TÉCNICO LISBOA

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Speculative Parallelization of Object-Oriented Applications

Ivo Filipe Silva Daniel Anjo

Supervisor: Doctor João Manuel Pinheiro Cachopo

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Merit

2015

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

Speculative Parallelization of Object-Oriented Applications

Ivo Filipe Silva Daniel Anjo

Supervisor: Doctor João Manuel Pinheiro Cachopo

Thesis approved in public session to obtain the PhD Degree in
Information Systems and Computer Engineering

Jury final classification: Pass with Merit

Funding Institutions
Fundação para a Ciência e a Tecnologia

2015

Resumo

Apesar da ubiquidade dos processadores *multicore*, e ainda que novas aplicações informáticas estejam a ser desenvolvidas para tirar partido das suas capacidades de processamento paralelo, muitas das aplicações mais comuns são ainda sequenciais. Como consequência, muitos dos designers de processadores, ainda que activamente publicitando e recomendando a utilização de programação paralela, dedicam ainda muitos dos seus recursos à obtenção de melhores velocidades de processamento para código sequencial. Esta persistência mostra que o desempenho na execução sequencial é ainda um importante desafio em sistemas modernos.

Nesta dissertação, proponho a criação de uma *framework* para paralelização automática com o objectivo de acelerar a execução — em *multicores* — de aplicações sequenciais orientadas a objectos que se executem em plataformas *managed*. A natureza fortemente dinâmica e irregular deste tipo de aplicações e das plataformas em que assentam derrota muitas das técnicas tradicionais de paralelização, requerendo uma aproximação diferente para se conseguir ter sucesso.

Para permitir a execução especulativa de código com baixo impacto na performance, proponho um novo modelo de Memória Transaccional por Software especializado para paralelização especulativa. Para permitir à *framework* proposta a utilização de Especulação ao Nível do Método, proponho um novo algoritmo para transformar chamadas a métodos em pontos de início de especulação que retornam futuros. Examino ainda os problemas com filas de execução e mapeamento de tarefas para *threads*, e introduzo um conjunto de novas técnicas que permitem reduzir os tempos de espera e que aumentam a eficiência da utilização de recursos da máquina. Finalmente, descrevo a implementação das técnicas propostas na criação de um sistema coerente — a *framework* JaSPEx-MLS — que proporciona uma solução integrada para a aceleração da execução de aplicações Java/JVM sequenciais em hardware comum, sem necessidade de alterações prévias às mesmas, e com performance capaz de superar JVMs comerciais.

Abstract

Despite the ubiquity of multicores, and although new applications are being developed to take advantage of multicore systems, many commonly-used applications are still sequential. As a consequence, many chip designers, while actively advocating parallel programming, still dedicate many of their resources to providing faster sequential code execution. This shows that sequential code execution performance is still an important challenge for modern computing systems.

In this dissertation, I propose the creation of an automatic parallelization framework that boosts the execution of sequential object-oriented applications running atop managed runtimes. The highly dynamic and irregular nature of these applications and their supporting runtimes requires a different approach to parallelization that defeats many traditional parallelization techniques.

To support low-overhead speculative code execution, I propose a novel Software Transactional Memory model that is specialized for speculative parallelization. To allow the proposed framework to use Method-Level Speculation as a task identification strategy, I propose a new algorithm that allows the safe transformation of method call sites into spawn points that return futures. Then, I examine the issues of runtime task queuing and thread mapping, and introduce several new techniques to reduce stalling and increase resource usage efficiency. Finally, I describe the combination of the proposed techniques into a coherent system — the JaSPEx-MLS framework — that provides an end-to-end solution for boosting sequential Java/JVM applications on commodity hardware, without requiring applications to be previously modified, and yielding performance capable of surpassing production Java runtimes.

Palavras-chave

Paralelização Automática

Especulação ao Nível do Método

Aplicações Orientadas a Objectos

Memória Transaccional em Software

Máquina Virtual Java OpenJDK HotSpot

Arquitecturas Multicore

Keywords

Automatic Parallelization

Method-Level Speculation

Object-Oriented Applications

Software Transactional Memory

OpenJDK HotSpot JVM

Multicore Architectures

Acknowledgments

How much can I change on this journey? — A question that I didn't consciously ask myself when I decided to embark on this challenge, but that I feel has been hoisted on the flagship since day one. *A lot*, as it turns out. I have become a better engineer, a better scientist, and a better person.

The trip has been long and hard, but in the end I feel that this work is really my own, and it shows all my ideas, enthusiasm, and skills for getting it working. For a number of years I will be able to brag about knowing things about Java and the Java platform that no-one should have to know.

I hope that others will be able to learn from both my successes and my mistakes, and that future research will be better off than when I started my work, all the way back in 2008.

I would like to extend greetings and general *kudos* to my colleagues for putting up with me for the best part of 6 years (10! for some): Hugo, Stoyan, Luís and Sérgio; also other guys from ESW, GSD, the room across, and the new blood from INESC-ID room 138. I put up with you too, so don't complain too much :)

A big hug and thanks goes to Telma. Thanks for keeping me company all those lunches, thanks for kicking my butt when I deserved, and thanks for listening when I needed someone to talk to. We're almost out of here, home stretch!

