with all long traversals disabled and a read-dominated workload. ........................................ 59
5.4 The results of the STMBench7 benchmark with all long traversals and structural modifications disabled and a read-dominated workload. ........................................ 59
5.5 The results of the STMBench7 benchmark with all long read-write traversals and structural modifications disabled and a read-write workload. ........................................ 62
5.6 The results of the STMBench7 benchmark with all long traversals disabled and a read-write workload. ........................................ 62
5.7 The results of the STMBench7 benchmark with all long traversals and structural modifications disabled and a read-write workload. ........................................ 62
5.8 The results of the STMBench7 benchmark with all long read-write traversals and structural modifications disabled and a write-dominated workload. ........................................ 64
5.9 The results of the STMBench7 benchmark with all long traversals disabled and a write-dominated workload. ........................................ 65
5.10 The results of the STMBench7 benchmark with all long traversals and structural modifications disabled and a write-dominated workload. ........................................ 65
5.11 The results of the STMBench7 benchmark with all long read-write traversals and structural modifications disabled. ........................................ 67
5.12 The results of the STMBench7 benchmark with all long traversals disabled. ........................................ 67
5.13 The results of the STMBench7 benchmark with all long traversals and structural modifications disabled. ........................................ 68
Listings

2.1 Recursive implementation in Java of the Fibonacci function. .......................... 8
2.2 Memoized version of the Fibonacci function originally described in Listing 2.1. .... 9
2.3 Java implementation of the class Client. ......................................................... 13
2.4 Java implementation of the class Account, with the basic methods getBalance, deposit, and withdraw. ......................................................... 14
2.5 Java implementation of the Bank class. ......................................................... 15
3.1 Java implementation of the Fibonacci function using a bottom-up approach, as it is advocated by the dynamic programming style. ......................... 21
3.2 Simple function. .......................................................................................... 24
4.1 JVSTM-based implementation of the Client class from the banking domain example. .. 35
4.2 Use of the annotation Memo to memoize the method getTotalBalance. .......... 40
4.3 The ATOM's version of the getTotalBalance method from the banking domain example, originally shown in Listing 2.3. ........................................ 42
4.4 Skeleton of the class MemoCache. ................................................................. 42
4.5 Implementation of the method compareState for the VersionRelevantState class. .... 44
4.6 Implementation of the method compareState for the ValueRelevantState class. .... 45
4.7 Implementation of the method compareState for the IdentityRelevantState class. .... 46
4.8 Implementation of the Memo annotation. ....................................................... 46
4.9 Output of the MemoAdvisor with three methods of the STMBench7 benchmark memoized. 52
List of Abbreviations

API Application Programming Interface
CAD Computer Aided Design
DDG Dynamic Dependence Graph
HTM Hardware Transactional Memory
JVM Java Virtual Machine
STM Software Transactional Memory
TM Transactional Memory
UML Unified Modeling Language
Chapter 1

Introduction

The research work that I detail in this dissertation is about software development and how we can improve the performance of applications with a minimum of programmers’ intervention. Mainly, I address the problem of applying memoization to imperative object-oriented applications where methods are not functionally pure, contributing with an STM-based automatic memoization tool implemented in Java that accounts for reads from and writes to shared state.

In this introductory chapter, I begin with a brief discussion about why building a software solution is still an arduous task, even with all the advances in programming languages and in how software development is done. Next, I discuss the ever growing importance of non-functional requirements in software development and why they represent additional effort for programmers who need to build a solution that must comply with all these system qualities. Taking into account these elements, I continue by presenting my thesis statement, complemented with a brief description of how I intend to validate my thesis. I continue by introducing some basic notation and terminology that will be used throughout this dissertation, and end this introductory chapter with the outline of the dissertation.

1.1 Software Development

Since the invention of the first general purpose computer in the mid of the 20th century, there is the need to build software applications to solve complex problems or offer better services. But what started in a selective academical environment for military ends, grew and expanded to the point that nowadays software is a driving force of modern economy: Software applications, small or big, can be found in a myriad of contexts that range from telecommunications to transports and even small house appliances such as microwave ovens or televisions. Software is everywhere.

This proliferation of software applications in our everyday life gave computer science a pivotal position in modern society. Thus, we may be tempted to think that software development is a well defined, predictable, and structured process, much like bridge building. But that is not the case and, when compared to other engineering such as civil engineering, software engineering is still very young and with large margin for improvement.
But what is software engineering? From the definition given in the IEEE Standard Glossary of Software Engineering Terminology [26]:

Software engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.

Thus, software engineering main concerns are software applications and how they are developed.

Software development has always been a complex, time-consuming, costly, and ultimately hard task. This fact was comprehensible when computer science was giving its first steps and programming languages were few and very limiting. Now, approximately half a century after the inception of the first computer applications, despite the fast evolution observed in this area of engineering, software development is still an arduous task.

This difficulty can be in part explained because current software applications grew in size, scope and complexity, when compared to what was done in the past, encompassing not only programmers but also many others from departments like marketing, quality assurance, and general management, for example. When we combine this growth with the fact that software development is not repetitive nor predictable—there is a significant amount of intellectual and personal influence in the process of building software applications—we start to understand why non-trivial systems are so difficult to build.

One way to improve how software development is done is to improve the methodology used. Currently there are several different approaches to software development, some are structured and business-oriented, whereas others are incremental and look at software as an evolving entity developed piece-by-piece.

Strictly from a business point of view, an improved development process may translate to lower production costs, shorter development time, and better risk management. Improvements in this process may also benefit clients and end users because the resulting applications will better fulfill their needs.

Yet, despite all the advantages that arise from how the process of software development is structured and conducted, I firmly believe that the success of this intellectual activity can be directly traced to how well programmers materialize clients’ and users’ intent.

Because often programmers do not have the appropriate abstractions to deal with the current demands of the market, to make this activity less of a burden we must concentrate our efforts on supplying programmers with more adequate tools, thus facilitating the way they code software solutions.

One way to improve software development is to propose a new computational model from scratch and a novel programming language with better abstractions that allow a more appropriate development. The problem with this solution is that, although in the long-term it can facilitate the development of software applications if accepted by most developers, at an early stage it means an additional learning and adaptation step that many programmers are not willing to take.

Knowing that programmers have a great resistance to change, the solution commonly used is to introduce in already existing programming languages new abstractions in the least disruptive way possible. This approach takes advantage of programmers’ familiarity with the development environment, which is the same, and adds only a few special words to the language that transparently implement new functionality.
I strongly believe that small non-disruptive additions, which do not change the underlying programming language but rather extend it or complement it, are the most effective ones. Hence, the extensions proposed in this dissertation were implemented in pure Java, allowing the use of all the standard Java tools that programmers are familiar with and integrating transparently with the standard Java syntax and pragmatics.

1.2 Functional and Non-Functional Requirements

From the smallest home-made computer program to the largest corporate system, all forms of software applications share a common goal: they exist solely to solve a problem, offer a service, or carry a well-defined task. So, software development may be seen as the process in charge of building software solutions that fulfill clients’ and users’ needs.

If in the past the biggest concern of programmers was to build applications that produced correct results, with the maturation of the software industry it is now clear that clients are not only interested in the correctness of the solution delivered by software providers. The end system needs to solve a particular problem and show a set of qualities not directly related to how the problem is solved, thus the need to distinguish between functional requirements and non-functional requirements.

Functional requirements describe the behavior of the system in terms of functionality, tasks, activities, or goals that it must achieve. Non-functional requirements, also known as system qualities, on the other hand, are properties or overall characteristics of the system that affect stakeholders’ degree of satisfaction with the solution. Security, usability, scalability, and performance are some examples of non-functional requirements.

1.2.1 Performance

With the advent of interactive systems, it is now common for users to submit requests and wait for the respective system response. Thus, the faster a request is processed and replied, the sooner users can continue to interact with the system. Because user satisfaction is directly tied, not only with the correctness of the service, but also with how fast the system is able to produce responses, performance is a driving factor of software development.

Performance is timing-related. As events are received and processed by the system, performance concerns with how long it takes the system to issue a response. The performance of a system may be assessed in a multitude of ways, but for this dissertation I am interested only in the throughput—number of transactions that a system is able to process per-second—and the response time—time elapsed between the arrival of an event and its respective response.

One of the oldest, and simplest, performance optimization technique is memoization, also known as function caching. Proposed in 1968 in the context of Artificial Intelligence [29], it is based on the idea that it is possible to speedup the execution of a function if we maintain a cache of previous computations and return results from that cache instead of computing them again.

Memoization is often used in functional programming to increase the performance of a program, but is seldom used in other programming paradigms, such as object-oriented imperative programming.
1.3 Object-Oriented Paradigm

Programmers, when faced with the challenge of coding an application, have an almost endless collection of programming languages to choose from, each with its share of advantages and drawbacks.

**Functional languages**, such as Lisp, SCHEME, and Haskell, are inspired by Alonzo Church’s lambda calculus, employing a computational model based on the recursive definition of functions. In its essence, a program can be thought of as a mathematical function that accepts an input and produces an output that is defined in terms of simpler functions through a refinement process.

**Imperative languages** are based on statements that change memory values, influencing subsequent operations. This model of computation through side-effects has always been largely used thanks to the success of languages such as C/C++ and, more recently, Java and C#.

The **object-oriented paradigm**, since its introduction in the 1960’s [13], has attained a prominent position in the industry, representing in the last few years the preferred approach for the development of large-scale software applications, thus justifying the effort to introduce memoization in such imperative systems.

1.4 Thesis Statement

This dissertation’s thesis is that it is possible to simplify the task of programming imperative object-oriented systems, through the addition of minimal, non-disruptive, and easy to use abstractions to current object-oriented programming languages. In particular, I claim that it is possible to aid programmers in the task of improving the performance of object-oriented imperative programs through an automatic memoization tool that uses the work already made by a software transactional memory runtime to register shared state accesses.

As a result of imperative programs offering different challenges than those of pure functional ones, it is of the most importance to extend memoization to encompass the particular characteristics of environments where methods are not pure functions. Namely, it is necessary to:

- Build the list of relevant values for the result of a method, when this list is not confined to the arguments supplied to the memoized method but contains also values retrieved from shared state.
- Dynamically detect side-effect-free methods which are safe to memoize.
- Define the semantics associated with the memoization of methods with side-effects.

Taking advantage of previous work on software transactional memory, I detail how each of these new elements can be incorporated into an automatic memoization tool. Furthermore, I give a detailed description on how each of these elements can be implemented in Java and demonstrate, by comparing with what is disallowed in traditional memoization, what are the benefits attained from this imperative extension of memoization.

Throughout this dissertation, I introduce a simple example in the banking domain that demonstrates why it is so difficult to apply memoization in imperative contexts without introducing semantic changes or
producing incorrect results. I then show how my automatic memoization system ensures correct system behavior and how it can be easily used by programmers.

To show that memoization can be seamlessly incorporated at the programming level, the implementation described in this dissertation requires minimal learning effort from programmers, while offering a simple and customizable interface.

Taking into account that memoizing each and every method in the system may be less effective than just optimizing a few, to further increase the appeal of memoization I describe a set of tools that collect per-method performance statistics with the ultimate goal of aiding programmers in their task of choosing which methods are profitable to memoize.

I present three caching policies to test values for equality, discuss their implementation details, relative merits and drawbacks, and show how programmers can easily choose which one to use in their programs in a per-method basis.

I explain how memoization can be introduced almost for free in systems that use already a software transactional memory to ensure the atomicity of operations, and discuss why this synergy between memoization and STMs can be particularly useful.

This proposal is backed-up by results collected using a standard and well-known benchmark to test the performance of concurrent systems.

1.5 Notation

Throughout this dissertation I will use source code examples to drive the discussion. Because there are many object-oriented languages, and to standardize the examples shown, I shall use always the Java language in listings.

The code shown will be as simple as possible, absent of all irrelevant details that do not contribute to the discussion at hand. Therefore, rarely there will be a complete Java class in favor of short and concise code fragments that range from the simplest where a single method is shown, to the more complex ones depicting a class with just a few of its members and slot fields.

The sequence “...” will be often used to mark pieces of code that are not relevant to the understanding of the fragment shown. Because all fragments in this dissertation will be of Java code, when not relevant, the usual access control modifiers will also be elided.

1.6 Outline of the Dissertation

This dissertation is divided in 6 chapters:

- **Introduction.** Introductory chapter of this dissertation, describes the subject of concern of my research work. Mainly, it addresses the problem of software development and the importance of non-functional requirements, or system qualities, in the construction of software solutions, with special attention to performance. Additionally, it presents my thesis statement, how it will be validated, and introduces some basic notation that will be used throughout this dissertation.
- **Motivation and Problem Statement.** The second chapter complements on the motivation given in the introductory chapter, introducing the concept of memoization and how it can be applied to improve the performance of software systems. This chapter continues by explaining the pitfalls and difficulties of applying memoization in imperative object-oriented programs, which is done using an example on the banking domain. Finally, it identifies the problem that this dissertation addresses.

- **Related Work.** Chapter 3 introduces a couple of performance enhancing alternatives to memoization, discusses the merits of automatic memoization tools, how it is possible to overcome memoization’s high memory consumption levels, and the limitations an overly restrictive cache-key may originate. But, despite all these related solutions, this chapter mainly concentrates on presenting current approaches that partly solve the problems of applying memoization in imperative object-oriented systems. In particular, this chapter concentrates on current tools that aid programmers on memoizing methods that are not pure functions, and the guarantees that they offer.

- **Solution: The ATOM System.** Chapter 4 of this dissertation introduces the concept of software transactional memory, discussing in detail a particular STM implementation—the Java Versioned STM. Building upon previous work on STMs, this chapter continues presenting how it is possible to extend the applicability of memoization to imperative object-oriented contexts. In particular, it shows how shared-state dependencies can be automatically captured and how side-effects can be detected or incorporated into memoization. Next, it presents the Automatic Transaction-Oriented Memoization system, its API, architecture, organization, and how it ensures flexibility and easiness of use. Finally, the chapter concludes introducing an advisory system that helps programmers choose which methods may benefit from memoization and which memoization strategy best fits the run-time behavior of memoized methods, in a per-method basis.

- **Validation.** To validate the ideas introduced and described throughout this dissertation, Chapter 5 assesses the performance speedup obtained by employing the ATOM system in an imperative object-oriented program, using a highly-customizable and standard benchmark developed for evaluating STM implementations.

- **Conclusions.** The final chapter of this dissertation summarizes the main contributions of this work, discussing future research paths that may help improve the attained results.
Chapter 2

Motivation and Problem Statement

In the previous chapter, I briefly mentioned that memoization is an optimization technique used to improve system’s performance and stated that it is seldom used in imperative object-oriented environments. However, I have not discussed how memoization boosts the performance of a system, and I have still to present the reasons that make the use of memoization difficult in object-oriented imperative programs.

So, in this chapter, I detail both subjects and highlight the reasons that led me to pursue this work. I begin, in Section 2.1, with a quick overview of memoization, describing what memoization is, which characteristics functions must have to be eligible for memoization, and how memoization may be implemented. Then, in Section 2.2, I introduce a simplified example of an object-oriented application in the bank domain that will help drive the discussion in Section 2.3, where I address the problems of applying memoization in imperative contexts.

I end this chapter by stating in Section 2.4 what are the specific problems that I am trying to solve with my work.

2.1 Memoization

When programming a function, if we have a calculated value that is used more than once throughout the computation, then it is better to calculate its value just once and store it in an accessible place, from where it will be fetched the next time that it is needed, rather than spend time calculating it every time that it is used.

This way we may speedup execution time by preventing repeated computations. The performance boost is directly influenced by the amount of work that is skipped: As the number of operations that are skipped grows, so does the obtained performance boost.

In the same way, a function, as a whole, that recalculates a value that it had previously calculated executes repeated computations. Therefore, executing this function is redundant if we remember its return value. I will use both terms—redundant and repeated—throughout this dissertation to classify functions or computations that, when executed, spend time calculating a previously calculated value.

But there are some cases where redundant computations represent not only unnecessary work, but also the sole reason why an algorithm behaves so poorly in terms of performance. To better illustrate this
public int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}

**Listing 2.1:** Recursive implementation in Java of the Fibonacci function.

**Figure 2.1:** Computation-tree of fib(5). Each node represents a function call and the enclosed number the respective value of parameter n.

problem, consider the code in Listing 2.1, which depicts a simple, classic, and merely academic example of repetition within a function call, thus being perfect to demonstrate the negative influence on system performance that extra and unnecessary work may have.

The Fibonacci function, as it is presented in Listing 2.1, is a doubly recursive mathematical function that generates a tree of function calls. As we can see in Figure 2.1, each function call generates two recursive calls, each of which, in turn, makes two more function calls and so on until the function is called with a number less or equal than 1.

This example uses a rather small number for the parameter n (n = 5) and it makes 14 recursive calls. Whereas with n equal to 6 and 7 we get 24 and 40 recursive calls, respectively. In fact, the running time of the Fibonacci function grows exponentially with increasing n.

The problem, of course, is that there are too many repeated computations slowing down the process. Centering our attentions back on Figure 2.1, we can clearly see that fib(0) and fib(2) are calculated three times, fib(3) is calculated twice, and fib(1) is calculated five times. Yet, each of these needs to be calculated only once, thanks to the fact that they always return the same value: 0, 1, 2, and 1, respectively.