I would not be here if it were not for the support from my advisor, Professor João Cachopo. His insights and steering were invaluable for the success of this work, and his technical skills and engineer mind are among the best I had the pleasure to cross paths with. I wish you the best on your UK adventure, and well, here's hoping I'll get to it someday too ;)

Last, but by no stretch of mind least, my friends and family. Friends, past and present: Thanks are never enough. Francisco, Eduardo, Tiago, Hugo, Camila, João^2, Ivo, Rafael, Luena, Patrícia, Ana. I love you seems to not really cover it. As I sit here writing this text, I fail to see how these characters can convey my deep thanks for your support. You know who you are. Thanks for being a part of my life, and hope you've enjoyed doing so.

Thanks to my younger sister, Mafalda. You've grown so much these past years — I have to say I kinda envy you! Thanks to my father, Carlos, and to Dulce, for believing in me and giving me a place where I can always go back to. Thanks to Mário, who doesn't get enough credit for setting me straight one too many times and for teaching me to go further. A wink goes to Márcia, the latest entry on this list. Where were you? You almost didn't make it in time to be present on this list, you late *baka*-head. I love you!

Finally, I dedicate this dissertation to my mother, Isabel, and to my grandfather, Ivo, who are no longer amongst us. Your departure left me with great sorrow, but I like to think that you'd approve of where I am now and what I decided to do with my life. I daresay you'd be proud.

Ivo Anjo
July 2014

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations and Acronyms

API	*Application Programming Interface*
CMP	*Chip Multiprocessor*
TM	*Transactional Memory*
HTM	*Hardware Transactional Memory*
STM	*Software Transactional Memory*
TLS	*Thread-Level Speculation*
MLS	*Method-Level Speculation*
RVP	*Return Value Prediction*
VM	*Virtual Machine*
JIT	*Just-in-time*
GC	*Garbage Collection/Collector*
JDK	*Java Development Kit*
JNI	*Java Native Interface*
JRE	*Java Runtime Environment*
JVM	*Java Virtual Machine*

Chapter 1

Introduction

Having processors capable of working on multiple tasks concurrently has been generally accepted as the next step in computing: Multicore processors now range the entire device spectrum, from the largest supercomputers to the cellphones on our pockets and the smart watches on our wrists. Modern applications are expected to use the multiple cores at their disposal to perform their work and each new CPU generation is expected to bring more and more cores for them to work with.

But why is it then, that many hardware manufacturers are still trying to squeeze better sequential performance out of their designs? Why are most commonly-available multicore machines still on the single-digit number of cores, with manufacturers preferring a small number of ever more complex processor cores over a bigger number of simpler ones, almost a decade after the first commonly-available dual-core desktop CPU designs?[1]

What is holding back the manycore era, keeping it at bay?

Unfortunately, although some application classes can be parallelized very effectively using hundreds of CPU or GPU cores, there are still many applications that are either sequential or that only make use of a few processor cores. Because of this, and to this day, chip designers are still going to great lengths to extract further sequential performance from their chips:

- The ARM processor architecture was historically developed to be both simple to implement and low-power to run, and was aimed at embedded designs. Over the last decade, ARM processors made up the majority of PDA, smartphone and tablet processor designs. Alas, performance concerns have led the once-simple designs to evolve considerably: 2005 saw the introduction of the ARM Cortex-A8,[2] the first superscalar ARM processor; just two years later, 2007 saw the unveiling of the ARM Cortex-A9,[3] the first out-of-order ARM processor. Modern ARM-based designs still use a single-digit number of complex cores, with flagship smartphones and tablets hitting clock rates as high as 2.7GHz.

- In 2013, after competitor Mediatek announced the first commercially-available octa-core ARM designs, Qualcomm, the leader in ARM processor shipments published a marketing

[1]Both the Intel Pentium EE/Pentium D *Smithfield* and the AMD Athlon 64 X2 were introduced by Intel and AMD (respectively) in the first half of 2005.

[2]*ARM Introduces Industry's Fastest Processor For Low-Power Mobile And Consumer Applications*, http://arm.com/about/newsroom/10548.php (2005)

[3]*ARM Unveils Cortex-A9 Processors For Scalable Performance and Low-Power Designs*, http://arm.com/about/newsroom/18688.php (2007)

video criticizing its competitor, entitled "Better Cores, Not More Cores", where they state that "More cores won't make your apps move faster. Faster cores will make your apps move faster".[4]

- When in 2008 Intel needed a low-power processor aimed at embedded and low-cost systems, it introduced the Intel Atom based on the Bonnell microarchitecture, the first Intel processor to issue instructions in-order since the Pentium Pro/Pentium II. Years later, in 2013, its replacement — Silvermont — reintroduced an out-of-order architecture.

There are many similar examples of processors that purposely started out as simple, in-order affairs, and that eventually evolved into their present-day complex superscalar and out-of-order designs — in some cases, by even sacrificing core count.[5]

In addition, most modern processor designs from both AMD and Intel support *boost* modes:[6] Not only can they scale down their operating frequency whenever the system load is low as a means of power saving, but they also support the reverse operation — they are able to scale up and above their normal operating frequency whenever some conditions are met, effectively performing automatic overclocking. Usually, these boost modes kick in when processor usage is highly unbalanced — whenever a single thread is doing a lot of work while other CPU cores are idle or lightly used.

All the above examples show that chip manufacturers, while continuing their push for developers to target parallel designs, are still willing to spend considerable resources to extract even a small amount of extra sequential performance.

This continued thirst for sequential execution performance is a result of the development of better-suited concurrent programming languages, models, and tools not being able to keep up with the needs of the multicore computing era. It is still hard for a typical programmer to design and build performant, correct, and portable concurrent applications; it is even harder for him to parallelize an existing legacy application.

A possible solution for this predicament is automatic parallelization, which shifts the job of making an application execute concurrently away from the programmer and into a compiler or parallelization framework: Parallelizing compilers such as those presented in [Wilson et al., 1994, Blume et al., 1996] work by breaking up the application code into several independent tasks (usually the iterations of a loop), and then checking that they do not interfere with each other. If the compiler is able to prove non-interference of the tasks, the application is changed to execute them in parallel. Such compilers have been successful for application domains that make use of regular loops and simpler data structures that can be statically analyzed, as is the case with many scientific applications.

Because the approach taken by classic parallelizing compilers hinges on their being able to prove statically that a given parallelization is valid, it is unsuitable for irregular applications — such as object-oriented applications — that employ techniques such as dynamic data structures,

[4] *Snapdragon –Better Cores, Not More Cores*, http://youtu.be/qdauwqhmsas (2013)

[5] The Nvidia Tegra K1 will be available as either a 32-bit 3-way superscalar 2.3GHz quad-core, or as a 64-bit 7-way superscalar 2.5GHz dual-core, http://www.tomshardware.com/reviews/tegrak1,3718.html

[6] *AMD Divulges Phenom II X6 Secrets, Turbo Core Enabled*, http://anandtech.com/show/3641/; *Intel® Turbo Boost Technology—On-Demand Processor Performance*, http://intel.com/content/www/us/en/architecture-and-technology/turbo-boost/turbo-boost-technology.html; See also [Charles et al., 2009]

loops with complex dependences/control flows, and polymorphism [Lam and Wilson, 1992, Oplinger et al., 1999, Kulkarni et al., 2009b].

We can extend automatic parallelization to support irregular applications through the use of speculative execution. Speculative parallelization systems work by running parts of the application in parallel even if the parallelization system is not able to prove statically that the result will be correct. Instead, correctness validations are dynamically performed at run-time, during or after execution of parallel tasks. To implement this strategy, automatic parallelization systems rely on mechanisms that buffer and track memory read and write operations in a manner similar to Transactional Memory (TM) systems. These mechanisms need to provide both the ability to validate the results from each task and the ability to undo the results from erroneous executions.

There have been numerous speculative parallelization systems proposed. Many (e.g. [Oancea et al., 2009, Mehrara et al., 2009, Spear et al., 2010, Raman et al., 2010, Mehrara et al., 2011]) target parallelization of loops and very small code blocks; others (e.g. [Steffan et al., 2000, Chen and Olukotun, 2003, Liu et al., 2006, Yoo and Lee, 2008]) depend on non-standard hardware with specialized speculative features. This leaves out commonly-available modern multicore hardware and many dynamic, irregular, and object-oriented applications, which possess rich object domain models, and that make full use of class hierarchies, interfaces, and other object-oriented programming features.

With my work, I aim to address this issue by developing techniques suitable for speculatively parallelizing object-oriented applications that run atop managed runtimes, and I propose to implement them in the creation of a novel software-based automatic parallelization framework that boosts the execution of sequential object-oriented applications.

1.1 Thesis Statement

This dissertation's thesis is that it is possible to design and build a practical speculative parallelization framework for sequential object-oriented applications that run atop a managed runtime. Such a framework need not depend on special speculation-aware hardware, nor on programmer assistance or access to the application's source code.

To validate this thesis, I propose the creation of a novel Java/JVM-based automatic framework that works at the Java bytecode level, employs Transactional Memory to allow safe software-based speculative code execution, and works atop a high-performance virtual machine runtime.

I define as being practical a framework that achieves speedup when compared to executing the same application using a common virtual machine environment. If a speculative parallelization framework is able to achieve speedup when compared to a sequential execution using a simpler virtual machine, but is still slower than commonly-available production virtual machines from Oracle or Microsoft, then it is not practical.

Creating a practical framework entails effort in both developing parallelization techniques suited for the target applications and in solving the engineering issues posed by their needed high-performance and low-overhead implementations.

While it should not be considered as a replacement for a properly-designed parallel application, the proposed framework will help to bridge the growing divide between sequential and parallel application performance: It should be seen as an alternative to (but can also work as a complement for) the hardware boost modes provided by modern chips from Intel and AMD that take advantage of unused CPU cores to improve sequential code execution performance at the expense of added energy consumption.

By not requiring special transactional or speculation-aware hardware, the proposed framework can be executed on readily-available commodity multicore hardware.

By working without access to the source code, and by making optional the input from a programmer or knowledgeable user, many existing legacy applications can be targeted — even when the codebase has been lost, or is otherwise unavailable.

Execution safety must always be respected. Speculatively parallelized applications must be indistinguishable from their original brethren, save for their increased execution performance. When in doubt, the framework should err on the side of safety and preserve the original sequential program semantics.

Most of the techniques I propose can work separately, and can be adopted by other parallelization systems and schemes. Nevertheless, they were developed to fit together coherently, and the proposed system draws its highlights from their combined usage.