Thus, one way to improve the performance of the Fibonacci function is to collect results as they are computed. Then, as we travel down the recursion-tree, if we are about to start a redundant computation, which can be recognized by looking at the value of n, we can skip it and return the previously calculated value. This way, whole branches can be cut down, thus saving time, resources, and transforming an
This is the idea behind memoization: Improve system performance by avoiding redundant computations. This goal can be fulfilled only if we are able to identify computations as redundant and if we can remember which value redundant computations produce when executed.

Memoization is a function-level optimization technique. It detects repeated computations by monitoring function calls and by associating with each memoized function a cache that maps a list of relevant values to the return value of the function. This list of relevant values contains all the values that are significant for obtaining a particular function’s result, so that if at least one of these values change the result of the function may be different.

Because in a functional context the result of a function is fully defined by the supplied arguments, the list of relevant values is in fact the list of arguments received as input by the memoized function. This way, a function call made with a list of arguments that is already stored in the cache is considered to be redundant, it is skipped, and the associated result is returned instead.

```java
Map<Integer, Integer> cache;

public int fib(int n) {
    int res = 0;
    if (cache.containsKey(n)) {
        res = cache.get(n);
    } else {
        res = memo$fib(n);
        cache.put(n, res);
    }
    return res;
}

private int memo$fib(int n) {
    /* Original body of fib */
}
```

**Listing 2.2:** Memoized version of the Fibonacci function originally described in Listing 2.1. A cache mapping an arguments-to-result association was added and fib was changed to consult the cache to decide if it should compute the value or retrieve it from the cache.

A memoized version of fib is shown in Listing 2.2, whereas Figure 2.2 compares side-by-side the execution flow of the unmemoized and the memoized version of the Fibonacci function.

Unmemoized functions usually take some values as arguments and return another value that is computed from those arguments, regardless of whether that same computation was already done before.

On the other hand, a memoized function first does a cache search to check whether it has already performed the asked computation. If a mapping is found, it returns the stored result to the caller. If it is the first time that such computation is done, the requested result is computed and, just before its return to the caller, a new mapping is added to the cache representing this new arguments-to-result association.

A performance boost happens every time it is faster to search the cache for a match than to reexecute
Figure 2.2: Side-by-side comparison of Fibonacci function’s execution flow for both the unmemoized and memoized version.
the original function, and, of course, there is a cache hit (meaning that the result stored in the cache is returned). This gain in performance must be sufficient to compensate for possible cache misses and the cost of constructing and managing cache entries.

With such behavior, memoization is a very appealing technique in systems where there is some kind of repetition, either within a function call, as in the recursive Fibonacci example (Listing 2.1), or over time. Repetition over time relates to a function, or set of functions, that, during system execution, are used frequently by other functions or by users that interact with the system to produce the same result value.

Given that in imperative contexts recursion is often replaced by iteration, we expect that repetition over time is where memoization will help improve the performance of object-oriented imperative programs. If a system has a subset of services that is used considerably more often than the rest, memoizing these services can yield a performance boost as long as they perform redundant computations.

2.2 Example of an Application in the Banking Domain

To discuss the problems of applying memoization in imperative object-oriented contexts, I shall use throughout this dissertation a simple example of an application from the banking domain. This choice is justified not by any particularity in the domain model or functionality of software banking systems, but essentially because it is a well known and easy to understand example, often used in the literature of object-oriented programming.

Although this bank example tries to mimic some of the functionality of banking software applications, it is still an oversimplification of what can be found in real banking systems. Despite the banking domain being a common and frequent one, there are enough alternatives and design choices that can lead to different implementations. Thus, I present an architecture that I deem as reasonable and simple enough not to adulterate the purpose of this kind of applications, though managing to evidence common coding patterns of object-oriented imperative programs.

2.2.1 Banking Application Functionality

The core functionality of any banking application is the management of bank clients and their respective accounts. For this particular example, suppose that the banking software just manipulates a couple of entities: clients and accounts.

Everyone with at least one bank account is considered a bank client. There is no limit on the number of accounts that a single client can have, but on the other hand, each account has a single owner, the bank client who opened it.

As usual, accounts have a current balance, which corresponds to some monetary amount in some particular monetary currency, and an interest rate. The interest rate is calculated according to the account’s balance and is used by the bank to periodically pay interest to its clients: Every thirty days, each account’s balance is increased by an amount that is calculated from the current interest rate and the previous balance.

Clients may deposit money to any account but can only issue withdrawals from accounts that they
own. These operations change the respective account’s current balance in accordance with the amount deposited or withdrawn. Clients may also query the bank for their total balance, that is, the sum of the balance of all the accounts that they own.

Figure 2.3: Implementation-level class diagram for the banking application’s domain model.

Figure 2.3 shows the UML diagram of the class structure that I chose to capture the functionality described in this section.

2.3 The Pitfalls of Applying Memoization in Object-Oriented Imperative Programs

In this section I show a possible object-oriented Java implementation of the bank system described in Section 2.2.1. This particular implementation shall be used to highlight the difficulties and the problems that arise when programmers try to apply traditional memoization to methods that read from shared system state (Section 2.3.1) or that produce side-effects (Section 2.3.2).

2.3.1 Shared State Reads

Consider the implementation of the Client class and the Account class shown in Listing 2.3 and in Listing 2.4, respectively.

Almost all of the methods of both classes are relatively simple and take constant time to execute, thus, theoretically, not profitable to memoize. The clear exception here is the method responsible for calculating a client’s total balance (getTotalBalance): This computed value is calculated by iterating over the set of accounts owned by a client and adding their current balance. Because the time the method getTotalBalance takes to compute the intended result grows linearly with the number of Account instances associated with the instance of Client on which the method is invoked on, we can try to improve its performance by memoizing it.

If we recall what was mentioned in Section 2.1, to memoize a method, we must augment it with a cache that associates a result to the list of all relevant values used to construct that result. So, what are the relevant values for the output of getTotalBalance?

Taking into account the definition of relevant values, and the intended behavior of the method getTotalBalance, we can state that the relevant values for this particular method includes the value of the accounts slot of the instance of Client class on which the total balance is being calculated on, as well as the value of slot balance of each Account instance contained within that accounts set.
From this example it is clear that the approach followed in functional programs, of using the supplied list of arguments as the cache-key, cannot be directly applied in imperative object-oriented programs, because the list of relevant values encompasses not the list of arguments supplied to the memoized method, but values retrieved from shared state.

Knowing that the list of arguments it is not suited to classify computations as redundant or not, a naive approach would be to extend the memo cache to include in the cache-key the value of the accounts slot and the balance of each Account instance contained within the accounts slot. In fact, in this simple example that may solve the problem. Yet, in the general case, of more complex methods where the set of fields accessed is much larger, depends on which execution path is taken or on what other methods are called, it is unfeasible to determine manually the correct set of fields to use in the cache-key.

This is one of the difficulties of applying memoization to object-oriented programs: Methods often depend on the state of other objects, many of which cannot be easily determined by code inspection alone.

### 2.3.2 Side-Effects

Consider now the method responsible for paying interest depicted in Listing 2.5.

As with `getTotalBalance`, the method `payInterest` cannot be memoized. But what this follow up example highlights is not the difficulty of constructing the list of relevant values for the result returned by `payInterest`. As a matter of fact, `payInterest` has no return value. It is the problem that arises when memoization prevents operations from executing.

As previously stated, every time there is a cache hit in a memoized method, memoization accelerates
class Account {
    long balance;
    double interestRate;

    Account(long balance) {
        setBalance(balance);
    }

    long getBalance() {
        return balance;
    }

    private void setBalance(long balance) {
        this.balance = balance;
    }

    double getInterestRate() {
        return interestRate;
    }

    void deposit(long amount) {
        setBalance(getBalance() + amount);
    }

    void withdraw(long amount) {
        setBalance(getBalance() - amount);
    }
}

Listing 2.4: Java implementation of the class Account, with the basic methods getBalance, deposit, and withdraw.

its execution by substituting the execution of the method by its return value. This principle, if applied to the method payInterest, may result in future calls to payInterest being skipped, thus preventing the state changing deposit operation from executing.

In fact, given that memoizing payInterest can be so harmful, and that it is even arguable that payInterest will ever execute under the same list of relevant values, thus producing a cache hit, the question that poses is: Why do we even want to memoize payInterest? The answer is simple: We do not. Yet, that does not mean that we should not care about payInterest being inadvertently memoized.

In this small example it is easy to guarantee that payInterest will never be memoized, because through code inspection we can conclude that no other method in the system calls payInterest. But, in more real cases, with deep and complex call-chains, it is not trivial to assure that a method’s execution that produces side-effects will never be skipped due to being called inside a memoized method. Even in this simple example we cannot assure that in future iterations of this banking system, new methods emerge and one of these methods is memoized even though it calls payInterest.

Again, methods that produce side-effects such as payInterest are common in object-oriented imperative programs.
public Bank {
    Set<Client> clients;
    Set<Account> accounts;

    void payInterest() {
        for (Account account : accounts) {
            int interest = account.getBalance() * account.getInterestRate();
            account.deposit(interest);
        }
    }
}

Listing 2.5: Java implementation of the Bank class. The bank has an unlimited number of clients and accounts, and is responsible for paying interests to its clients.

2.4 Problem Statement

Memoization is a well-known technique that has been successfully used in a myriad of functional contexts such as artificial intelligence systems [18], mathematical systems [19], or even software configuration management systems [23]. Yet, it is seldom used in object-oriented imperative programs.

This disparity in usage can be mainly explained by the fact that in functional environments all functions are pure, being very easy and always safe to memoize. A function is pure if its result value depends only on the values of the supplied arguments and so, it always evaluates to the same result value given the same argument values. Therefore, it is easy to apply memoization in pure functional programs because the list of relevant values is always limited to the list of arguments supplied to memoized functions, existing this way a very clear pattern in memoizing a function.

On the other hand, objects in object-oriented programs often have internal state that affects the result of their methods. This way, the relevant values for the outcome of a method often origin not only from the supplied arguments but also from that shared state. So, to benefit from memoization’s performance boost in imperative stateful contexts, programmers must endure the chore of identifying these hard to build relevant state dependencies, which must be incorporated into the cache-key.

The problem with shared state dependencies is twofold. First, it is difficult to identify shared state read operations and to register all the values read in those accesses. Second, it is not simple to store in the cache information regarding the expected shared values that constitute the relevant state and, more importantly, the locations from where those values were retrieved, so that, in future reexecutions, we can inspect those cached locations to assess if they still hold the same value.

In general, and without risking a semantic change, we can skip the execution of a method only if that method is referentially transparent. A method is considered to be referentially transparent if calling the method has the exact same effect as replacing the method call with its return value. So, memoization is not applicable to methods that are not deterministic or methods that have side-effects such as doing I/O or changing the program’s state.

The problem is that in imperative systems many computations have side-effects. Therefore, the applicability of memoization in such contexts may be confined to a small subset of the methods of the system. Memoizing this subset may not be enough to obtain more than marginal improvements in performance. Thus, how strict are the constraints that we impose on methods so that they are eligible
to memoization greatly influences the benefits extracted from memoization.

On the other hand, if we relax this prohibition, allowing methods that produce side-effects to be memoized, we extend the applicability of memoization to any method of an imperative system. The difficulty here is how to replicate the behavior of a memoized method that is not referentially transparent—that is, how to preserve the semantics of the program in the presence of memoized methods that may produce side-effects.

To summarize, the problem is that, when applying memoization in imperative contexts, programmers always face the following questions: Is there any possible execution path taken from the method that I want to memoize that will lead to a method that is not pure? If not now, in future versions of the code may that happen? If any of those questions is answered affirmatively then, there is the possibility that memoization will change the semantics of the program.

As briefly discussed in Section 2.1, programmers must be extra careful with the choices that they make, because these choices affect the performance boost obtained from memoization.

By virtue of memoization only speeding up redundant executions, it is obvious that if a method is never called with the same list of relevant values, memoization will only introduce overheads in the form of searching, storing, and managing cache entries. Even if a method is often called with the same list of relevant values, an improved performance happens only if it is faster to retrieve the value from the cache than to recalculate it.

Thus, effective use of memoization depends not only on finding which methods of the system will be repeatedly called with the same list of relevant values, but essentially on knowing which of them spend enough time executing to benefit from memoization.

2.5 Summary

All software systems must satisfy a list of requirements that dictate not only what the system must do but also how well. This list of requirements will ultimately influence the design and architecture of an application, thus being an ever present concern when developing software and an additional concern for programmers.

Memoization is an optimization technique that improves the performance of a system by trading computation time for memory consumption. This is accomplished by augmenting functions with a cache that allows functions to “remember” the result obtained when executed with some list of relevant values.

The boost in performance happens every time a subsequent function call with remembered inputs returns the cached result rather than recalculate it. Because this performance boost must be enough to compensate for the time spent on cache interactions, memoizing all the methods in a system may not be the best choice. But finding the best functions to memoize just by code inspection or by reasoning about the expected behavior of the system is a complex and ultimately time consuming task that should be done with the help of some more automatic tool.

Due to its simplicity to use and to understand by programmers, allied to a great potential in systems that spend a lot of time executing repeated work, memoization, when correctly applied, improves the performance of a system almost effortlessly, transparently, and without changing system semantics.
Despite being widely used in functional contexts, memoization is still seldom used in imperative contexts, such as in imperative object-oriented systems. This almost absence of memoization in imperative contexts relates to the fact that shared state accesses difficult or, in some cases, prohibit at all the use of memoization.

If a method during its execution uses values belonging to shared state, then these values must be incorporated in the cache-key and validated every time the method does a cache search. The problem is that it is not simple to capture this relevant state and it is not trivial to come up with a solution on how to represent the locations read by the method so that they can be validated, as well.

To worsen this situation, it is expected that many of the methods in an imperative system may produce side-effects when executed. Therefore, their execution cannot be skipped, because doing so would prevent changes to shared state from being made, thus changing the overall semantics of the system.

In conclusion, to apply memoization in imperative object-oriented contexts it must be easy for programmers to answer two questions: (1) which methods in the system I want to memoize because their execution will be accelerated with a cache?, and (2) which methods in the system I can memoize because their behavior will not be changed?
Chapter 3

Related Work

Section 2.1 introduced the concept of memoization and in Section 2.3 I demonstrated that, if we want to apply memoization in imperative contexts, we must deal with methods that produce side-effects and read values from shared state.

In this chapter I briefly discuss a couple of alternatives to memoization, but I concentrate mainly on current solutions to memoization’s problems underlying shared state reads and changes to shared memory, discussing the relative merits of each solution and why they still do not solve many of the limitations introduced throughout Chapter 2.

In Section 3.1, I introduce the concept of incremental computation, which is an optimization technique that tries to boost a function’s execution by limiting the amount of work that must be done upon a function call. Then, in Section 3.2, I present an alternative to memoization that prevents repeated work by decomposing complex problems into simpler ones and by reusing their results.

I continue by discussing, in Section 3.3, the merits of memoization tools that automatically and transparently, upon programmers request, transform a function into a memoized version of itself.

In sections 3.4 and 3.5 I center the discussion on the function cache, addressing the problem of memory management and the benefits of precise dependencies in the cache-key, respectively.

I conclude this chapter by presenting some proposals to address the unique characteristics of imperative object-oriented systems that difficult the use of memoization. Section 3.6 describes how to detect methods that may produce side-effects, whereas Section 3.7 describes some attempts to detect shared state dependencies or that partly allow memoization in methods that read values from shared mutable state.

3.1 Incremental Computation

Incremental computation is another way to improve the performance of a program by limiting the amount of work done upon a function call. Solutions based on this technique define how to calculate the result of a function not only from the newly supplied list of arguments, but also from the results returned from previous computations.
Depending on the approach taken, it may not be possible to avoid a complete recomputation of the output, but in many cases, the result of the previous computation may be reused to produce an updated output faster than a complete reexecution of the function would do. Thus, incremental computation is useful whenever a small input change leads to a relatively small change in the output.

Many solutions build upon this simple idea of reusing or adapting previous calculations. Demers et al. [14] in 1981 proposed the use of dependence graphs as a way to achieve incremental algorithms. Dependence graphs record the dependencies between computation data in a directed graph. The idea is to associate to each vertex \( v \) of the graph a value function that determines the value of \( v \) based on the values of the vertexes with incoming edges incident to \( v \). Thus, upon an input change, specialized change-propagation algorithms update the data that is affected by the change, thus producing the new return value.

This update is done on all the vertexes that belong to the transitive closure of vertexes that changed. In other words, if there is at least a path from a changed vertex to another vertex, the latter needs to be updated.

Dependence-graphs were originally static data structures that disallowed change-propagation algorithms to update the dependence structure of computation [14]. This limitation was later addressed by Acar et al. [6], who proposed dynamic dependence graphs (DDG). In their work, Acar et al. introduced the concept of modifiable reference to implement DDGs and to enable adaptivity in programs. A modifiable reference holds the value of an expression whose value may change when the input changes—that is, a modifiable reference records the dependence of one computation on the value of another.

To make one program adaptive, programmers store all data that can change inside modifiable references and modify the program to read and write to modifiable references explicitly. As a self-adjusting program executes, it automatically constructs the DDG, so, when the contents of a modifiable reference changes, it can update the computation by performing change propagation.

There is an interesting duality between DDGs and memoization: DDGs identify parts of a computation that are affected by a particular change and reevaluates them, whereas memoization identify parts of a computation that remain the same and reuses their results. Umut Acar [5], and then Umut Acar et al. [3, 2, 4] combined these two techniques to limit the amount of work that must be redone in pure functional environments.