1.2 Contributions

With this dissertation, I provide the following contributions:

- I propose a new model for Software Transactional Memory that is specialized for speculative parallelization. This specialization allows the proposed model to better match the semantics expected from the speculative execution of originally-sequential code, allowing optimizations such as distinguishing between non-speculative and speculative threads, the integration of return value prediction, and support for captured memory.

- I provide a high-performance implementation of the proposed Software Transactional Memory model that, when combined with a modern optimizing VM, allows very low-overhead instrumentation of memory accesses, with minimal memory footprint.

- I develop an algorithm for automatically transforming method invocations into speculation spawn points that return futures. This algorithm is able to analyze the multiple possible control flows in a method, and also includes analysis passes to avoid overeager spawn point insertion. Together with the proposed STM model, this technique allows for speculation of method continuations without relying on value prediction nor needing explicit refactoring of program code.

- I describe the design and development of software components that have already enabled other research work and been deployed onto production systems:

  - My extension of previous work on VM-based continuations used to create a version of the OpenJDK HotSpot JVM with support for first-class continuations has already enabled further research on speculation in the context of distributed Software Transactional Memories [Fernandes, 2011, Peluso et al., 2012].

- The bytecode-based reflective invocation mechanism that I developed has also been extracted into a library that is used by the JVSTM[7] and the Fénix Framework,[8] both of which power the FenixEdu project.[9] FenixEdu is used daily by thousands of students across multiple Universities and Schools.

• I identify and present solutions to a number of issues related to the transformation of code to work with transactional semantics, including dealing with non-transactional operations, which are not undoable, and dangerous operations, which if allowed to execute may cause deviations from the program's original sequential semantics.

• I propose novel techniques that enable a speculative parallelization system to make better use of the resources available on a multicore machine by enabling safe buffering of speculative tasks and safe speculative task checkpointing and migration.

1.3 Outline of the Dissertation

This dissertation comprises 9 chapters:

• *Introduction.* In Chapter 1, I introduce the main subject of this dissertation. I establish the ongoing need for sequential execution performance that is caused by many applications still being single-core or lightly-threaded, as evidenced by the continued investment by chip manufacturers on low-core high-frequency multicore designs. I then present automatic parallelization of applications as a possible solution and identify object-oriented applications as an application domain that still presents a number of challenges, leading up to the presentation of my thesis statement: the creation of a practical speculative parallelization framework for object-oriented applications that execute atop a managed runtime. I then finish up with a list of the contributions provided by this work.

• *Motivation, Problem Statement, and Approach.* In Chapter 2, I propose that automatic parallelization be used as a bridge that would allow multicores to offer increased performance to sequential applications, and identify a number of characteristics that set object-oriented applications apart from many other application domains. I then introduce Java as a practical target for the creation of a speculation framework for object-oriented applications, and describe a number of challenges for this endeavor that I address with this dissertation. I finish with a high-level overview of the proposed JaSPEx-MLS speculative parallelization framework.

• *Related Work.* In Chapter 3, I present a review of notable research work related to parallelization of applications and speculative execution of code, exploring both software and hardware-based solutions. I go into detail on speculative parallelization, with a special focus on Thread-Level and Method-Level Speculation techniques. I present an exploration of current Java-based solutions for parallel programming and transactional execution,

[7] *JVSTM - Java Versioned STM*, http://inesc-id-esw.github.io/jvstm/
[8] *Fénix Framework*, http://fenix-framework.github.io/
[9] *FenixEdu*, http://fenixedu.org/

followed by works that focus on the enablement of parallelism in Java applications. I finish with a discussion contrasting the various works and their positions in the solution spectrum.

- *Building Blocks.* In Chapter 4, I introduce the concept of Transactional Memory, and examine both hardware and software-based implementations of Transactional Memory. Afterwards, I provide an overview of the Java Platform, highlighting important characteristics that I shall need to consider as part of my implementation. Finally, I end with a brief description of a special low-level API that will enable low-overhead speculative execution of code by the JaSPEx-MLS framework.

- *First-class Continuations on the JVM.* In Chapter 5, I describe how first-class continuations can be used as a tool on top of which Method-Level Speculation can be built. I then examine existing implementations of continuations in Java, and describe the enhancement of an existing limited VM-based continuation implementation into a more generic and efficient mechanism to be used as an enabler for the JaSPEx-MLS framework. I also introduce an alternative bytecode-based reflective invocation mechanism as a solution to a limitation of the VM-based continuation implementation.

- *Bytecode Preparation.* In Chapter 6, I describe the bytecode modifications that the JaSPEx-MLS framework needs to perform to a sequential application. The first set of modifications concern the *transactification* of code — modifying it to behave with transactional semantics. The second set of modifications prepare an already transactional but still sequential application for speculative parallelization by choosing and inserting speculative task spawn points. After being modified, application bytecode is loaded into the Java VM and is ready to be speculatively executed in parallel by the JaSPEx-MLS framework.

- *Application Execution.* In Chapter 7, I describe the run-time orchestration performed by the JaSPEx-MLS framework: how and when speculative tasks are spawned, queued for execution, executed, and their results propagated to the global program state. I also discuss novel techniques for making better use of the available machine resources, go into detail on the design choices and trade-offs of JaSPEx-MLS's custom STM model, and describe how the profiling modes included in the framework can be used to better guide the process of speculative parallelization.