Pugh and Teitelbaum [34] were the first to apply memoization to incremental computation. Their key observation is that, in recursive algorithms, to maximize the applicability of memoization it is fundamental that two similar problems are broken down in a way that they share common sub-problems, because, that way, solving one problem will involve parts previously solved for the other.

Incremental computation is a very promising technique that, thanks to recent work by Acar et al. [1], can be applied even to imperative contexts. Its major limitations relates to functions that are rarely called consecutively with similar input, which leads to the change propagation mechanism constantly updating the DDG. Additionally, it still lacks support for mainstream languages, the available implementations are only for SML and C, the available languages require programmer guidance to ensure efficient change propagation, and, except for some very preliminary results [20], it is still not known how self-adjusting computation can be applied in parallel programs.
3.2 Dynamic Programming Style

Dynamic programming is a tactic for solving complex decision problems that decomposes them into smaller problems. Introduced in 1957 [8] by Richard Bellman, just like memoization, dynamic programming improves functions’ execution time by preventing redundant computations from happening. But, unlike memoization that improves performance by remembering return values, the emphasis in dynamic algorithms goes to the way a function builds the solution, that is, solves the problem.

The key for better performance is on how a large and complex problem is decomposed into smaller, simpler, and overlapping subproblems. Dynamic algorithms formulate their solution to a problem recursively in a bottom-up fashion: Smaller subproblems are solved first and their results are then used to calculate the solution to more complex ones. Usually, all is done within a single function call.

There are many examples of dynamic algorithms in computer science, specially related to finding shortest paths between two nodes of a graph [15, 7], bioinformatics [32], or string matching [25].

To exemplify the differences between memoization and dynamic programming, I present in Listing 3.1 a possible dynamic solution to calculate the Fibonacci function.

```java
public int fib(int n) {
    int prevFib = 0;
    int currentFib = 1;
    if (n <= 1) return n;
    for (int i = n - 1; i > 0; --i) {
        int newFib = prevFib + currentFib;
        prevFib = currentFib;
        currentFib = newFib;
    }
    return currentFib;
}
```

Listing 3.1: Java implementation of the Fibonacci function using a bottom-up approach, as it is advocated by the dynamic programming style.

Comparing this version of the Fibonacci function with the one shown in Listing 2.1 that uses memoization, we can conclude that both methods are linear in time, because they calculate the value of the nth element in the Fibonacci sequence only once. The difference is that the memoized version requires $O(n)$ space to store the cache, whereas the dynamic solution needs constant space to produce the correct result.

Thus, performance-wise, dynamic programming style is a good alternative to memoization because it also limits the amount of redundant computations and it lacks the cache look-up, store, and retrieval overheads specific to memoization.

Despite the possibility of better performance, we can list three big disadvantages related to dynamic algorithms: (1) they are difficult to put together because programmers are trained to think in a top-down manner and dynamic solutions use a bottom-up approach, (2) we can use dynamic algorithms only in problems that can be broken down recursively, and (3) it is not trivial to automatically transform any given algorithm to use a dynamic programming style, whereas the same does not apply to memoization.
3.3 Automatic Memoization Tools

Even though programmers are allowed to explicitly augment functions with caches, because in pure functional programs there is a simple and clear pattern to memoize a function—it suffices to collect the list of arguments given as input to the function and use it as the cache-key—memoization can be accomplished with minimal effort from programmers through external transparent mechanisms. These automatic memoization tools represent a very tangible way to reduce costs because programmers can concentrate in coding the solution, needing not to worry about implementing themselves this optimization strategy.

In general, automatic memoization tools introduce new primitives in the programming language that change a function into a memoized one upon programmers request. How this transformation is accomplished ultimately depends on the mechanisms available on the target programming language. For example, a LISP implementation [18] may use a combination of macros and closures that make the intended transformation, whereas a C++ solution [28], because it lacks first-class functions, may need to use a pre-processor to transform a function into a memoized version of itself at source code level.

Adding to a natural ease of use, fast learning curve, modularity, and a reduction in the time spent on manually implementing memoization, automatic tools have the additional benefit of improving performance without introducing bugs, otherwise frequent in handmade memoization solutions.

The problem with current automatic memoization tools is that none deals with the unique characteristics of imperative stateful systems. As these automatic tools were built with pure functional contexts in mind, they cannot be used in methods that read values from shared state or that produce side-effects, without introducing semantic changes.

3.4 Memory Management

As already discussed, memoization is an optimization technique that trades execution time for memory consumption. So, to extract the best results from memoization it is crucial to control the amount of memory used by the system, because the moment primary memory is full, and the system halts to collect unreachable objects (garbage collection) or to transfer objects to secondary memory (swap), performance deteriorates to the point where all the benefits of memoization are lost. This problem is aggravated as the number of memoized methods grows because so does the number of memo caches. Therefore, memory management is a classic problem ever associated with memoization.

But since the proposal of memoization in 1968, memory accesses became faster and memory grew substantially in size. Further, by skipping the execution of a function we may be preventing the construction of large local data structures or even preventing object loads from databases. Thus, we could think that, when designing a cache, memory constraints are no longer important.

The reality is that as memory grew bigger so did software systems. This increase in size of software systems is enough to offset the gains obtained from larger memories and to justify the need for, at some point during system execution, actions that lower memory consumption levels.

Usually, memory management is done by controlling the information stored in cache—that is, by removing elements from the cache. The most efficient algorithm would be to discard always information
that will never be needed again or that will be relevant only in a distant future. Because, generally, this is not possible to know in advance, there are some approaches that try to offer more realistic solutions based on how frequent information is used or the last time information was useful.

The Least Frequently Used (LFU) policy discards first those entries that are used the least, by counting how often an item is needed, and Least Recently Used (LRU) policies discard the least recently used entries first, by keeping track of the last time that an item was used.

Even though these general purpose policies can be applied to control the amount of space occupied by a function cache, the way they choose which cache entry to purge does not take into consideration the unique characteristics that distinguish function caching from other forms of caching, such as buffering or memory pages.

To better understand these differentiating characteristics, consider once again the memoized version of the Fibonacci function, implemented in Listing 2.2 on page 9, and a memo cache with both the values for \( \text{fib}(4) \) and \( \text{fib}(5) \). In this scenario, a call to \( \text{fib}(5) \) results in a cache hit. Because of this top-level hit, \( \text{fib}(4) \) is not called and its respective cache entry is not used. Therefore, the probability of using the cache entry for \( \text{fib}(4) \) is lower if the entry of \( \text{fib}(5) \) is still cached.

Similarly, if calling \( \text{fib}(5) \) results in a cache miss, the time spent calculating the 5th element of the Fibonacci sequence depends on whether the values of \( \text{fib}(4) \) and \( \text{fib}(3) \) are cached.

Hence, a function cache differentiates itself from other caching solutions because the frequency of use of an entry in the cache varies based on what else is cached, and the cost to recalculate an entry depends on the complexity of the function itself and on what else is cached.

William Pugh proposed an improved replacement strategy that accounts for those differentiating properties [33]. His practical algorithm classifies entries according to their use frequency and recomputation time—that is, the amount of time that it is expected to take to recompute a particular entry if that entry is removed from the cache.

The purge algorithm, when it must free space from the cache, randomly selects an entry using a probability distribution that is highly skewed towards replacing a lower ranked entry, but can also evict a highly ranked one. This way, if by mistake an entry was highly rated it does not stay in the cache forever.

### 3.5 Precise Dependencies

The intent of memoization is to prevent functions that will execute redundant computations from running. This goal is accomplished using a cache and doing a cache search upon each call to a memoized function. Because the obtained speedup is directly proportional to the number of cache hits, false negatives when searching the cache will lessen the benefits yielded by memoization.

A false negative happens every time, that due to a too strict match between the supplied arguments and those stored in the cache, a recomputation is not prevented. Heydon et al. [23, 24] observed, in the context of a software configuration management system for building large-scale software similar to Make, that false hits happen because the dependency that exists between the result of a function and its respective arguments is too limiting, as if cache entries are constructed in an overly restrictive way, limiting their usability.
```c
int f1(int x, int y, int z) {
    if (x > 0) return y;
    else return z;
}
```

**Listing 3.2:** Simple function. The value returned by the function depends on the value of x and y, if x is positive, or on the value of x and z, if x is not positive.

To illustrate, consider the function in Listing 3.2 and two consecutive calls:

1. `f1(1, 2, 3);`
2. `f1(1, 2, 4);`

As discussed, each cache entry, for a particular function, maps a variable number of arguments to the respective output they produce. In this case a naive approach will combine the three arguments of function `f1` to construct the cache key. A careful observation can conclude that, after the first invocation to `f1`, the second invocation will result in a cache miss, but there should be a hit on the cache entry created by the first call, as the value of `z` is not relevant to the outcome of the function.

This behavior can be accomplished only by noticing that the result of a function depends on two aspects: those that influence the result at call-time and those that influence it dynamically at execution-time. These two aspects, respectively, will form the primary and secondary parts of the cache key. In the above example, the primary key is simply formed by the function body (so that if the function is modified there will be no hit in the cache with that function’s previous body), while the secondary key includes the dynamic dependencies on the arguments `x` and `y` (x=1, y=2). Employing this scheme, the second call will now produce a hit and the return of value 2, without executing `f1`.

The authors also highlight the fact that even with this scheme, the secondary key is still too strong, in the sense that the result only depends on `x` being positive, not necessarily equal to 1. Thus, a smarter caching policy could use predicates on the values of secondary keys instead of concrete values to improve performance. Unfortunately no further explanation is given on how to automatically extract these predicates or how to represent them.

### 3.6 Identifying Side-Effects

We have seen that in imperative programs methods often produce side-effects. But it is not less true that there is nothing in the imperative paradigm that prohibit the construction of functional programs, if that is the intent of the programmer. Thus, there is also nothing that prohibit imperative programs to have some methods that are side-effect-free. The problem is how to find this subset of referentially transparent methods.

Section 3.6.1 starts by describing the most common and easiest solution to the problem of memoizing methods that produce side-effects, which depends heavily on programmers’ knowledge of the problem and their ability to recognize state-changing methods.

Because programmers cannot easily infer which methods are side-effect-free, the memoization tools presented in Section 3.6.2 and Section 3.6.3 offer a mechanism to identify which methods are safe to
memoize, guaranteeing that none of the problems listed in Section 2.3.2 regarding side-effects happen. This identification can be done resorting to one of two approaches, each with its share of advantages and drawbacks: static or dynamic analysis.

3.6.1 Programmer-based Solutions

The simplest solution, and the one followed by all automatic memoization tools referred in Section 3.3, is to transfer the responsibility of identifying the subset of pure methods to the programmer, whereas the memoization tool does the transformation blindly according to what is specified.

These tools disclaim that programmers are fully aware of when they can use automatic memoization tools as well as what are the pitfalls of memoizing non-pure methods. Thus, the problem is solved because there is no problem to solve: If a programmer memoizes a non-pure method, the behavioral change between the original and the memoized version of the method is wanted; after all, the programmer is aware of this variation.

Because the only guarantee given by this solution is that memoization works as expected for pure methods, this approach is error-prone and, I argue, one of the reasons why memoization is seldom used in object-oriented programming. And worst, if programmers assume the risk, memoize part of the system and the risk does not pay off, then the system must be debugged and the programmer has no clue where to start looking for the problem because none of the tools mentioned before help on this task.

Another solution proposed by Jack Mostow and Donald Cohen [31] is in tune with the one described before, because it does not prevent non-pure methods from being memoized, but is a little more relevant due to the feedback it gives to programmers and users.

It is true that memoization can change system behavior, but whereas other solutions take this change as something to avoid, Mostow and Cohen advocate that this changed semantic can be acceptable, unimportant, or beneficial, depending only on the intent of programmers or users who interact with the system. This may happen in the following cases:

- **Elimination of output:** Memoizing a method that logs to a file, for example, results in a message being printed only once, rather than several times.
- **Elimination of input:** A memoized method that receives user input will cache such information and save the burden of re-entering it again and again.
- **Elimination of break:** Memoizing a method that generates an error from which the user can recover, by caching the result saves the task of fixing it the next time.

As this intent cannot be inferred by code analysis only, in the article mentioned above, the memoization system makes use of Interlisp’s [37] Masterscope to identify all the potential side-effects of executing a method, leaving to the user, by providing an interactive user interface, the task of deciding what to do next.

With such feature, programmers may memoize a program, run it to extract which methods produce side-effects when executed, and then remove memoization from such problematic methods before sending the system to production. The problem here is that if the system is large enough it is difficult to make
a test-suite that exercises all the possible call paths and it may happen that some non-referentially transparent methods remain memoized.

3.6.2 Static Purity Analysis

Static analysis, such like the one proposed by Franke et al. [16], work at source code level. This particular solution transforms a Perl program into a semantically richer textual representation that is easier to analyse, and was designed to facilitate the identification of all methods that are safe to memoize from the system. In general, all static source code solutions follow a similar approach.

These solutions traverse the source code and, for each method \( m \), they build a list of all other methods callable from \( m \). If any of the callable methods produces side-effects, depends on the external state of the program, or was previously marked as not memoizable, \( m \) is also marked as unsafe. In the end, the system outputs all the memoizable and non-memoizable methods, accompanied with a list of variables that are the reason why the method is not safe to memoize. Such description can then serve as input to a memoization tool, which will memoize all the safe-to-memoize methods, or to inform the programmer, leaving to him the decision of what to do next.

Static solutions have the advantage of enforcing correct system semantics while introducing no runtime overheads, which, for an optimization technique such as memoization, is essential to obtain the best speedup possible. The biggest problem with static solutions is that they are conservative: a method is classified as not cacheable when there is, at least, one path of execution where it may cause side-effects. So, a method that rarely causes side-effects, will be classified as not cacheable even if, when executed, the path which leads to side-effects is not taken.

One way to improve the applicability of memoization is to redefine which assignment operations are considered harmful and which are not. If we can allow writes to some extent, then less methods will be deemed as not memoizable.

That was the idea of Rountev in his work [35]: a method can execute write-operations as long as the observable state before and after its execution is exactly the same, that is, the changes are not observable outside of the method that produces them.

Both works highlight the difficulty that arises when there are calls to external methods, for which no implementation is available. The conservative approach is to mark all those methods as not memoizable, with the respective penalty of classifying all the methods that use them, also as not memoizable. Franke et al., in the previously cited work, proposed the introduction of keywords that help the classification algorithm with information about the nature of such methods.

3.6.3 Dynamic Purity Analysis

Theoretically, runtime dynamic solutions may broaden the applicability of memoization because they do not classify methods, they classify executions. Given that runtime approaches have access to the instructions that are being executed in the system, they can know if any state-changing operation is executed. If so, and for that particular set of arguments, the method is not-pure, otherwise it is.

As mentioned, the biggest advantage of these approaches is the possibility of memoizing more methods,
with the drawback of introducing runtime overheads that reduce the performance due to the necessity of monitoring instructions.

One particular relevant example of dynamic purity analysis is the work by Xu et al. [38], not only because it tries to apply memoization to an object-oriented environment (Java) but essentially because it introduces several definitions of purity that help broaden the applicability of memoization, much like the previously cited work of Rountev.

Given that Java programs are compiled to Java bytecode—instructions that the Java virtual machine executes—the runtime analysis is done by defining which of these instructions are potentially dangerous and, if any is executed, the method is classified according to their newly introduced purity categorization:

1. **Strongly Pure**: if the method behaves like a pure function.
2. **Moderately Pure**: if the method is strongly pure and creates/alters objects that do not escape the method execution.
3. **Weakly Pure**: if the method is moderately pure and reads information from the heap that is accessible from the supplied arguments.
4. **Once-Impure**: if the method is an impure method that behaves like a weakly pure one after the first invocation.

In their article, Xu et al. state that even once-impure methods are good candidates for memoization because they are executed at least once before being memoized, therefore, they will not be deemed as impure just because mandatory class loading and initialization, during the first invocation, causes writes to memory. Thus, they broadened the application of memoization to once-impure methods.

Because their dynamic analysis tool is implemented in the SableVM\(^1\) Java bytecode interpreter, programmers are limited to use this particular JVM to run their programs.

Further, they state that, due to a limitation in their memoization tool, only a fraction of the memoizable methods in tested applications can be safely memoized. Additionally, their implementation of memoization and dynamic purity analysis is still very inefficient. On all tests conducted, the memoized version performed worse than the unmemoized one, taking in average twice as long to complete. Finally, as we will see in the next section, their memoization tool does not detect or account for all types of shared state reads.

### 3.7 Shared State Reads

As we saw in Section 2.3.1, if a method’s result depends on values retrieved from shared state, it is not possible to apply memoization by resorting to the functional approach of using the supplied arguments as the cache-key. Because fully handling shared state dependencies represents an additional difficulty in the memoization process, if we could identify methods that access shared locations, then we could reuse previous work on memoizing functional programs, given that we prohibit memoization on the identified methods that read from shared state.

\(^1\)A Java Virtual Machine implementation.
This was the idea of Franke et al. [16] in their previously cited work (Section 3.6.2). The identification is done at source code level by collecting the name associated with local variables, in a per-method basis. With this information, it is possible to assess the source (local or external) of all read operations done inside a method. If it is an access to a shared variable then the method is marked as non-memoizable. Once again, the biggest problem with this solution relates to the drawbacks associated with how static approaches classify methods, which can render memoization useless in imperative contexts.

The work by Xu et al. cited before tries to go one step further. By “flattening” reference arguments—recursively gathering object type and primitive field values for all reachable types—and incorporating such information in the cache-key, the authors allow heap reads as long as the retrieved values are reachable from the supplied arguments. This attempt to extend memoization to object-oriented programs does not entirely solve the problem because it does not account for reads of static fields or class fields that influence the outcome of the method, and were not received as input. For example, the scenario described in Section 2.3.1 and implemented in Listing 2.3 would still fail.

To the best of my knowledge there are no previous works on a memoization system that automatically identifies the relevant state for a method’s result and incorporates such information in the cache-key. Thus, current automatic memoization systems still are unsuited for imperative object-oriented programs.

3.8 Summary

This chapter gives several examples of current alternatives, techniques, and approaches that may be used to improve the performance of a program or to deal with the difficulty of applying memoization in the presence of methods that read from or write to shared state.

The chapter begins by discussing some alternative techniques to memoization that programmers can use to improve the performance of a software system. It presents a very promising technique—incremental computation—that, after capturing the runtime behavior and data dependencies of the computations performed by a function, adapts future outputs of the function, finding the parts of the computation that changed and recomputing only those parts. Although promising, it is a work that still needs further investigation. Its current formulation does not deal with concurrent systems and still lacks a mainstream language implementation.

Then, this chapter introduces and presents the benefits of using dynamic programming algorithms that, like memoization, avoid redundant computations but lack the extra memory needed to store the function cache and the overheads of executing cache operations. The problem with dynamic algorithms is that, unlike memoization, they are difficult to build and cannot be automatized.

After that, I discuss current memoization tools that are capable of automatically memoizing functions, upon programmers request. These tools have the benefit of being transparent to programmers, not introducing bugs common in handmade solutions, but are limited to functional environments.

Related to caches, this chapter continues by discussing the problems of managing the amount of memory used by a function cache and why a correctly constructed cache-key is important to guarantee more cache hits.

Finally, this chapter concludes by presenting current techniques to detect methods that access shared state.
First, it discusses static approaches that, through source code analysis, classify methods as functionally pure or impure. This pre-run approach has the clear advantage of not introducing runtime overheads, at the cost of classifying a method as impure, and thus not memoizable, if there is the possibility that it produces side-effects. To overcome this drawback, I then introduced dynamic purity approaches that classify executions, and not methods, as producing side-effects, but introduce runtime overheads.

Second, this chapter ends concluding that there are no current robust and complete solutions to the problem of capturing shared state reads. It presents an approach to detect reads from shared state that can be used to prohibit memoization in methods that execute them, and a solution that incorporates, to some extent, shared state values in memo caches.
Chapter 4

Solution: The ATOM System

In this chapter I present my solution to the problem of memoizing methods that read from or write to shared state. I propose an STM-based approach to capture shared state reads, to detect executions with side-effects, and to incorporate side-effects in the memoization process. I describe the ATOM system, a Java implementation of my memoization proposal that, I believe, is the first automatic memoization system adequate to the unique characteristics of imperative object-oriented programs.

Given that my solution relies on previous work on software transactional memory, I begin this chapter by introducing the concept of STM in Section 4.1, centering the discussion on a particular STM that uses versioned boxes to keep multiple versions of each transactional location, and that will give support to my memoization implementation.

In Section 4.2, I present my proposed solution to automatically build the relevant state for the output of a method, detect side-effect-free executions of methods, and incorporate in the memo cache information about side-effects, so that they can be reapplied in the future upon a cache hit.

I continue by describing the ATOM system in Section 4.3. I start with a brief discussion about the rationale behind my implementation. Next, I present the ATOM API, how the memo cache works, how the memo cache limits memory consumption, and discuss a memoization advisory system, which is fully integrated within the ATOM, that helps programmers choose the best methods to memoize. I end this chapter with a discussion of why memoization can be particularly useful in systems that already use an STM, in Section 4.5.

4.1 Software Transactional Memory

So far I presented solely how we can use software solutions to improve the performance of a system, having yet to discuss the merits of hardware approaches. Given that performance is a runtime characteristic of the system, one simple way to boost the performance of a system is to run it on a better hardware—that is, if a system needs a performance boost one easy way to achieve it is to upgrade the running environment with a faster processor.

The problem is that, as processors became faster throughout the years, physical constraints, such as power consumption and the consequent high level heat generation, started to limit CPU frequency
scalability to the point where the hardware industry decided to lean towards a multi-core processor solution. This decision defined not only a new generation of computers, but essentially laid down the rules for the development of efficient software, bringing concurrent programming into the mainstream and highlighting the necessity for adequate parallel programming abstractions that help programmers build systems that take full advantage of the underlying hardware platform.

In this section I introduce the concept of software transactional memory, which brought the expressiveness of transactions to mainstream programming, leaving behind the cumbersome work of explicit lock-based constructions. Section 4.1.1 does a quick overview of the evolution and the reasons behind the work on software transactional memory, whereas Section 4.1.2 centers the discussion on a particular STM implementation.

### 4.1.1 Introduction to Software Transactional Memory

 Originally proposed by Herlihy and Moss [22] in 1993, the work on Transactional Memory (TM) started as an hardware architecture (HTM) based on cache-coherence protocols that intended to make lock-free synchronization as efficient as conventional techniques based on mutual exclusion, but without the pitfalls associated with lock-based solutions such as priority inversion, convoying, and deadlocks.

 The idea later matured in the work by Shavit and Touitou [36], who proposed a software transactional memory (STM) implementation to tackle the absence of hardware architectures capable to offer the necessary support to implement HTMs, and later in the proposal by Herlihy et al. [21] that was the first to allow transactions and transactional objects to be created dynamically, thus being well suited to the implementation of dynamic-sized data structures such as trees and lists.

 The novelty about this approach is the introduction of the notion of atomic actions, or transactions, into the programming model. Hence, instead of restricting access to data through locks, with an STM programmers specify which operations must execute atomically, leaving to the STM the responsibility of providing the intended semantics, while maintaining as much parallelism and concurrency as possible.

 From the perspective of an STM, operations executed within a transaction do not have a special meaning associated with them: They are just a series of reads from and writes to shared memory locations. STMs intercept these accesses to shared memory so that they may detect when two concurrent transactions are interfering with one another, in which case the STM stalls, aborts, or restarts at least one of the transactions.

 There are numerous STM implementations, varying considerably in how they ensure the atomicity of operations. For this work I chose to use the Java Versioned STM (JVSTM) [11, 10], so I concentrate the discussion on this particular STM implementation.

### 4.1.2 The Java Versioned STM

 The JVSTM is a multi-versioned object-oriented STM implemented as a pure Java library\(^1\) that was conceived with read-dominated, domain-intensive applications in mind. Unlike previous STMs, the JVSTM uses versioned boxes (VBox) to keep multiple versions of each transactional location.

\(^1\)The source code of the JVSTM is available at \url{http://web.ist.utl.pt/~joao.cachopo/jvstm/}
The versioned box is the central element of the JVSTM and a correct behavior by the transactional system depends on versioned boxes (instances of the class VBox) being used to encapsulate mutable values. Each VBox instance holds a tagged sequence of values that represents the history of that versioned box. Each element of this history is implemented by the VBoxBody class, corresponds to a successful assignment made to the box, and it is tagged with the number of the transaction that made the change. This tag number is the value’s version.

For instance, consider an account’s balance box, used to hold the history of an account’s balance. This box was created in transaction number 1 with the initial balance of 500, and then changed in transaction number 42 due to a withdraw of 200 monetary units. In Figure 4.1 we can see a graphical representation of the state of the box that holds the balance after the operations described before.

In the JVSTM, as in most nonblocking STMs, changes made during a transaction are made permanent only at the transaction’s end (commit time) and only when the transactional system can assure that there are no consistency violations. So, the commit of a transaction is responsible for detecting conflicts and can yield one of two possible results:

- success—all of the values written by the transaction were applied to the shared system state.
- fail—none of the values written were applied and the transaction should be restarted.

To detect conflicts among concurrent transactions, the JVSTM logs into a per-transaction read-set which boxes, and respective versions, were read inside that transaction. Likewise, it logs into a write-set which boxes were written and with what value. The read-set and the write-set will assume great relevance later in this dissertation because they are the key elements that will allow us to extend the applicability of memoization.

Transactions can be decomposed into subtransactions, which, in turn, can be decomposed into more subtransactions, forming an arbitrarily deep hierarchy of nested transactions. A nested transaction is created every time a new transaction is started in the context of a surrounding transaction and, in the specific case of the JVSTM, a transaction does not execute while its only child does (linear nesting [30]).

To make methods transactional, the JVSTM provides a method annotation—the Atomic annotation—that drives a post-processing tool that wraps methods’ bodies inside a transaction.

### 4.2 Extending the Applicability of Memoization

To extend the applicability of memoization to object-oriented programs, as we saw in Section 2.3, we need to address two main problems. First, we need to be able to capture all the relevant values for a method’s
result, which typically include not only the values received as arguments, but also values belonging to the program’s shared state. Second, we need to identify which methods have side-effects, so that we may choose either to not memoize them or to collect sufficient information to correctly replicate their behavior. I shall show in this section how I intend to use STMs to achieve both goals.

4.2.1 Finding the Relevant State

To solve the problem of constructing the relevant state for a method’s result, I propose to use the support already provided by an STM. If a memoized method executes inside an STM transaction, then all of the memory read operations made within that operation will be registered in the transaction’s read-set. Thus, at the end of the transaction we will know which values were read to compute the method’s result, thereby capturing all of the relevant state for the computation of this particular result.

This approach has a second advantage. If we recall what was discussed in Section 2.4, to correctly handle relevant states, besides registering which external values were read, we must be able to store in the cache information regarding the locations from where those values were retrieved from.

Once again the underlying transactional system offers the solution for this problem. Because versioned boxes reify the concept of mutable locations, we can simply store in the cache all the instances of versioned boxes belonging to the read-set, knowing that a particular versioned box instance uniquely identifies a shared location and allows the memoization system to query its current content.

Thus, by storing the read-set in the cache, the next time the memoized method is called we can check if each versioned box that belongs to the stored read-set preserves the same value as when the method originally executed. If so, there is a cache hit and the method’s execution is skipped. Otherwise, we must reexecute the method.

Now, after presenting the key idea of my solution to automatically capture and validate relevant states, it is time to return to the example of the banking application introduced in Section 2.2 to discuss briefly on what extent this proposal solves the problems identified. In Listing 4.1, I show a modified version of the Client class of the bank example.

The major difference between this new version of the Client class and the original one shown in Listing 2.3 is that the new Client class is transactional: I replaced the Client’s accounts field by an instance of the transactional versioned set VSet, and encapsulated the Client’s methods inside a transaction, which is accomplished using the Atomic method annotation, as shown. The transformation is not complete however if we do not transform the class Account to make it transactional, as well.

With these modifications in place, let’s assume also a simple scenario for the banking application in which the bank management application is running at a small banking agency that has a single client. This client goes to his local banking agency and decides to open two distinct accounts with initial balance of 400 and 600, respectively. Figure 4.2 depicts the state of the system after those account creation operations.

A few days later this client goes to an ATM machine and asks for his total balance. Because the method getTotalBalance, is now an atomic operation, at transaction’s end the resulting read-set can be seen in Figure 4.3. If the method getTotalBalance is memoized, the read-set will be stored in the memo cache.
class Client {
    VSet<Account> accounts;

    (...) 

    @Atomic
    long getTotalBalance() {
        long total = 0;
        for (Account acc : accounts) {
            total += acc.getBalance();
        }
        return total;
    }
}

Listing 4.1: JVSTM-based implementation of the Client class from the banking domain example. The boldfaced lines represent new code that was not present in the original implementation of the Client class (Listing 2.3).

Figure 4.2: UML object diagram showing the state of the banking system after a client opens two distinct accounts with initial balance of 400 and 600, respectively. Because it is the first transaction executed in the system, all versioned boxes are tagged with the number 1.

Figure 4.3: Read-set of the method getTotalBalance if executed with the system state depicted in Figure 4.2. The read-set, represented in the bottom of the figure, has three entries, one for each versioned box read by the method. The number enclosed in the smallest rectangle of each entry of the read-set is the version of the box read.

If the client decides then to open a third account, now with the initial balance of 500, this state changing operation will not only instantiate a new Account instance, but more importantly add a new value to the history of the versioned box accounts as shown in Figure 4.4.

35
Figure 4.4: State of the banking system after the bank’s only client opens a third account with initial balance of 500. This is the second transaction executed by the system, thus the newly created Account’s balance box is tagged with the version number 2. The versioned set VS1 is now at version 2 because a new value was added to the accounts set of the Client class.

Thus, when the client asks again for his total balance, the memoization system will validate the stored read-set. The validation process will find out that there is no hit in the cache because the versioned box VS1 changed. The method is reexecuted and the new read-set (Figure 4.5) is then added to the memo cache.

Figure 4.5: Read set of the method getTotalBalance if executed with the system state depicted in Figure 4.4.

4.2.2 Identifying Side-Effects

If we are trying to build an automatic memoization tool that is appropriate for imperative contexts, we must be aware that dealing with methods that are not referentially transparent is crucial not only for a successful memoization process, but more importantly, for determining the applicability of this optimization technique. Thus, we must adopt one of the two following approaches regarding side-effects:

- Offer the conventional semantics associated with memoization, not memoizing methods that produce side-effects.
- Extend the concept of memoization to imperative methods, so that the execution of a memoized method reproduces the side-effects that it would produce if it was not memoized.

Obviously, the first approach is simpler to implement because we need only to identify methods with side-effects and prohibit memoization in such cases. Therefore, leveraging once again on the support given by an STM, I propose a novel approach to identify whether a method produces side-effects or not.
My proposal is to execute a method inside an STM transaction. Thus, once it finishes, we may look at the transaction’s write-set to see whether the method wrote to any shared state; if it did, then we do not memoize this call; otherwise, we may memoize it as before.\footnote{This solution to the problem of detecting state changing operations is similar to the dynamic solutions described in Section 3.6.3, yet using a different approach.}

My STM-based solution does not classify methods as a whole as producing side-effects or not. Rather, it identifies whether a particular call produces side-effects or not, maximizing the possibility of using traditional memoization.

We turn our attention now to the second approach identified above, of being able to memoize method calls with side-effects. Going back to the example shown in Listing 2.5 on page 15, what does it mean to reproduce the behavior of \texttt{payInterest}, which has side-effects?

The externally observable behavior of \texttt{payInterest} is confined to the changes made to the accounts’ \texttt{balance}, because it has no return value. So, in this particular case, to reproduce the behavior of \texttt{payInterest} it is necessary to correctly change the value of \texttt{balance}.

The answer comes again from the STM. Looking at the write-set, we may see which boxes were written and with what value. So, it is possible to memoize any method that produces side-effects if we store the write-set in the cache and in the future, after a cache hit, we iterate over the associated write-set and reapply all the changes as an additional step of the memoization process.

### 4.2.3 Proposed Solution

To apply memoization in imperative object-oriented programs in a safe and robust way, I propose an STM-based solution that encapsulates memoized methods’ computations inside transactions to capture all the relevant information about a method’s behavior.

In my proposed extension to memoization, both the information captured by the transaction’s read-set, as well as the list of arguments, is used to classify computations as redundant or not, as opposed to only using the list of arguments in traditional memoization.

To deal with side-effects, I argue that programmers should be able to decide whether state changing operations should be elided or included in the memoization process. Therefore, I propose a run-time solution based on a transaction’s write-set to either accomplish dynamic purity analysis—a method’s execution is functionally pure if at the method’s end the associated transaction’s write-set is empty—or incorporate side-effects in memoization—the memoized version of a method with side-effects first applies the state-changing operations captured in the original run of the method and only then returns the saved result. In either case, the memoized method preserves the original program semantics.

Figure 4.6 shows the normal execution of a memoized method using my proposed extension to memoization. The novelty of my proposal is that as soon as a method starts, a transaction is created. Then, as usual, the system tests if it can skip the current execution of the method. If so, it must test whether the cached entry is read-only or not. If it is not, then it must apply the stored write-set. Despite the path taken, the memoization system proceeds by retrieving the cached result and committing the transaction. Otherwise, if there is no cache hit, the memoized method executes inside a transaction that registers all the relevant information, tests if it can memoize the current method execution, and only then it commits the transaction.
Figure 4.6: Execution flow of a memoized method using my memoization proposal. The novelty about my solution is to encapsulate a method’s body inside an STM transaction, to register shared state accesses, and then use that information to build cache entries. These entries are stored in the cache only if the method’s execution respects the semantics defined by the programmer. Because my memoization solution allows methods with side-effects to be memoized, the cache search algorithm may need to apply cached changes.

Because in multi-threaded systems transactions may abort, due to conflicts with other concurrent transactions, we must decide whether to add, or not, cache entries that represent executions that are aborted at commit time. Both alternatives are viable and preserve the semantics of the system, but I opted for the former because even a transaction that aborts due to a conflict can add valuable information to the memoization system.
A JVSTM transaction that does not survive the commit step and must be restarted, has seen a consistent state of the system. It happen that the state has changed and therefore the transaction cannot commit. But still, allowed the memoization system to collect information regarding a possible execution which could not be applied with the present state, but might be in the future. So, from transaction start and up until the commit phase, all the information collected constitutes valuable input to the memoization system because it represents the behavior of the system under a particular observable state that, if ever observed again, can be reused.