- *Experimental Results.* In Chapter 8, I present the results of experimental measurements of executing applications using the JaSPEx-MLS framework. I start by measuring the overheads introduced by the bytecode preparation performed by the framework, followed by the impact of using a production optimizing VM as the host platform for the framework and a comparison of benchmark execution performance across multiple Java VMs. I then present benchmark results of the proposed task buffering and freezing techniques and of speculatively parallelizing a number of applications.

- *Conclusions.* Finally, in Chapter 9, I summarize the main contributions of this work and discuss both their future application and further research directions, followed by a list of publications produced as a result of this research work.

Chapter 2

Motivation, Problem Statement, and Approach

In an era of GPUs, cloud computing and multicore cellphones, it is easy to overlook sequential code execution performance. While many modern applications are able to take advantage of these advances — in both availability and accessibility to the general public — there are still a large number of legacy applications that are sequential or mostly sequential.

In the previous chapter, I discussed how — almost a decade after the introduction of the first commonly-available multicore chips — most chip designers are still working towards faster sequential processors, many times at the expense of not having more processing cores in a multicore. This happens because there are many common workloads that do not benefit from an increasing number of cores, being only single or lightly threaded.

As a further example, manufacturers such as Intel and AMD are even introducing very advanced power management systems into their own chips. Reportedly, by 2008 Intel was dedicating as many transistors to power management as those needed by a 486 chip.[1] One of the features provided by those systems is the dynamic and independent adjustment of core frequency to adapt to the current workload: if a core is lightly used, its frequency is reduced, and if it is unused, it is shut off. Lowering the operating frequency of individual cores (or even better, shutting them off) reduces power draw, and by consequence heat dissipation.

Modern chips are designed to target a certain power and thermal budget. Clever power management frees up a part of that budget, which can then be applied towards boosting heavily-used cores, allowing their frequency to go above the stock configured speeds — what is commonly known as *overclocking*. This performance boost technique is announced by manufacturers with names such as Turbo Boost (Intel) and Turbo Core (AMD). In practice, this means that, for instance, while Intel's Core i7 920XM is normally clocked at 2.0GHz, if cooling is adequate, it can run at 2.26GHz; if only two cores are being used, it can go up to 3.06GHz; and when only a single core is loaded, 3.2GHz can be reached.

The quest for greater sequential speed and the increasing expenditure of transistor budget towards complex power management mechanisms has lead to the current landscape of low-core-count high-frequency multicore chips, greatly handicapping modern parallel applications.

[1] *Nehalem - Everything You Need to Know about Intel's New Architecture*, http://anandtech.com/show/2594/12 (2008)

Even worse, manufacturers that invest in manycore designs for common (non-scientific) usage tend to be quickly penalized due to their poor performance on lightly-threaded benchmarks.

This mismatch between sequential and parallel applications is, I believe, the biggest issue holding back the coming of the manycore era that was promised to us by chip designers at the dawn of multicore parallelism. Because the majority of general-purpose multiprocessors in the market are composed of symmetric cores, chip designers must make a choice to favor sequential or parallel execution, and they seem to overwhelmingly bias toward sequential execution, making for a very limited availability of general-purpose manycore processors.

But, can we do better?

Is there a way to have the best of both worlds?

Is it possible to have a manycore chip that provides lots of parallel processing power for modern applications, while still providing adequate sequential performance?

I propose to work on a first step towards that goal by creating a software-based automatic parallelization system for object-oriented applications with similar trade-offs to those present in hardware-based CPU boost implementations. Such a system would take advantage of unused CPU cores to execute parts of a sequential application in parallel, leading to an increase in application execution performance at the expense of increased power usage.

Like CPU boost, the proposed system should not be seen as a replacement for a properly-written parallel application, but as an enabler for existing sequential applications to extract performance gains from multicore processors without depending on a rewrite.

2.1 Towards the Parallelization of Object-Oriented Applications

As I introduced in the previous chapter, there are a number of automatic parallelization systems (e.g. [Wilson et al., 1994, Blume et al. [1996]]) that successfully tackle mostly-regular scientific applications. Unlike those approaches, I aim to work on a class of applications that I refer to as *object-oriented* applications, or more generally, applications which are both irregular and dynamic:

- An *irregular* application is one that employs pointer-based data structures with data access patterns that are normally not known until run-time.
- A *dynamic* application is one that employs control abstractions such as inheritance, polymorphism, and encapsulation, which also make static analysis hard or, in some cases, even impossible.

These kinds of applications are very hard to analyze statically. Many of the features of object-oriented programming that enable code reuse and help a developer to structure his program work against any framework or compiler that attempts to understand what code will execute, what are its data access patterns, and generally how the program will behave at run-time.

Thus, an approach towards the parallelization of these applications will also need to be dynamic and employ *speculation* in combination with a runtime framework that orchestrates correct execution.

The defining characteristic of speculative parallelization techniques — also known as Thread-Level Speculation (TLS) — is that parallelization decisions are only tentative, as the parallelization framework does not usually have enough information to hard code a decision. Instead, a combination of static analysis, profiling and run-time adjustments are employed, so that parallelization decisions can be guided by the currently-executing code's profile and behavior.

While neither automatic nor manually-applied speculative approaches to parallelization are new (e.g. [Oancea et al., 2009, Mehrara et al., 2009, Spear et al., 2010, Raman et al., 2010, Chen and Olukotun, 2003, Yoo and Lee, 2008]), similarly to non-speculative parallelization approaches, there is no "one size fits all" technique for speculative parallelization: Each system is tuned for a certain kind of application, and presents a set of different trade-offs.