Further, if to abort a transaction, the transactional system simply checks if any of the locations read by the committing transaction were concurrently modified by an already committed transaction, there may be a cache hit on the newly added cache entry, that represents the aborted execution. This may happen because, even though the transactional location that triggered the abort changed, its current value may not.

As a matter of fact, if multiple threads share the same memo cache, it may even help concurrent transactions to conflict faster. Skipping the execution of a memoized method, that already originated the abort of a concurrent transaction and was cached, brings a read-write transaction faster to the commit step, where it is known that it will abort and restart, thus saving the time of executing work known to read boxes that were concurrently modified. Therefore, I chose to cache information about method’s execution, regardless of whether the surrounding transaction aborts or not.

4.3 The ATOM System

In this section I introduce the Automatic Transaction-Oriented Memoization (ATOM) system. To memoize correctly object-oriented imperative programs, the ATOM system implements the STM-based approach, described in Section 4.2, of using a transaction’s read-set to capture the relevant state for a particular result and using a transaction’s write-set whether to identify side-effect-free executions or to register possible write operations, so that it may apply them in future reexecutions of the same memoized method.

To better understand the choices made during the conception and implementation of the ATOM system, which is presented in the remainder of this chapter, it is important to know what are the architectural drivers of my solution. Thus, I discuss next the rationale for the ATOM system.

4.3.1 The Rationale for the Memoization System

An automatic memoization tool is perceived as relevant for developers only if the performance upgrade that programmers get after using memoization in their system is not marginal. In other words, programmers’ level of satisfaction is greater the higher the final speedup is. We know that the time spent on cache accesses, may those be queries, retrievals, or additions, directly influence the magnitude of the resulting global performance boost. Therefore, the ATOM system must thrive towards a solution that minimizes the time spent on cache operations. Specially, it must favor query operations because this type of operations is expected to be the most common one.

3As with the STM implementation used to implement this memoization proposal.
Another factor that greatly influences the acceptability of external tools is how easy it is to use them in a system. It does not matter whether the memoized version of the system is two times faster or can process more one thousand operations per second than the original unmemoized one, if achieving such improvements is an arduous, difficult, and complex task. Thus, the ATOM system must be easy to use, yet flexible and customizable enough, to further increase the appealing of memoization and the acceptance by the developers' community.

As mentioned in Section 4.1, parallelism and concurrency is now a hot topic due to the advent of multi-core processors. Further, to solve the problems related to shared state reads and writes my approach is to use an STM, which is used in concurrent programming to facilitate the development of parallel systems. Thus, the memoization system must be thread-safe, continuing to behave correctly despite the number of threads concurrently retrieving or adding cache entries.

As a final note, the main goal of this work is the development of a memoization system that is capable of dealing with the unique characteristics of imperative object-oriented systems. Thus, I make no contributions regarding memory management or cache eviction policies.

### 4.3.2 The ATOM API

I start this detailed description of the ATOM system with an overview of the ATOM’s API, which is the visible interface of this automatic memoization system and constitutes all the information that programmers need to know to memoize their programs.

The ATOM is implemented as a pure-Java library and, given that I tried to keep its interface as small and simple to use as possible, provides only a single interface for programmers—the Memo annotation.

Classes with methods that use this annotation are post-processed and rewritten, simplifying the task of memoizing methods: Programmers just need to express their intention, as shown in Listing 4.2.4

```
@Memo
long getTotalBalance() {
    /* getTotalBalance's Body */
}
```

**Listing 4.2:** Use of the annotation Memo to memoize the method getTotalBalance. The necessary code for memoization is introduced by post-processing the bytecode.

The automatic transformation is done as a step of the compilation phase and uses the ASM [9] library for bytecode manipulation. The rationale behind the decision to make all transformations at class-file level is essentially due to the fact that by making changes statically, and prior to system execution, the transformation process introduces no overheads in the system at run-time, just like a manual memoization solution.

Without going into much detail, the transformation class (ProcessMemoAnnotations) receives a list of Java class files that it must scan for Memo annotations. For each annotated method, the transformation process inserts in the class a new slot, named after the method to memoize, to hold an instance of the MemoCache. Because Java allows for method overload (several methods may have the same name as long

4The Memo annotation provides also the same semantics of the Atomic annotation, and, thus, replaces it.
as they differ from each other in terms of the type of the input arguments), to ensure the uniqueness of this name as well as an unambiguous relationship between the memoized method and its respective method cache, each inserted slot’s name is prefixed with “$cache,” and suffixed with the type of the arguments of the memoized method. If the annotated method is static the transformation process inserts a static slot to hold the memo cache.

The transformation class augments each memoized method with a memo cache because, most often, the receiver of the message (the instance on which the method is being invoked on) influences the outcome of the method. Thus, by spreading the cache by all of the objects of a class, rather than having a single cache for all of them, naturally partitions the cache and simplifies its maintenance, because when an object is garbage-collected, so is the portion of the cache that belongs to it.

Then, each memoized method’s body is changed. The transformation process adds a preamble that does a cache search complemented with the respective decision of whether to execute the method or not, whereas each return instruction is preceded by a call to the memo cache method to collect information regarding the execution. When this process ends, each new class definition is then written over the original class file.

As a complement to this process, the automatic memoization tool also adds, for each memoized method, a printing method that can be used by programmers, through reflection, to inspect the contents of the respective cache.

Figure 4.3 shows the body of a memoized method after being transformed by the ProcessMemoAnnotations class. As we can see, all the memoization behavior is implemented by the MemoCache class, which will be discussed in the next section.

4.3.3 The MemoCache Class

The MemoCache class, shown in skeletal form in Listing 4.4, implements the memoization strategy described in Section 4.2. Each instance of this class is a method cache that holds information regarding previous executions of a method.

Apart from the initialization method, which will be further discussed in Section 4.3.4 where I introduce the various memoization strategies and comparison policies, the methods search and collectInformation are the only visible interface of the MemoCache class.

The method search receives an array with all the arguments passed as input to the memoized method and is responsible for searching the cache for a hit. Failed searches are communicated through a statically constructed instance of class NotFound, whereas a cache hit returns the associated result and, if necessary, applies the stored write-set.

The method collectInformation is responsible for collecting all the relevant information about a method’s execution. Just like the search method, the method collectInformation receives an array with the arguments passed to the memoized method, receiving as well the result that the memoized method will return.
class Client {
    VSet<Account> accounts;
    MemoCache $cache_getTotalBalance;

    @Atomic
    long getTotalBalance() {
        Object[] argList = new Object[];
        Object res = $cache_getTotalBalance.search(argList);

        if (res != MemoCache.notFound) {
            return res;
        }

        long total = 0;
        for (Account acc : accounts) {
            total += acc.getBalance();
        }

        $cache_getTotalBalance.collectInformation(argList, total);

        return total;
    }
}

Listing 4.3: The ATOM’s version of the getTotalBalance method from the banking domain example, originally shown in Listing 2.3. This example highlights in boldface the changes done by the transformation class, responsible for automatically memoizing an annotated method.

public class MemoCache {
    public MemoCache(CachePolicyType type, MemoStrategy strategy) { ... }
    public Object search(Object[] args) { ... }
    public Object collectInformation(Object[] args, Object res) { ... }
}

Listing 4.4: Skeleton of the class MemoCache.

4.3.4 Implementation of the Memoization Cache

My implementation of the memo cache (Figure 4.7) is organized in two levels: The first level is composed of all the information available at call time—the arguments supplied to the memoized method—and maps to a second level which holds information observable only after the method executes—that is, the captured read-set, the returned result, and, possibly, a write-set.

The need for a two-level cache is not entirely justified by the source of information captured in each level being different: The way I conduct searches in each level is also distinct and represents the main reason for this multi-layered organization.

The first level of the memoization cache follows a similar approach to classic caching systems, adopting an map-based solution where the supplied arguments are used as the map-key, and maps to a first level entry. This first level entry contains a list of possible second level entries, which, in turn, hold the method’s result value, the relevant state that must still be valid, and possibly a write-set.
Figure 4.7: Organization of the method cache

Thus, I use a table lookup to search the first level of the MemoCache because the search method receives the list of arguments supplied to the memoized method, allowing the memoization system to uniquely identify a possible cached entry. With the relevant state the same does not apply: There is no way to know in advance which versioned boxes are relevant for a particular execution of the method and respective list of arguments. Hence, once I obtain a second level entry, the search algorithm must iterate over all its entries looking for a valid read-set. Intuitively, a read-set is valid if and only if all the boxes in it still hold the same value as when the read-set was stored in the cache. To see whether a box still has the same value as before, the ATOM system offers three caching policies: CachePolicyType.VERSION, CachePolicyType.VALUE, and CachePolicyType.IDENTITY.

The decision to offer three caching policies relates to offering programmers an opportunity to choose which equality policy adapts better to the situation, that is, which will translate to a better performance. The memoization system is robust enough to allow this choice to be made on a per-method basis, placing no restrictions on the caching policy used for a particular method.

**Version Cache Policy**

The first caching policy, implemented by the VersionRelevantState class, considers that a box still has the same value if and only if it is still in the same version (meaning that no write occurred to this box).

The VersionMemoCache is theoretically the fastest of the three caching policies because it just needs to do a simple integer equality test to assert if one entry of the relevant state is still valid, but can lead to less hits: from the moment one box changes value, any cache entry that depends on that particular box will forever be invalid.

It is also the one that consumes less memory, because it saves only the versioned boxes read by the memoized method, not caching the values that were read. This fact has an inconvenient repercussion in the validation step regarding values that were already written by any parent transaction, which will be discussed in Section 4.3.7.

The implementation of the method compareState, responsible for validating a relevant state, for the class VersionMemoCache is shown in Listing 4.5.

**Value Cache Policy**

The second policy type, implemented by the ValueRelevantState class, checks whether the current value of the box, regardless of its version, is equals to the value seen before.

The ValueMemoCache may lead to more hits than the VersionMemoCache because it allows a versioned box to change value, and still be valid, if this change has not altered the value of the box. It is also the
class VersionRelevantState implements RelevantState {
    protected final VBox[] _readSet;
    protected final int _version;

    @Override
    public boolean compareState() {
        for (int i = _readSet.length - 1; i >= 0; --i) {
            VBoxBody body = getLocalBody(_readSet[i]);
            if (body == null) return false;
            if (body.version > _version) return false;
        }
        return true;
    }
}

Listing 4.5: Implementation of the method compareState for the VersionRelevantState class. The slot _version holds the version of the transaction that instantiated the relevant state. The method getLocalBody returns the correct body of the versioned box passed as argument for the current transaction version, or null, if the versioned box has already been written by any parent transaction.

only cache policy that gives programmers the opportunity to control the validation process by defining their own custom implementation of the method equals, that is used to compare cached values.

The drawbacks of this cache policy relates to the fact that it assumes that the method equals is correctly defined for the values stored in the boxes and its time complexity depends on the complexity of such method.

Listing 4.6 shows the implementation of the method compareState for a value caching policy.

Identity Cache Policy

The third, and last, caching policy type, implemented by the IdentityRelevantState class, checks whether the cached entry and the respective versioned box reference the same object in memory, that is, if both objects are “==”-equal. If so, then the entry is still valid.

The IdentityRelevantState not only offers conventional semantics used in reference models to compare values for equality, but also constitute a very interesting compromise between the other two caching policies: a simple and fast comparison test. Furthermore, changing the value of a box does not invalidate forever cache entries that refer that box.

The implementation of the method compareState for the class ValueMemoCache is shown in Listing 4.7.

Memoization Strategy

The addition of new cache entries follows the semantics defined in Section 4.2—that is, programmers can opt either to memoize executions that produce side-effects, or not. Hence the MemoCache implements two memoization strategies: read-only and write-allowed.

The read-only memoization strategy follows the traditional memoization behavior of disallowing executions with side-effects to be memoized, whereas the write-allowed strategy caches any method execution,
class ValueRelevantState implements RelevantState {
    protected final VBox[] _readSet;
    protected final Object[] _values;

    @Override
    public boolean compareState() {
        for (int i = _readSet.length - 1; i >= 0; --i) {
            Object actual = getBoxValueDontRegister(_readSet[i]);
            Object stored = _values[i];

            if (actual == null) {
                if (stored != null) return false;
            } else {
                if (!actual.equals(stored)) return false;
            }

            return true;
        }
    }
}

Listing 4.6: Implementation of the method compareState for the ValueRelevantState class. The method getBoxValueDontRegister returns the value of the versioned box supplied as argument for the current transaction, without registering any information in the read-set.

saving the write-set and reapplying it if necessary after a cache hit.

Parametrizing Memo Annotations

Given that the ATOM system allows for three types of comparison policies and also two memoization strategies, the Memo annotation may be parametrized to inform the transformation process what is the combination of caching policy type and memoization strategy that the programmer intends to use in the annotated method. I finally present, in Listing 4.8, the detailed implementation of the Memo annotation.

The attribute type accepts three possible values—CachePolicyType.VERSION, CachePolicyType.VALUE, and CachePolicyType.IDENTITY—and defines how the cache will compare values for equality—using the version of the box, the value of the box, or the object referenced, respectively.

On the other hand, the attribute policy controls how side-effects are handled by the memoization system. If parametrized with the value MemoStrategy.READONLY, any method execution that writes to shared-state will not be memoized, whereas with the value MemoStrategy.WRITEALLOWED all calls to the memoized method will be cached, regardless of whether they produced side-effects or not, and the side-effects will be reapplied in future cache-hits, if necessary.

Both parameters are optional and, by default, the instrumentation tool inserts in Memo annotated methods a read-only cache that uses the version of the box to decide if the value that it holds is still valid.
class IdentityRelevantState implements RelevantState {
    protected final VBox[] _readSet;
    protected final Object[] _values;

    @Override
    public boolean compareState() {
        for (int i = _readSet.length - 1; i >= 0; --i) {
            if (getBoxValueDontRegister(_readSet[i]) != _values[i]) return false;
        }
        return true;
    }
}

Listing 4.7: Implementation of the method `compareState` for the `IdentityRelevantState` class. The method `getBoxValueDontRegister` returns the value of the versioned box supplied as argument for the current transaction, without registering any information in the read-set.

@Retention(RetentionPolicy.CLASS)
@Target(ElementType.METHOD)
public @interface Memo {
    CachePolicyType type() default CachePolicyType.VERSION;
    MemoStrategy strategy() default MemoStrategy.READONLY;
}

Listing 4.8: Implementation of the `Memo` annotation. The `Memo` annotation has two optional attributes that control the cache policy type (`type`) and the memoization strategy (`strategy`).

### 4.3.5 Granularity of the Read Set

The time that it takes to validate a cached relevant state depends not only on the caching policy used, but also on the amount of information that must be validated. The current implementation of the JVSTM, and the of ATOM system because it uses its transactional system, advises that each mutable class field should be encapsulated within a `VBox`.

This proliferation of versioned boxes in the memoized application has two direct consequences: First, the memoization system, to ensure that a cache entry is still valid, has to perform many comparison tests, and, therefore, its time complexity grows with the number of boxes to validate. The second consequence, on the other hand, is not as bad because getting slot-level information allows the memoization system to store precise dependencies. If we remember the discussion in Section 3.5, recording precise dependencies in the cache may improve the number of cache hits, because fine-grained dependencies are more easily met than coarse-grained ones.

So, if the result of a method `m` exclusively depends on the field `f1` of an instance of class `Cl`, only `f1` will be added to `m`'s read-set. Hence, if `m` is reexecuted with the same value of `f1`, even if all the other fields of the instance of `Cl` change, there will be a cache hit and `m` will not reexecute.

### 4.3.6 Memory Management in the MemoCache

To control memory consumption, the `MemoCache` adopts two distinct approaches, one for each level of the method cache.
The first level uses weak references to hold the values of the map—that is, second level entries are only weakly referenced. Using weak references helps controlling the amount of memory occupied by the method cache because, as specified in the Java memory model [27], weak references, unlike strong ones, do not prevent their referents from being made finalizable, finalized, and then reclaimed by the garbage collector (GC). Hence, if at a certain point in time the GC algorithm decides that it needs to free some memory, entries in the cache will be dropped, without affecting cache semantics.

On the other hand, the second level of the cache relies on a fixed-size number of entries, equal to the number of threads that may access the cache concurrently. This decision relates to the fact that by limiting the size of the cache we can control the amount of memory used by the memoized system. Further, by making the number of entries in each cache equal to the number of threads executing in the system we allow a thread to lag behind other concurrent threads, that are executing in a more recent state of the system and already changed caches belonging to memoized methods that the lagging transaction may access, and still find a cache hit. Finally, I decided to purge second level entries using a simple round-robin policy.

4.3.7 Extending JVSTM Transactions

My memoization implementation depends highly on the support offered by the transactional system. More specifically, on the ability to obtain from the JVSTM the read-set and the write-set, as well as the guarantee that all memory accesses are correctly registered in the respective set. But the JVSTM was not designed with memoization in mind. Thus, despite the large collection of classes implementing different transactions behavior, I decided to extend the already available transactions with a new type of transaction—the MemoTransaction.

Figure 4.8 shows the JVSTM and ATOM transaction hierarchy. The ATOM system implements two new transaction types, implemented in the MemoTopLevelTransaction class and in the MemoNestedTransaction class, which represent a top-level memoized transaction and a nested memoized transaction, respectively.

This decision to extend the transactions of the JVSTM with new memoization-specific transactions, is in part explained by the need to implement memoization specific semantics or optimization strategies that I will detail on the remainder of this section, but also due to a design choice to keep the implementation of the ATOM system separated from the JVSTM, meaning that I would not change the source code of the JVSTM just because, for example, I needed to access the read-set and write-set and both are private to the transaction. If this was the choice, it would result on an alternative version of the JVSTM that needed to be updated every time the original code of the JVSTM changes. This way, both projects can evolve separately, because they are independent, and the ATOM system may be seen as a plugable extension to the JVSTM.

Written Values Read Inside Nested Transactions

One of the first problems that memo transactions try to solve relates to how nested transactions’ read-set is populated. If we recall the discussion in Section 4.1.2, every time that a versioned box is read the transactional system adds a new mapping to the read-set of the running transaction. The only exception is when the box read has already been modified by the current transaction or any of its parents, in which case no new mapping is added.
Figure 4.8: The hierarchy of transactions of the JVSTM and the ATOM. The elements inside the dashed rectangle are specific to the ATOM system and complement the hierarchy of transactions implemented by the JVSTM. The class MemoTopLevelTransaction extends (solid arrow) the class TopLevelTransaction. Likewise, the class MemoNestedTransaction extends the class NestedTransaction. Both the MemoTopLevelTransaction class and the MemoNestedTransaction class implement (dashed arrow) the MemoTransaction interface.

Thus, consider a scenario where a memoized method m2, executing in the context of another memoized method m1, queries the transaction about the value of the versioned box vbox that is in the write-set of m1 with the new value of val1. The transaction will correctly return the value val1 but miss to add the mapping between vbox and val1 in m2’s read-set. Henceforth, any memo search made by m2, regardless of whether m2 is called at a top-level or in the context of another memoized method, will not validate correctly the cached read-set. Thus, the value of vbox may change and the cached entry still be considered valid by the memoization system because the memoization system failed to register the dependency on the value of vbox.

When a top-level transaction is queried about the current value of a versioned box that was previously written in the transaction, there are only two possible cases: (1) both the read-set and the write-set hold information regarding the versioned box, or (2) only the write-set has an entry for the versioned box.

In both cases, the search operation does not change or add new information to the read-set, and that is the correct semantics for transactions. If the read-set holds information about the queried versioned box, then the output of the method depends on the value registered in the read-set, and not on the one present in the write-set. If not, then the method does not depend at all on the value of that shared location, because if it did it would have read that location’s value before writing into it. Hence, this
problem occurs only with nested transactions.

The solution was to add a new set to the class `MemoNestedTransaction`—the `read-value-set`—and change the method that obtains the value of a versioned box. This new version, specific for memo nested transactions, guarantees that if the value of the versioned box is obtained from the write-set of a parent transaction, then a new mapping between the versioned box and this retrieved value is added to the read-value-set.

With this modification in place, when creating a new cache entry for a memoized method that executed inside a nested transaction, the relevant state portion of the cache-key is formed by the read-set and the read-value-set. For memoized methods that execute inside top-level transactions, no modification is necessary and only the read-set is used in the second level of the method cache.

Furthermore, in version memo caches, it is not possible to collect sufficient information to build a cache entry for executions of memoized methods that have a non-empty read-value-set. This fact happens as a result of the values referenced in the read-value-set not having a version number because they were not yet committed. Hence, during the collect information phase, the memo cache guarantees that if the caching policy is of the version type, new information will be added to the cache only if the read-value-set is empty.

**Populating the Read Set After a Cache Hit**

In concurrent environments, multiple threads may execute concurrently and STMs use the write-set as a buffer for the changes that a transaction wishes to do. These changes will be applied if the transactional system can guarantee that the committing transaction does not conflict with a previously committed transaction. In this validation process, the read-set is used to ensure that the committing transaction did not read a transactional location that other already committed transaction wrote.

Because the ATOM system relies on a transactional system, if we skip the execution of a read-only memoized method simply by returning its cached value upon a cache hit, there are lost side-effects that may change the semantics of the program: All transactional methods, even read-only ones, when executed write to the transaction’s read-set. Thus, to replicate the behavior of a memoized transactional method, besides replacing the method call with the cached result, and possibly applying the cached changes, the memoization system needs to populate the transaction’s read-set with the versioned boxes that would be read by the method if it executed normally.

One observation made during early tests with the ATOM system was that it spent too much time populating the read-set to ensure correct transactional semantics and that this action was negatively influencing the performance of the memoization system, especially for methods that read a large number of versioned boxes.

Given that the information registered in the read-set is necessary only if at commit time the transaction is read-write—the JVSTM’s validation task first looks at the transaction’s write-set and it uses the read-set only if the write-set is not empty—we could improve the performance of the memoization system by not populating the read-set when it is known to be not necessary. For example, we may skip the population step after a cache hit on a top-level, read-only, memoized method, because top-level read-only transactions never fail.

The problem is that, after a cache hit on a memoized method executing inside a nested transaction,
populating the read-set is unnecessary only if the nested transaction is read-only, its parents are readonly, and other nested transactions that may execute in the context of this transaction are also read-only. Thus, when a nested transaction commits it is not possible to know in advance if the read-set will be necessary in the future, or not.

Because the conservative approach of populating the read-set when a memoized nested transaction commits is very inefficient, even more inefficient in read-dominated contexts where the frequency of read-write transactions is low, the ATOM system enforces a speculative read-only policy on memoized methods.

Upon a cache hit, a promise is added to the running memo transaction regarding the values that would be added to the read-set. This promise is realized immediately if the transaction is already read-write. Otherwise, it is delayed until a future nested transaction tries to write to a box.

4.4 The Memoization Advisory System

As discussed in Section 2.1, memoization results in a performance boost only when it is faster to search the cache for a match than to reexecute the original method, and, of course, there is a cache hit. From this observation comes a crucial conclusion: If programmers memoize a method that never executes under the same system state or that computes the return value faster than the memoization system retrieves it from the method cache, memoization will deteriorate the performance of the system.

Therefore, the choice of which methods to memoize, although fundamental, is a difficult and arduous task due to the impossibility of, by code inspection alone or by reasoning about the intended system behavior, obtaining the list of methods that will render a better system performance after being memoized. This problem becomes even more complex in the case of the ATOM system because programmers need to worry not only about which methods are beneficial to memoize, but also which memoization strategy and comparison policy is best to use.

To simplify the whole selection process, I introduced the TestMemoCache class. This new type of method cache subclasses the original MemoCache and it is designed to collect per-memoized-method information.

Unlike the conventional MemoCache that enforces a single memoization strategy and uses a pre-defined equality policy, the TestMemoCache is in fact composed of six independent MemoCache instances, one for each possible memoization strategy/policy combination—WRITEALLOWED-VERSION, WRITEALLOWED-VALUE, WRITEALLOWED-IDENTITY, READONLY-VERSION, READONLY-VALUE, and READONLY-IDENTITY.

Thereby, upon a search request, the TestMemoCache queries each of its private MemoCache instances for a hit, collecting information about the time it took the cache to issue the response and if the obtained response was a cache hit or a cache miss. Independently of any combination of responses given by each of the four caches, the result of a search request made to the TestMemoCache always returns a miss, as a way to force the execution of the original unmemoized code portion of the memoized method. This way, the TestMemoCache can collect the time it would take the normal version of the code to execute and compare it with the various memoized solutions.

It is important to note that the time a search request takes to complete is equal to the time spent on the cache search, plus the time spent on executing the method’s body if the cache yielded a miss.
Each of the six searches is done inside a transaction that is aborted as soon as the TestMemoCache collects enough information about the request. This extra work is necessary to correctly recreate the execution scenario where only one cache is used. In particular it prevents side-effects done by WRITEALLOWED caches to influence the outcome of subsequent searches and computations.

With such behavior, for this selection process to produce relevant information it is important that programmers memoize all the methods of the system, or at least those that they sense will execute redundant work, and run some representative portion of the use cases of the application.

The runtime information collected by the TestMemoCache is then serialized to the file system and used as input to the MemoAdvisor class, responsible for calculating the total number of cache hits and cache misses, average execution time, and expected speedup over the unmemoized version, with each combination of cache strategy and comparison policy.

This overall information is presented to the programmer divided in two sections: methods where it is expected that memoization can accelerate their execution and those that do not. The advisory system complements such information with which combination of memoization strategy and caching policy type is best suited for each memoized method. It is then up to the programmer the task of choosing the final configuration of the system.

Listing 4.9 shows a possible output for the MemoAdvisor when three methods are memoized. Only expected average times are shown.

Once again, the automatic memoization tool can be instructed to automatically insert a TestMemoCache on annotated methods. This is accomplished by passing a simple flag (--test) along with the path to the folder on which the TestMemoCache can save the run-time information.

4.5 Memoizing Transactions

Programmers use STMs to build concurrent software systems more easily. But in terms of performance, and comparing with lock-based solutions, STMs in general, and the JVSTM in particular, introduce overheads in the form of:

- **Intercepted memory accesses**: Every access to shared memory is done through the respective versioned box. As if this intercepted access was not enough to introduce overheads, because each versioned box holds an history of values, the transactional system must spend additional time searching for the appropriate tagged value.

- **Extra Memory**: Each transaction stores a private read-set and write-set, and each shared location, because it uses versioned boxes, may hold more than one possible value. This need for auxiliary information increases the amount of memory that a transactified system uses.

- **Transaction Validation Step**: The that time it takes to execute a transactional method is the sum of the time it takes to execute its body plus the time it takes to create and validate the associated transaction. Whereas creating transactions is relatively simple and almost effortless, the validation and commit step may be rather expensive, especially in read-write transactions due to the validation of the read-set.
### Good

<table>
<thead>
<tr>
<th>Method</th>
<th>Memo Time</th>
<th>Normal Time</th>
<th># of Hits</th>
<th># of Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITEALLOWED_VERSION</td>
<td>561221</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>WRITEALLOWED_VALUE</td>
<td>558751</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>WRITEALLOWED_IDENTITY</td>
<td>558458</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>READONLY_VERSION</td>
<td>558149</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>READONLY_VALUE</td>
<td>558015</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
</tr>
<tr>
<td>READONLY_IDENTITY</td>
<td>557800</td>
<td>8827988</td>
<td>47</td>
<td>1</td>
</tr>
</tbody>
</table>

### Traversal1.performOperation()I

<table>
<thead>
<tr>
<th>Method</th>
<th>Memo Time</th>
<th>Normal Time</th>
<th># of Hits</th>
<th># of Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITEALLOWED_VERSION</td>
<td>51194045</td>
<td>641339196</td>
<td>114</td>
<td>25</td>
</tr>
<tr>
<td>WRITEALLOWED_VALUE</td>
<td>73801930</td>
<td>641339196</td>
<td>114</td>
<td>25</td>
</tr>
<tr>
<td>WRITEALLOWED_IDENTITY</td>
<td>62052028</td>
<td>641339196</td>
<td>114</td>
<td>25</td>
</tr>
<tr>
<td>READONLY_VERSION</td>
<td>50907897</td>
<td>641339196</td>
<td>114</td>
<td>25</td>
</tr>
<tr>
<td>READONLY_VALUE</td>
<td>73788021</td>
<td>641339196</td>
<td>114</td>
<td>25</td>
</tr>
<tr>
<td>READONLY_IDENTITY</td>
<td>62051084</td>
<td>641339196</td>
<td>114</td>
<td>25</td>
</tr>
</tbody>
</table>

### Bad

<table>
<thead>
<tr>
<th>Method</th>
<th>Memo Time</th>
<th>Normal Time</th>
<th># of Hits</th>
<th># of Misses</th>
</tr>
</thead>
<tbody>
<tr>
<td>WRITEALLOWED_VERSION</td>
<td>145072</td>
<td>125019</td>
<td>0</td>
<td>184</td>
</tr>
<tr>
<td>WRITEALLOWED_VALUE</td>
<td>137762</td>
<td>125019</td>
<td>0</td>
<td>184</td>
</tr>
<tr>
<td>WRITEALLOWED_IDENTITY</td>
<td>136493</td>
<td>125019</td>
<td>0</td>
<td>184</td>
</tr>
<tr>
<td>READONLY_VERSION</td>
<td>127957</td>
<td>125019</td>
<td>0</td>
<td>184</td>
</tr>
<tr>
<td>READONLY_VALUE</td>
<td>127917</td>
<td>125019</td>
<td>0</td>
<td>184</td>
</tr>
<tr>
<td>READONLY_IDENTITY</td>
<td>127873</td>
<td>125019</td>
<td>0</td>
<td>184</td>
</tr>
</tbody>
</table>

#### Listing 4.9: Output of the MemoAdvisor with three methods of the STMBench7 benchmark memoized.

All the methods that, in average, execute faster when memoized are classified as "Good" and presented in descending order of speedup. Likewise, those methods that do not benefit from memoization are presented in the section "Bad", in descending order of speedup. The time is in nanoseconds.

- **Transaction Re executions**: Every time a transaction aborts, all changes done inside it are discarded and the transaction restarts. Thus, each aborted execution negatively influences the performance of the system because it will continue to use valuable resources, until it successfully commits, and its execution time increases linearly with the number of times that it restarts.

Hence, to encourage the use of STMs it is important to lower the overheads introduced by them. With this thought in mind, the JVSTM uses speculative read-only transactions as a way to lower the overheads associated with collecting transactional information.

Given that the JVSTM targets applications where the amount of read-only transactions outnumbers the number of write transactions, it tries to improve the performance of the transactional system by speculatively assuming that each newly created transaction is read-only. This way, it saves memory and skips over the validation step when the transaction is indeed read-only. If this assumption proves to be wrong, once a write is attempted, the JVSTM aborts the transaction and restarts it as a generic read-write transaction.

Memoization, in its original formulation, is also a technique that improves the performance of repeated read-only executions. So, it would be expected that we could combine speculative read-only transactions with memoization to improve the performance of the JVSTM. Unfortunately, that is not the case because...
read-only transactions do not populate the read-set, thus not capturing shared state accesses that are important to the memoization system.

Even though we cannot combine memoization with speculative read-only transactions, I strongly believe that the synergy between memoization and STMs may in fact justify paying the extra cost of populating the read-set, when we already know that in read-only transactions the read-set is not used at commit time.

For once, memoization can be used to accelerate the reexecution of transactions that abort at the validation step. In fact, this second advantage comes for free if all transactions are memoized. In the best case scenario, there is a valid cache entry at the top-most method and all the reexecution is skipped. If not, then it may happen that another atomic methods, called by the running top-level transaction, are cached, accelerating the reexecution.

Second, because memoization gives a different end to the information already being generated by STMs, it lowers the perceived cost of constructing it in the first place. And, finally, the last advantage of this approach relates to the fact that it is implemented without the need for additional information: STMs already build the necessary data to apply memoization, we need only to extend the lifetime of such information, which is no longer discarded the moment a transaction successfully commits, being stored for future use.

In sum, I argue that this synergy between memoization and STMs is very appealing because memoization can be implemented almost for free in transactional systems and memoization constitutes a natural way to prevent redundant computations from happening after an unsuccessful commit.

It is important to highlight that Ziarek and Jagannathan [39] were the first to apply memoization in transactional environments. They centered their work solely on using memoization to prevent the reexecution of operations that do not conflict in transactional environments, as a way to accelerate forced reexecutions and not as way to speedup redundant operations, as I propose. Another difference between both works is that Ziarek and Jagannathan’s memoization system assumes that a method’s result depends solely on the list of arguments, not capturing shared state accesses.

4.6 Summary

This chapter starts with a brief overview of previous work on software transactional memories (STMs), introducing the concept of transaction, read-set, and write-set, which are used to memoize correctly methods that are not pure functions. It centers the discussion on the Java Versioned STM (JVSTM), a particular implementation of an STM that uses versioned boxes to keep multiple values of transactional locations.

After introducing the concept of STM, it describes my solution to correctly memoize imperative object-oriented methods. It shows how encapsulating a memoized method’s body inside a transaction may be used to capture shared state reads, to detect executions of memoized methods that produce side-effects, and to incorporate side-effects in the memo cache.

The chapter describes, also, the automatic transaction-oriented memoization (ATOM) system, an implementation of the memoization solution described in this chapter. The ATOM offers a simple, yet
flexible, interface based on annotations that direct a bytecode manipulation tool that automatically memoizes annotated methods.

After presenting the ATOM’s API, this chapter describes in detail how the memo cache is organized and how it validates cached relevant states. Because the ATOM allows for three caching policies—version, value, and identity—and also two memoization strategies—read-only of write-allowed—this chapter describes the relative merits of each approach.

This chapter continues by describing a memoization advisory tool that helps programmers memoize a system. This is accomplished by finding the methods of the system that execute faster when memoized and by finding the best combination of cache policy and memoization strategy to use, in a per-method basis.

Finally, the synergy between memoization and STMs is discussed, showing why memoization can be very useful in transactional contexts. First, as it uses information that is already generated by the STMs, it amortizes the cost of constructing transactional information by speeding up memoized methods with no significant overheads. Second, because it can accelerate the reexecution of conflicting transactions by avoiding repeated computations.
Chapter 5

Validation

This dissertation’s thesis is that it is possible to ease significantly the task of programming imperative object-oriented systems, through the addition of minimal, non-disruptive, and easy to use abstractions to current object-oriented programming languages. In particular, that it is possible to aid programmers in the task of improving the performance of object-oriented imperative programs through an automatic memoization tool, tailored to deal with the unique characteristics of this programming paradigm.