Object-oriented applications, due to the reasons presented above, tend to present different challenges from scientific or procedure-based applications. In more detail, I identify the following characteristics as challenges to many existing automatic parallelization approaches:

- *Complexity of operations used in loops.* This impacts many loop-based parallelization approaches, as even an apparently small loop can invoke a number of methods on multiple objects. Because the code executed for each individual object may change due to polymorphism, each iteration may be very different from the one preceding it, and static inlining is usually not applicable.

- *Most of the application needs to be transactional.* Because multiple code paths may be followed and data is usually spread across a number of objects, the resulting in-memory speculative transactions need to be dynamically-sized — able to handle a variable number of reads and writes spread across the program heap.

  In addition, it is hard to leverage existing hardware support for speculation and transactions, as it is very limited in both availability and capacity; most cannot support the transaction sizes that result from many of these applications.

- *Non-transactional operations are common.* It is common for these applications to depend on file-system, networking and user interaction APIs, but modern operating systems offer little in the way of transactional APIs. As a result, most interactions with systems outside the application cannot easily be undone. This is in stark contrast with scientific-like applications, where normally there are easily-delimited input and output processing phases.

  A common approach to dealing with these kinds of non-transactional operations is to never perform parallelization if there is a chance that a non-transactional operation may be triggered, but — again due to limitations in static analysis — such an approach is too pessimistic for object-oriented applications [Anjo, 2009].

- *Reliance on managed runtimes.* Most modern object-oriented applications — with C++ being a very large exception — execute atop a managed runtime environment. To obtain adequate performance, these managed runtimes normally include a number of optimizing compilers; for security and integrity reasons, they also perform a number of validations both at load and at run-time; and, in addition, they normally include features such as automatic memory management via garbage collection, and a large library of commonly used code.

To work effectively on a managed environment, a parallelization framework must take into account the limitations and restrictions of the managed platform, and either cooperate with or provide effective alternatives to runtime-provided services; otherwise many of the advantages drawn from managed runtimes can be lost.

- *Usage of thread-safe and parallel third-party libraries.* Many object-oriented applications rely on a number of third-party libraries. Even if the applications themselves are strictly sequential, many libraries include internal locking and may even include their own thread pools for offloading work.

  As part of its run-time orchestration, a parallelization framework must discover and take measures to protect against interference from unexpected usage of concurrency primitives. This enables applications that would otherwise be pessimistically rejected due to some unused part of their code not being strictly sequential to be successfully parallelized.

2.2 Case Study: The Java Platform

As a case study for implementing the ideas in this dissertation, I chose the Java/JVM Platform: The Java Platform is an excellent representative of modern object-oriented managed runtimes; It is used daily in business, personal and mobile applications; It is open and very well documented.

Whereas, in my aim to create a practical framework, I have to take into account many of the specificities of the Java Platform, most of the techniques developed are not specific to Java, and I expect that they may straightforwardly be applied to other similar platforms, such as Microsoft's .NET Framework.

As a managed runtime, the Java Platform abstracts away many hardware-specific issues by representing applications in a higher-level format, delaying the translation into a low-level optimized version until run-time. It also imposes restrictions on applications allowing them to be partly verified at load-time and the cheap enforcement of safety features such as pointer and bounds verifications at run-time.

2.2.1 Java or JVM Applications?

Originally designed with the intent of hosting applications written in the Java programming language, the Java Platform and its implementation in the form of a Java Virtual Machine (JVM) has since evolved into a common platform that is targeted by multiple programming languages such as Scala, Clojure, Ruby (JRuby) and Python (Jython).

This works because the JVM does not execute Java programs directly — it executes Java bytecode. Any language that can be compiled into a valid Java *bytecode* stream can be executed on the JVM.

In fact, there are a number of features supported or allowed by the JVM that are not supported or are disallowed to Java code; some of these were explicitly added to facilitate the execution of and solve design issues identified by popular non-Java JVM-hosted languages.

Because all JVM languages — including Java — eventually all share a common representation — Java *bytecode* — a framework targeting Java bytecode is able to work with any of these languages.

After its generation, Java bytecode is saved into Java class files, each one representing a single Java class — depending on the source language, it is not uncommon for a single source file to be compiled into several Java class files. A class file contains constant resources used by the class, optional debug information, optional meta-data in the form of annotations, a description of the class's fields and methods, and method bodies in the form of Java bytecode.

2.3 Problem Statement

With my work I aim to create an automatic speculative parallelization framework that is able to achieve practical speedup when compared to production VM runtimes.

The main goals for this implementation are:

- *Completeness.* The implementation will strive to work with all Java and JVM applications and their associated code and API use patterns, not just with a hand-picked simpler subset. Note that completeness does not imply that every application will achieve speedups.
- *Software-based implementation.* The implementation must work on commonly-available commercial multicore hardware, and not depend on experimental or custom hardware extensions.
- *As-if-sequential semantics.* The parallelized version of an application must only be distinguishable from its sequential version by its execution speed; its observable semantics must be equivalent to the sequential semantics originally expected by the application and its developer.
- *Does not depend on programmer input.* The process of parallelizing an application should not depend on a programmer or expert user's input — but it should also be able to take advantage of that valuable information whenever it is available.
- *Does not need access to the application's source code.* As working from the source code would greatly limit and complicate the applicability of an automatic approach to parallelization, the parallelization process should work from compiled Java bytecode and must not depend on having source-code access.
- *Low overhead, during or outside speculative execution.* The implementation will strive to execute an application with as little overhead as possible with the aim of, in the worst case, providing the same execution performance as the original sequential implementation.