In Chapter 4, I presented my memoization proposal that encapsulates method’s bodies inside STM transactions to register shared state accesses. Further, I presented the ATOM system, my STM-based implementation of memoization, that allows for three different caching policies, two memoization strategies, and offers a memoization advisory tool that is able to select methods in a program that are beneficial to memoize.

With this chapter I intend to complement the thesis’ validation that was given along my memoization proposal, by using a standard benchmark to evaluate STM implementations. In particular, I propose to verify whether memoization improves the performance of systems that already use an STM, and whether the overheads imposed by using an STM in single-threaded programs do not negate the benefits yielded by memoization.

5.1 Case study: the STMBench7 benchmark

The STMBench7 benchmark [17] is a highly customizable benchmark, developed for evaluating STM implementations. This benchmark was proposed to address the lack of realistic STM benchmarks. It adapts the OO7 benchmark’s [12] data structure, removes the database-related parts, and adds new types of operations that are more adequate to the evaluation of STMs.

5.1.1 The STMBench7 Data Structure

The data structure of the STMBench7 benchmark is similar to those used by CAD programs. It consists of atomic parts, assemblies, composite parts, connections, documents, manuals, and modules. There is a single module connected to a deep trinomial tree of assemblies. Base assemblies, situated at the bottom-
level, are comprised of several composite parts, which contain a document and a graph of atomic parts. This graph is connected through connection objects allowing bottom-up as well as top-down traversals.

The standard STMBench7 benchmark comes already with a pair of lock-based strategies implemented, each varying the granularity of the mutual exclusion zone and that can be used as a base line to compare against additional synchronization strategies. Each additional synchronization strategy must provide its own version of the elements that compose the aforementioned data structure and three factories: (1) a factory in charge of creating each of the design-library objects, (2) a factory for creating the collections and indexes needed to relate design-objects with one another, and (3) a last factory to create the operations of the benchmark.

Hence, to evaluate the ATOM as a memoization tool as well as to assess if memoization improves the performance of systems that already use an STM, I created three additional synchronization strategies.

First, I created a synchronization strategy to test the JVSTM and another to test the ATOM. The JVSTM implementation encapsulates operations within speculative read-only transactions that are aborted and reexecuted as read-write transactions upon a tentative write to a versioned box, whereas the ATOM specific factory encapsulates operations within memo transactions and enforces memoization on the selected methods.

Both implementations differ only on the factory responsible for executing the operations of the benchmark as they share the transactional versions of design-library objects, collections, and indexes. Comparing to the lock-based implementation, the transactional version of design-library objects encapsulates each mutable field inside a vbox and changes each method’s body to use the versioned box operations to access the fields.

The STMBench7 benchmark uses two different types of collections: Bags, used to contain references between base assemblies and composite parts, and sets, whenever it is needed to guarantee the uniqueness of the elements inside the collection. Further, the benchmark distinguishes between two types of sets, depending on the number of elements they hold. Thus, an instance of the class SmallSet is used to contain a small number of connection and assembly elements, whereas a LargeSet instance agglomerates large numbers of atomic parts.

To accommodate such differences, I used different approaches to implement each of the collections. Thus, I used a purely functional single linked list to implement bags, JVSTM’s VLinkedSet in the implementation of SmallSets, and JVSTM’s purely functional RedBlackTree to implement the LargeSet class.

The class VLinkedSet is a transactional set, hence it guarantees transactional semantics for each operation that changes the collection. To obtain the same guarantee with the remaining two collections, that are already thread-safe because they are purely functional, I used a single vbox to hold an instance of the collection, ensuring that any operation that changes the collection first creates a new instance of that collection and then changes the vbox to that new value.

The last strategy that I created simply executes the operations of the benchmark without further concerns regarding thread synchronization. This simple and straightforward implementation suffices because it will be used in single-threaded mode only, to assess if the overheads imposed by an STM in non-concurrent programs are high enough to render my memoization system useless due to deteriorating the overall performance of the system.
5.1.2 The STMBench7 Operations

In total, the STMBench7 benchmark implements 45 distinct operations. These operations are classified according to their category and their type.

The STMBench7 benchmark has four categories for the operations, which are:

- **Long traversals**: Operations that traverse the majority of the objects in the data structure, being the lengthier in the benchmark. Because the data structure can be traversed from top to bottom, and vice versa, long traversals either start at the top-level module and traverse down the design-object graph, or start at an atomic part going bottom-up.

- **Short traversals**: Operations similar to long traversals but traversing a smaller number of objects.

- **Short operations**: Operations that access only a few design-objects and perform some simple operation on them.

- **Structure modification operations**: Operations that create or delete graph elements, thus changing the data structure.

According to their type, operations are classified as read-only or read-write. The benchmark uses this classification to define three different workloads that, in conjunction with a predefined ratio assigned to each one of the four operation’s categories, are used by each thread to randomly select which operation to execute next. Workloads define different splits between read-only and read-write operations. In a read-dominated workload 90% of the operations executed are read-only and the remaining 10% are writes. With a read-write workload the benchmark enforces 60% of read operations and 40% of write operations, whereas in a write-dominated workload only 10% of the operations executed are read-only and 90% are writes.

I changed the benchmark so that all newly created BenchThread instances, which are responsible for executing in a separate thread the operations of the benchmark, share the same array of Operation instances, sharing, as a consequence, the associated method caches as well.

5.1.3 The STMBench7 Execution Setting

For each run of the benchmark, the STMBench7 allows the specification of the number of threads, the type of workload, the synchronization strategy, and the duration of the test. Further, it is robust enough to allow us to disable certain categories of operations independently. In the original benchmark, we can disable all the long traversals, all the structure modification operations, or both. I extended it to allow us to disable only the read-write long traversals, leaving active the read-only long traversals.

At the end of a run, the benchmark outputs the total throughput of the system, that is, the number of operations processed per second by the system, and the maximum time each operation took to complete (latency).

The benchmark executes an Operation by calling the method performOperation, that either reads or updates the design-object graph, depending on the Operation instance on which this method is invoked on. I decided to memoize only the various implementations of the method performOperation because that is
Table 5.1: Output of the memo advisory tool for the STMbench7 benchmark with 14 memoized operations. The time is in microseconds. The lowest average time each operation took to complete is marked in boldface, highlighting the best combination of cache policy type and memoization strategy.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Write Allowed</th>
<th>Read Only</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Version</td>
<td>Value</td>
</tr>
<tr>
<td>Query7</td>
<td>561</td>
<td>559</td>
</tr>
<tr>
<td>Traversal1</td>
<td>51194</td>
<td>73802</td>
</tr>
<tr>
<td>Query2</td>
<td>742</td>
<td>685</td>
</tr>
<tr>
<td>Traversal8</td>
<td>1522</td>
<td>878</td>
</tr>
<tr>
<td>Query5</td>
<td>397</td>
<td>404</td>
</tr>
<tr>
<td>Traversal9</td>
<td>16</td>
<td>839</td>
</tr>
<tr>
<td>Query6</td>
<td>1102</td>
<td>1121</td>
</tr>
<tr>
<td>ShortTraversal9</td>
<td>636</td>
<td>624</td>
</tr>
<tr>
<td>Traversal6</td>
<td>145</td>
<td>138</td>
</tr>
<tr>
<td>Operation15</td>
<td>54</td>
<td>48</td>
</tr>
<tr>
<td>Traversal7</td>
<td>44</td>
<td>38</td>
</tr>
<tr>
<td>Operation6</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>Operation7</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>Operation8</td>
<td>17</td>
<td>13</td>
</tr>
</tbody>
</table>

where I believe memoization will yield the best performance boost. Unfortunately, many implementations of the method `performOperation` use random numbers to, for example, index the design-object graph. Thus, I had extra care to memoize deterministic methods only, which led, for example, to not being able to memoize structure modification operations.

I made a first run of the benchmark with all of the 14 remaining operations annotated with the `Memo` annotation and using the `TestMemoCache` to extract the beneficial methods to memoize. This preliminary test run for 120 seconds with all long read-write traversals and structural modifications disabled, under a read-dominated workload. Table 5.1 summarizes the output of the memo advisory tool.

I decided to memoize all the operations that, in average, executed faster when memoized. Therefore, I memoized the operations Query2, Query5, Query6, Query7, ShortTraversal9, Traversal1, Traversal8, and Traversal9.

It is important to note the almost complete absence of operations that are best memoized with the value cache policy. This fact may be explained by the lack of domain objects that implement the method `equals`. Therefore, if the method `equals` is not reimplemented by a class, the value cache policy behaves almost like the identity cache policy, with the extra overhead of the method call.

I present results for all three workloads and with three possible mixes of operations: (1) all operations except long read-write traversals and structural modifications, (2) all operations except long traversals (both read-write and read-only), and (3) all operations except long traversals and structural modifications.

I ran each test five times, removed both the best and the worst throughput values, and averaged the three remaining throughput values. All tests ran for 120 seconds using 1, 2, 4, 8, and 16 threads in a Dual-Quadcore Intel Nehalem-based Xeon E5520, with 12Gb of RAM running Ubuntu Linux 9.04, and Java SE version 1.6.0_13. While the test ran, no other relevant processes were executing in the system.
Table 5.2: The results of the STMBench7 benchmark with all long read-write traversals and structural modifications disabled and a read-dominated workload.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by number of threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>JVSTM</td>
<td>79</td>
<td>173</td>
</tr>
<tr>
<td>ATOM</td>
<td>1131</td>
<td>1433</td>
</tr>
<tr>
<td>Speedup</td>
<td>14.34</td>
<td>8.30</td>
</tr>
</tbody>
</table>

Table 5.3: The results of the STMBench7 benchmark with all long traversals disabled and a read-dominated workload.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by number of threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>JVSTM</td>
<td>2311</td>
<td>4126</td>
</tr>
<tr>
<td>ATOM</td>
<td>3186</td>
<td>5225</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.38</td>
<td>1.27</td>
</tr>
</tbody>
</table>

5.1.4 The STMBench7 Throughput Results: Multi-Threaded

In this section I present the throughput results of the STMBench7 benchmark under various workloads and mixes of operations. By comparing the throughput of the benchmark with the ATOM and with the JVSTM, that uses speculative read-only transactions, I want to assess if the benefits extractable from memoization are sufficient to compensate the overheads of constructing the read-set and also if memoization can be correctly applied in an imperative object-oriented context with beneficial results to the overall performance of the system.

Read-Dominated Workload

I show the throughput results obtained for the various mixes of operations and a read-dominated workload in Table 5.2 through Table 5.4. I show these values graphically in Figure 5.1 through Figure 5.3.

These results show a clear increase in performance when using memoization. The memoized version performs better than the JVSTM in almost all scenarios, achieving the best results in the first mix of operations (shown in Figure 5.1), where the throughput of the system increases by a factor of 14. The first mix of operations includes long read-only traversals, which are the most computationally intensive operations in the benchmark—the maximum time to completion of long traversals is over half a second, whereas for the remaining operations is below 9 milliseconds. So, it makes sense that memoization gives

Table 5.4: The results of the STMBench7 benchmark with all long traversals and structural modifications disabled and a read-dominated workload.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by number of threads</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>JVSTM</td>
<td>3460</td>
<td>7102</td>
</tr>
<tr>
<td>ATOM</td>
<td>6564</td>
<td>11728</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.90</td>
<td>1.65</td>
</tr>
</tbody>
</table>
Figure 5.1: Operations per second processed by the STM Bench7 benchmark with the JVSTM and with the ATOM, for a read-dominated workload with all long read-write traversals and structural modifications disabled.

Figure 5.2: Operations per second processed by the STM Bench7 benchmark with the JVSTM and with the ATOM, for a read-dominated workload with all long traversals disabled.
For the second mix of operation, as we can see in Figure 5.2, for 1, 2, or 4 threads memoization continues to behave better or at least as good as the JVSTM with speculative read-only transactions. The same cannot be said for 8 or 16 threads and, to understand these worse results it is important to remember that in this second mix of operations I turned on all structural modifications. So, the state of the system is constantly changing, reducing the number of cache hits. Further, more cache misses translates directly to more operations that are not skipped and, thus, concurrently add new cache entries, generating contention in the cache.

This problem with operations that change the structure of the design-object graph, is confirmed by the third mix of operations. I disabled once again all structural modifications and, as we can see, the ATOM outperforms the standard solution, demonstrating clear advantages of using memoization even for operations that are not computationally demanding.

With the second mix of operations, we can see a reduction in performance for both the JVSTM and the ATOM, with 16 threads. This result is typical of the STMBench7 when we have more threads than available processors,¹ because the number of conflicts rises and so do the number of restarted transactions.

Overall, these results show that memoization is able to improve significantly the performance of read-dominated workloads. My solution scales almost perfectly and, given that the STMBench7 benchmark traverses the object graph but performs no operations on the leaves, it is reasonable to expect better results under a more realistic test because the unmemoized version of the system would take longer to complete each operation.

¹It is important to remember that the processor used in these tests can run up to 8 real threads or 16 logical threads with hyperthreading.
Read-Write Workload

Both speculative read-only transactions and memoization are optimization strategies best fitted to read
intensive workloads, so augmenting the number of write operations theoretically translates to a decrease
in performance obtained from these techniques. Hence, it is interesting to see which approach better
adapts to a scenario where more transactions are wrongly assumed as read-only, resulting in more aborts,
and where the state of the system constantly changes, invalidating cached memo entries.

In Table 5.5 through Table 5.7 I show the throughput results obtained for the same various mixes
of operations as before but with a read-write workload. These values are graphically represented in
Figure 5.4 through Figure 5.6.

As far as memoization is concerned, augmenting the number of write operations not only has an
impact on the validity of cached information, but also under this workload the benchmark runs less
memoized operations. Nevertheless, overall these results show that memoization continues to behave
better than speculation. Once again, the first mix of operations is where we obtain the most representative
improvements, with speedups that range from 9 to 4 times better than the JVSTM.

In terms of speedup, the remaining results are not as impressive but good nonetheless, with increases
in the order of 10% and 60% with the second and third mixes of operations, respectively. Once again,
structural modifications negatively influence memoization’s performance but, overall, what these values

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>JVSTM</td>
<td>1245</td>
</tr>
<tr>
<td>ATOM</td>
<td>1422</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.14</td>
</tr>
</tbody>
</table>

Table 5.6: The results of the STMBench7 benchmark with all long traversals disabled and a read-write
workload.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>JVSTM</td>
<td>2277</td>
</tr>
<tr>
<td>ATOM</td>
<td>3681</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.62</td>
</tr>
</tbody>
</table>

Table 5.7: The results of the STMBench7 benchmark with all long traversals and structural modifications
disabled and a read-write workload.
Figure 5.4: Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-write workload with all long read-write traversals and structural modifications disabled.

Figure 5.5: Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-write workload with all long traversals disabled.
No traversals / No structural modifications

![Graph](image)

**Figure 5.6:** Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a read-write workload with all long traversals and structural modifications disabled.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by number of threads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>JVSTM</td>
<td>562</td>
</tr>
<tr>
<td>ATOM</td>
<td>2175</td>
</tr>
<tr>
<td>Speedup</td>
<td>3.87</td>
</tr>
</tbody>
</table>

**Table 5.8:** The results of the STMBench7 benchmark with all long read-write traversals and structural modifications disabled and a write-dominated workload.

confirm is that with a read-write workload, as with read-dominated workloads, even in the worst case scenarios, the benefits extracted from memoization are enough to compensate the overhead of constructing transactional information.

**Write-Dominated Workload**

In the final test I explored to the extreme the vulnerabilities of both memoization and speculation with a write-dominated workload. I show the results obtained for the throughput tests for the various mixes of operations in Table 5.8 through Table 5.10. Once again, I present these values graphically, in Figure 5.7 through Figure 5.9.

Memoizing a system with only 10% of read operations and with many methods that constantly change shared state, I was expecting to see the costs of cache operations influencing negatively the performance of the system. In part that is what these results show, with the worst overall throughput results of the three tested workloads. Nonetheless, the ATOM continues to be the best choice.

These results also show a very interesting characteristic. For the first time, the ATOM consistently
Table 5.9: The results of the STMBench7 benchmark with all long traversals disabled and a write-dominated workload.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVSTM</td>
<td>730</td>
<td>1047</td>
<td>1213</td>
<td>959</td>
<td>602</td>
</tr>
<tr>
<td>ATOM</td>
<td>1084</td>
<td>1401</td>
<td>1499</td>
<td>1121</td>
<td>806</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.48</td>
<td>1.34</td>
<td>1.24</td>
<td>1.17</td>
<td>1.34</td>
</tr>
</tbody>
</table>

Table 5.10: The results of the STMBench7 benchmark with all long traversals and structural modifications disabled and a write-dominated workload.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td>JVSTM</td>
<td>1608</td>
<td>2659</td>
<td>3323</td>
<td>3347</td>
<td>1857</td>
</tr>
<tr>
<td>ATOM</td>
<td>3384</td>
<td>5097</td>
<td>6050</td>
<td>4912</td>
<td>3041</td>
</tr>
<tr>
<td>Speedup</td>
<td>2.10</td>
<td>1.92</td>
<td>1.82</td>
<td>1.47</td>
<td>1.64</td>
</tr>
</tbody>
</table>

Figure 5.7: Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a write-dominated workload with all long read-write traversals and structural modifications disabled.
Figure 5.8: Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a write-dominated workload with all long traversals disabled.

Figure 5.9: Operations per second processed by the STMBench7 benchmark with the JVSTM and with the ATOM, for a write-dominated workload with all long traversals and structural modifications disabled.
behaves better than the JVSTM. These results show that in the best case, with all the long read-write traversals and structural modifications disabled, we get a speedup of 4. In the worst case, memoization only improves the performance of the system by 17%, but the majority of these results show an improvement of over 50%.

With many write operations, many cache queries miss and, at method’s end, new entries are added to the memo cache. In concurrent contexts, where many threads try to update the cache, even in read-only methods the memo cache represents a contention point of the system. This problem is even more noticeable in the second and third mix of operations, where the decrease in performance can be seen as early as with more than 4 threads.

I believe that this behavior may be explained not only by the fact that all the memoized methods are short-lived, hence each thread is constantly blocked in cache updates, but also because in multi-threaded scenarios each second-level cache search validates several relevant states that, due to the write-dominated nature of the workload, are, most of the time, all invalid.

### 5.1.5 The STMBench7 Throughput Results: Single-Threaded

My memoization proposal heavily depends on the support offered by an STM and, as I stated in Section 4.5, can be implemented almost for free in systems that already use an STM. Because memoization is to be used also in single-threaded environments, where the introduction of an STM is semantically irrelevant but computationally heavy, it is important to assess if the overhead imposed by the usage of a software transactional memory to capture shared state accesses is high enough to negate the performance benefits extractable from memoization.

I show in Table 5.11 through Table 5.13 the throughput results for each of the various workloads and operations mixes in a single-threaded environment with and without memoization.

As we can observe in Table 5.11, my memoization solution is able to improve the performance of the system at most by a factor of 7, in a read-dominated workload, or at least by a factor of 2, in a write-dominated workload.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by workload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read-dominated</td>
</tr>
<tr>
<td>Vanilla</td>
<td>154</td>
</tr>
<tr>
<td>ATOM</td>
<td>1131</td>
</tr>
<tr>
<td>Speedup</td>
<td>7.35</td>
</tr>
</tbody>
</table>

**Table 5.11:** The results of the STMBench7 benchmark with all long read-write traversals and structural modifications disabled.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by workload</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read-dominated</td>
</tr>
<tr>
<td>Vanilla</td>
<td>6291</td>
</tr>
<tr>
<td>ATOM</td>
<td>3186</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.51</td>
</tr>
</tbody>
</table>

**Table 5.12:** The results of the STMBench7 benchmark with all long traversals disabled.
Table 5.13: The results of the STMBench7 benchmark with all long traversals and structural modifications disabled.

<table>
<thead>
<tr>
<th>Synchronization strategy</th>
<th>Operations / second by workload</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>read-dominated</td>
<td>read-write</td>
<td>write-dominated</td>
</tr>
<tr>
<td>Vanilla</td>
<td>7719</td>
<td>5768</td>
<td>4409</td>
</tr>
<tr>
<td>ATOM</td>
<td>6564</td>
<td>3681</td>
<td>2284</td>
</tr>
<tr>
<td>Speedup</td>
<td>0.85</td>
<td>0.64</td>
<td>0.77</td>
</tr>
</tbody>
</table>

These results are not as good as those shown before in Section 5.1.4, where we obtained a speedup of 14, and get even worse under the other two mixes of operations because the use of memoization only deteriorates the performance of the system, but are promising enough to advise programmers to use the ATOM.

We can conclude that, as expected, even though in a single-threaded environment transactions never conflict or abort, overall, STMs introduce a significant overhead to the system, mainly due to intercepted memory accesses. Memoization improves the performance of the system under the first workload because this is the only workload where read-only long traversals are enabled, which, when memoized, may skip large chunks of work.

Thus, in single-threaded systems, the amount of work that memoization skips must be large enough to compensate not only the time spent on cache operations, but also the overheads imposed by the use of a software transactional memory. When it happens, these results show that the ATOM is able to improve significantly the performance of a program.

5.2 Summary

In this chapter I presented an extensive performance evaluation of the ATOM, using a standard benchmark for evaluating STMs—the STMBench7 benchmark.

Due to the complexity of the benchmark, that has more than 40 possible operations, each with several methods, I used my memoization advisory system to extract which of the available operations are beneficial to memoize. The results obtained after this preliminary step show the effectiveness of my advisory tool, both in the selection of which methods can benefit from memoization and also in the reduction of the time spent on analysing and deciding which methods to memoize in complex systems.

In the context of software transactional memories, the results show that memoization, even though it needs to populate the read-set, behaves better than speculative read-only transactions on all tests. More specifically, the ATOM is up to fourteen times faster than the JVSTM with speculation. Also, we concluded that the ATOM adapts fairly well to workload changes, showing a performance boost even in write-dominated executions of the benchmark.

In single-threaded environments we concluded that STMs introduce a significant overhead that limits the applicability of memoization. The amount of work skipped by memoization must be sufficient to encompass the usual cache operations and the overhead of intercepting and registering memory accesses, as well. When they do, as the results show, despite the type of workload, memoization can improve the performance of a system.
To summarize, these results show that, as long as there are read-only repeated operations, memoization is useful to improve the performance of the underlying system, regardless if, as a whole, the system is functionally pure or not. With such strong results, I have shown that programmers can obtain the same beneficial characteristics of memoization in imperative object-oriented contexts, with little effort, and flexibility regarding which cache policy and memoization strategy to use.
Chapter 6

Conclusions

This concluding chapter outlines the major contributions of my work, some limitations of my implementation of the proposals made through this dissertation, and discuss what lies ahead in terms of future research.

6.1 Main Contributions

This dissertation’s main goal was to make memoization more appealing and appropriate to the unique characteristics of imperative object-oriented programs. In particular, I identified the difficulty of constructing the list of relevant values for the output of a method, when this list includes the arguments of the method as well as values from shared mutable state, as the major obstacle to using memoization in imperative object-oriented systems. The fact that methods may also write to shared state, i.e., produce side-effects, is an additional factor that hinders and influences the applicability of memoization in imperative programs.

Throughout this dissertation I described the ATOM system, an automatic memoization tool that uses software transactional memories (STM) to capture automatically the relevant state for a memoized method’s result, identify side-effect-free methods that are safe to memoize, and capture write operations to shared state so that they can be reapplied in future reexecutions of the method, thus extending memoization even to methods that are not referentially transparent. Furthermore, the implementation of this extended memoization is transparent and easy to use, needing minimum knowledge from programmers.

Because STMs already do all of the expensive work of collecting the relevant information for the memoization tool, this extended memoization approach comes for free for a system that already uses an STM. Moreover, because it increases the performance of the system at almost no extra cost, it amortizes the upfront cost of using an STM, thereby promoting the adoption of STMs.

The major contributions of this dissertation in the area of memoization are the following:

• I described an STM-based solution that automatically identifies the relevant values for the output of a method, including not only the list of arguments but also the values belonging to shared mutable state, as well. My proposal encapsulates the body of memoized methods inside an STM transaction so that, at transaction’s end, the transaction’s read-set holds all the locations and respective values
read by the method. This information can then be incorporated in the cache-key and used in future executions to validate cached entries.

- I proposed a runtime decision algorithm that is capable of determining if a method’s execution can be memoized, based on the fact that it produces no side-effects. My novel solution, unlike previous dynamic purity analysis tools that monitor a method’s execution looking for any potential instructions with side-effects, encapsulates memoized methods inside a transaction and uses the information registered in the transaction’s write-set to determine their purity. An empty write-set identifies a side-effect-free method, whereas a non-empty write-set classifies a method as not side-effect-free.

- I defined the semantics associated with the memoization of methods with side-effects. To replicate the behavior of a memoized method that produces side-effects, memoization needs not only to substitute the invocation of the method with its respective return value, but also reapply all the changes that would be done by the method if it executed normally.

- To implement my proposal of memoizing methods that produce side-effects, I proposed the incorporation of a transaction’s write-set into the method cache has a way to reify side-effect operations. This way, upon a cache hit, if the original execution of the method produced side-effect, the memoization system can replicate the changes done in the original run of the method iterating over the cached write-set.

- I contribute with a thread-safe automatic memoization tool—the ATOM—specifically designed to accommodate the unique characteristics of imperative object-oriented systems. The ATOM implements my proposals regarding automatic construction of the relevant state, dynamic detection of side-effects, and integration of side-effects in memoization by encapsulating a method’s body inside a transaction. The ATOM offers flexibility to programmers, allowing them to choose between three caching policies types—version, value, or identity—and two memoization strategies—read-only or write-allowed. Because the ATOM has a minimalistic interface, it is easy to understand and to use by programmers.

- I contribute with an annotation-driven automatic instrumentation tool that is flexible, customizable, and simple to use. Because all the intended transformations are done at compile time and at bytecode level, the ATOM facilitates the use of memoization without introducing runtime overheads.

- I developed a memoization advisory tool to aid programmers find the methods that can benefit from memoization. Because the ATOM offers a three caching policies and a couple of memoization strategies, the advisory tool not only lists methods that memoization can accelerate but also what is the best combination of caching policy and memoization strategy to use, in a per-method basis.

In the area of Software Transactional Memory I described in detail how memoization can be implemented almost for free in transactional systems and showed how it can help reduce the perceived cost of constructing transactional information by giving it a different end. Further, I showed how memoized methods guarantee correct transactional semantic upon a cache hit, with a lightweight optimistic approach that populates the read-set only if proved that such information will be necessary at commit time.

Finally, in the area of imperative object-oriented programming I showed how memoization can be safely applied in complex systems even with methods that read or write values to shared-state, and demonstrated that, even though memoization is nowadays seldom used in imperative contexts, it is a very useful optimization technique, that may be useful outside of the functional programming paradigm.
6.2 Limitations

The current implementation of the ATOM still presents a small number of limitations, not yet discussed in this dissertation. In this Section I explain what these limitations are, under what circumstances they happen, and how they influence the behavior of the ATOM.

6.2.1 Local Instances of Transactional Objects

A transactional object, using the JVSTM encapsulates mutable class slots inside versioned boxes. Thus, at creation time, normally the constructor of a transactional object will instantiate a versioned box for each slot of the class and populate it with its initial value, as well. Hence, creating a transactional object inside a transaction adds information to the write-set.

If that newly created object is local to a method, it is then obvious that its lifetime is bounded by the lifetime of the method where it is created. The definition of side-effect clearly states that a write operation is only considered a side-effect if it causes an observable change in the system, thus the creation of local transactional objects, or any modification made to them, is not a side-effect because no other method, besides those invoked in the context of the method that created the object, will see the object nor the changes made to it.

The problem is that the ATOM uses the write-set to classify methods as side-effect-free, so it will wrongly classify this type of methods as producing side-effects, and not cacheable if the memo strategy is read-only. In fact, this limitation is not a limitation of the ATOM itself, as it is of the JVSTM and arises as a consequence of its use. To the JVSTM, methods with the aforementioned behavior result in read-only transactions being aborted and reexecuted as read-write, even if the only write operations done by the transaction are to local transactional objects, and forces the validation of the read-set at commit-time, what may lead to another needless restart because the only changes made inside do not influence the state of the application.

6.2.2 Input/Output

Up until this point, I used the concept of side-effect as a write to shared memory that is visible outside of the method that produced it. But side-effects encompass not only changes to memory values, but input and output operations as well. The problem is that these I/O operations are not captured nor identified by the ATOM, thus, if executed within a memoized method, they will not be detected nor replicated upon a cache hit.

This problem, in the context of transactional systems, is solved because the need to abort failed transactions leads programmers to be aware that the atomic portion of their system cannot perform any operation that cannot be undone. I/O operations are examples of undoable operations. So it is safe to assume that I/O operations are not a problem because they are not transactional and should not be used in atomic sections.

The same cannot be said about a single-threaded system that do not use an STM and is transactified to use the ATOM. In this case, methods may execute I/O operations that, if memoized, are skipped, potentially changing the semantic of the system.
As discussed in Section 3.6.1, sometimes skipping an I/O operation can be acceptable, unimportant, or beneficial, depending only on the intent of programmers, thus using the ATOM is such cases is not harmful. When it is not, programmers should be extra careful with the choices that they make.

6.3 Future Research

Although the work described in this dissertation is self-contained and the ATOM system, at its current state, may be used without further developments, I finish this dissertation stating which new ideas will be explored in the future and discuss what lies ahead in terms of future research.

6.3.1 Read-Set Invalidation Strategy

My strategy to validate a cached read-set queries each versioned box belonging to the read-set and assesses if its current value is the same as the value stored in the cache for it. This strategy is computationally heavy, specially in read-dominated workloads where the stored read-sets are almost always valid because the versioned boxes that they hold rarely change.

The alternative is to invalidate the read-set when its relevant state changes. To do so, we need to store in each versioned box the list of cache entries that depend on the value of the box and, each time the box changes, it informs all its dependents that it has changed value. The memo cache acts accordingly by removing the entry. This way, to search for a cache hit it suffices to acquire a second level memo entry from the memo cache knowing that, if it exists, it is valid.

This new alternative to verify if a cache entry is still valid will also allow us to introduce the concept of calculated attributes in the programming model. Often programmers are aware that some value is constantly calculated, despite the end result value being the same. So, they create a field to hold the calculated attribute and recalculate it every time that the values it depends on change. The problem with this handmade solution, as usual, is that it is error-prone because programmers may forget to recalculate the calculated attribute.

Taking advantage of memoization in general, and of the invalidation strategy in particular, we could create a new specialized method to compute the calculated attribute and memoize it, knowing that the invalidation strategy guarantees a recomputation every time the calculated attribute is accessed after the state it depends on changes. This cannot be done with the current validation strategy because every access to a calculated attribute would need to validate the read-set, not mimic the behavior of the handmade solution.

The attribute balance in the class Account is one such example. We could model the domain to include the concept of account movement and change the concept of account to include a list of all the movements (deposits and withdraws) done to that particular account. This way we do not need a slot balance in the class Account because the balance of an account would be given by the sum of all the registered movements.
6.3.2 Runtime Choice of Methods to Memoize

As it was extensively discussed in this dissertation, choosing the best methods to memoize is hard. This difficulty can be in part explained because the result of this process depends on the nature of the methods themselves and on the runtime workload of the system, as well. Thus, to aid in this process I proposed a memory advisory tool that, by pre-selecting some methods and with a pre-run of the system, helps programmers make a more informed decision.

Unfortunately, this is not the best solution in all cases, because choices done statically may not be well suited to runtime changes in the workload of the system. Further, if after choosing the best methods to memoize, the code of the application changes, programmers need to rerun the advisory tool and make the appropriate modifications.

To solve this problem the ATOM needs a runtime decision algorithm able to temporarily disable memoization in the system, unmemoize some methods, and even memoize others that it sees profitable. All with no programmers’ intervention.

To do so, the decision algorithm must collect run-time information. For example, monitoring the ratio between hits and misses in memoized methods, the number of write transactions, or the time that a method takes to complete if the cache yields a hit or a miss, in a per-method basis.

Fragments of this information are already being generated, needing to be aggregated in a central decision system and then paired with some heuristics. This way, the memoization system can be made completely transparent to the programmer and autonomic.

6.3.3 Transaction Reexecution

I strongly believe that the synergy between STMs and a memoization system implemented using an STM may be extremely beneficial for STMs. In this dissertation I concentrated on applying memoization in read-dominated workloads but, as mentioned in Section 4.5, memoizing transactions is also beneficial in read-write or write-dominated workloads because it may accelerate transaction reexecution after an invalid commit.

Although faster reexecutions come for free, as a result of some transactional methods being memoized, it is limited by the number of memoized transactions called by the aborting transaction. At commit time, a flat memoized transaction once marked to abort will restart and reexecute completely, increasing the amount of work that must be redone in the presence of a restart, whereas nesting offers more flexibility at the expense of creating more transactions, merging read-sets and write-sets when nested transactions commit, and associating memo caches to methods that otherwise would not be beneficial to cache, just to accelerate reexecutions.

So, for this particular use of memoization, programmers must balance the relation between memoized and unmemoized methods because numerous cache searches may deteriorate the overall performance of a transaction, but too few can lead to a lot of work that must be redone if that transaction ever aborts. Programmers may need, as well, to change the structure of their code, creating methods just to offer more memoization points.

Therefore, as a first approach, it would be interesting to further integrate memoization in the transactional system, creating a memoization subsystem that starts by memoizing all methods invoked by a
transaction and then maintains caches local to each transaction. These local caches are used to accelerate only the reexecution of a transaction, if it eventually aborts, being discarded as soon as the transaction successfully commits.

As an evolution of this idea, and to alleviate programmers of the chore of artificially creating new methods so that they can be memoized, it would be interesting to explore some automatic code extraction approaches to isolate segments of the code that usually do not contribute to the abort, that is, code segments that usually do not read values concurrently being modified by some other committed transaction, still holding the same value in the reexecution.

Taking notice that aborts happen only because a read-write transaction read something that another transaction wrote and committed, another alternative would be to partly abort a transaction. This partial abort discards all computations affected by boxes that caused the conflict, and restarts the transaction from the point it is known to be valid.

The idea to use checkpoints in STMs to achieve partial aborts is not new, Cachopo and Rito-Silva [11] proposed nested transactions has a way to simulate checkpoints, but this approach is limited to the number of nested transactions executed inside an aborted transaction. Thus, I propose the use of continuations to allow partial aborts. By creating a checkpoint whenever a new value is about to be added to the read-set, upon a conflict we could restart the transaction from the earliest valid checkpoint, that is, the point before the addition to the read-set of the earliest box that conflicted.
Bibliography