In addition, the field of automatic parallelization poses many issues, and I explicitly consider the following as being out-of-scope of my work:

- *Energy efficiency.* While the topic of energy efficiency is very important for computing, the goal of my work is to provide a software-based framework that provides a trade-off similar to the hardware boosting modes from commercial chip designers such as Intel and AMD: In these chips, unused processor capacity and thermal headroom are taken advantage of to increase sequential processing speeds, e.g. a processor may run faster and consume more energy than it would normally do if cooling is adequate and there are unused CPU cores.

Many times in computing, efficiency and raw performance are at odds with each other, and it is the user that should be the final arbiter for this choice.

- *Replace the need for a properly-designed parallel application.* The proposed implementation should not be considered a replacement for a properly-designed rewrite or adaptation of an application; it is instead a bridge that provides better performance for sequential applications by taking advantage of unused resources on a multicore, enabling chip designers to shift from the "larger and more complex cores" to the "more cores" mindset.

  In general, it is hard for an automatically-parallelized application that still needs to provide as-if-sequential semantics to compete with parallel applications that for instance allow non-deterministic (but still correct) output depending on execution ordering.

2.4 Approach: Creation of a Speculative Parallelization Framework

To tackle the challenges identified in previous sections, I designed and implemented the JaSPEx-MLS speculative parallelization framework. Based on my earlier MSc work on the JaSPEx framework [Anjo and Cachopo, 2009, Anjo, 2009], JaSPEx-MLS is a fully software-based speculative parallelization framework for object-oriented applications that works atop a high-performance Java runtime (Section 2.4.3), and employs both Method-Level Speculation (Section 3.3) and Software Transactional Memory (Section 4.1.1).

JaSPEx-MLS is implemented using a combination of both Java and dynamically generated Java bytecode. Normal, unmodified Java or JVM applications are executed by first intercepting the JVM class load procedure, replacing it with a custom class loader that analyzes and prepares application bytecode for speculative execution. I describe in detail the bytecode preparation step in Chapter 6. Afterwards, the modified application starts executing, and a number of injected hooks allows the JaSPEx-MLS framework to orchestrate, at run-time, the safe speculative execution of parts of the application in parallel. This run-time coordination is discussed in Chapter 7.

2.4.1 Solution Overview

Practical automatic parallelization of object-oriented applications poses very different challenges from other application domains. Both control and data abstraction greatly limit static program analysis, and the reliance on managed code execution environments such as the Java Virtual Machine both facilitates program transformation and hinders direct control over its run-time execution.

In this dissertation, I address several of the issues posed by attempting speculative parallelization of object-oriented applications:

- *Task identification and insertion.* To parallelize a sequential application, the proposed framework must first break it up into individual units of execution — tasks — and perform a number of modifications so that the framework is able to, in cooperation with the Java VM, control task creation and execution, as we shall see in Section 6.2.

  Code dependences hinder parallelism by forcing concurrent tasks to synchronize and to wait for values from other tasks. My framework attempts to uncover latent parallelism and boost the effectiveness of automatic parallelization by modifying parts of an application's execution, but any such changes must never break the expected sequential semantics.

- *Speculative code execution.* The defining characteristic of speculative execution is that its results are tentative. Because they are tentative, it is important that they be kept isolated from definitive (non-tentative) parts of the application. There must also exist a mechanism that verifies if/when a set of tentative results can be made definitive, and that performs that change. Whenever the mechanism deems the tentative results unsuitable — usually as a consequence of a conflict with other concurrently-executing parts of the application — it must also be able to discard the tentative results without triggering any side effects.

  Because the JVM provides no such mechanism for speculation, my framework must first intercept all accesses to the application's heap state performed by code that we want to execute with transactional semantics, as described in Section 6.1.2. Then, at run-time, my framework provides proper transactionally-safe replacements for these operations, as we shall see in Section 7.4.

- *Integration with non-transactional code.* Not all code can be made transactional, and it is common for object-oriented applications to make use of such code. To solve this, I propose to identify which parts of an application are non-transactional, and afterwards to provide a mechanism that allows the safe execution of such code, as described in Section 6.1.3.

- *Task execution.* While task identification is performed at load-time when an application is being prepared for speculative parallelization, task execution concerns itself with — at run-time — both managing the creation and execution of speculative tasks, taking into account the dynamic behavior of the application, and mapping the selected tasks onto the available machine resources, so that the available parallel resources of the host machine are fully used, as we shall see in Section 7.1.

  A very important part of this process is the correct communication of results between tasks, so that my framework can both ensure progress and that the observed execution respects the original sequential application's execution semantics.

The techniques introduced and used throughout this dissertation always take into account the goal of a practical implementation: They are designed to perform with minimal overheads so as to have a minimal impact on application code execution speed, and to maximize the obtained speedup. In addition, as most non-trivial Java applications exercise a great number of JVM mechanisms, and as I aim to support those applications, the described implementations need to be complete, with as few restrictions as possible. Finally, platform services such as garbage collection and JIT compiling should still work and remain largely unaffected by my work.

2.4.2 JaSPEx-MLS System Overview

Figure 2.1 shows the JaSPEx-MLS system stack, which is divided into two parts: bytecode preparation and application execution.

Before a Java class is loaded by the JaSPEx-MLS class loader, its bytecode goes through several changes in preparation for speculative parallelization (Chapter 6): This includes transactifying the code — changing it so that all memory accesses are done under the control of a Software Transactional Memory, making them behave transactionally; the addition of

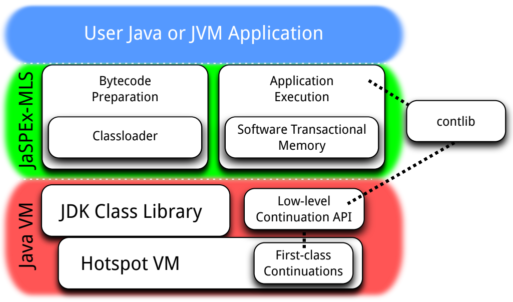

User Java or JVM Application
JaSPEx-MLS
Bytecode Preparation
Classloader
Application Execution
Software Transactional Memory
contlib
Java VM
JDK Class Library
Low-level Continuation API
Hotspot VM
First-class Continuations

Figure 2.1: The JaSPEx-MLS system stack. As requested, application bytecode is run through the bytecode preparation component, and loaded into the VM via the class loader. The JaSPEx-MLS runtime then takes over and coordinates the parallel execution of applications, supported by a custom Software Transactional Memory. The whole system is hosted by a special Java Virtual Machine that has been extended with support for first-class continuations.

hooks that allow non-transactional operations to execute safely and that disallow dangerous operations from executing; the insertion of spawn points that will be used at run-time to create speculative tasks to be executed concurrently; and morphing the code to employ futures as placeholders for the return values from spawned tasks.

After the modified bytecode is loaded into the VM, it starts executing under the guidance of the JaSPEx-MLS runtime (Chapter 7). The runtime is then responsible for controlling the creation of speculative tasks, establishing the commit order for the underlying transactional system, deciding when to validate and to commit speculative tasks, correctly handling aborting and retrying, and controlling the execution of non-transactional operations.

To implement thread state snapshotting and transfer, the JaSPEx-MLS framework depends on having a Java Virtual Machine with support for first-class continuations. Because this feature is not a part of the Java Platform standard, I rely on my own implementation, based on [Yamauchi, 2010], on top of the OpenJDK HotSpot VM. To avoid a direct dependence on a custom JVM implementation, I developed the `contlib` library, which wraps the low-level JVM continuations API, with the goal of allowing future portability to other VMs. Apart from the code contained in the `contlib` library, the framework itself is portable and capable of being executed atop any modern Java VM.

The OpenJDK project is the result of the open-sourcing of Oracle's Java technology, including the HotSpot VM. By working on top of OpenJDK, JaSPEx-MLS benefits from all the features of a modern production JVM: just-in-time compilation, adaptive optimization, state-of-the-art garbage collection algorithms, support for all of Java's features, and optimized concurrency primitives. Adopting HotSpot for my work created a number of challenges, that I describe in the next section, but its performance is a key component in keeping with this dissertation's goal of achieving practical speedup.

I believe that the combination of software-only speculation on top of a modern production JVM sets the JaSPEx-MLS framework apart from previous work: This approach works on commonly available hardware, on top of the same codebase regularly used to run the original sequential versions of the applications that I am targeting, and it is able to extract practical speedup on several applications given this real-world environment.

2.4.3 Why Work Atop the HotSpot JVM?

Most desktop and server Java real-world applications are executed using the HotSpot Virtual Machine; HotSpot is the virtual machine at the heart of Oracle's (formerly Sun Microsystems') Java Runtime Environment (JRE). It is a very high-performance VM, that was created as an answer to early Java versions' performance issues, allowing Java-based applications' speed to compete with native code execution, while still benefiting from Java's portability, safety and programmer friendliness. Included in HotSpot are features such as dynamic optimization via tiered just-in-time compiling, state-of-the-art parallel garbage collection algorithms, and highly-optimized concurrency primitives such as fast locking [Click, 2010].

In 2006, Sun Microsystems open-sourced HotSpot under the GPL license, sparking the creation of the OpenJDK Project.[2] Thus, modern releases of Java — done by either Oracle or the OpenJDK project — are based on HotSpot. HotSpot is also the reference implementation for Java, the standard for all Java implementations. For a VM to be considered as Java-compatible, it must pass a number of tests that check that its behavior is the same as the reference implementation.

As a cross-platform cross-architecture state-of-the-art virtual machine, HotSpot is incredibly complex. Its project website warns newcomers: "*The HotSpot code base has been worked on by dozens of people, over the course of 10 years, so far. (That's good and bad.) It's big. There are nearly 1500 C/C++ header and source files, comprising almost 250,000 lines of code. In addition to the expected class loader, bytecode interpreter, and supporting runtime routines, you get two runtime compilers from bytecode to native instructions, 3 (or so) garbage collectors, and a set of high-performance runtime libraries for synchronization, etc.*"[3]

Modifying HotSpot for speculation (or generally, for most non-trivial feature additions) is very challenging. [Schätti, 2008] documents an attempt to do so, that even by reducing the scope of the implementation to just the interpreter was not successful. Because of this, Java-based speculative parallelization systems end up being implemented as modifications of simpler research VMs (e.g. [Pickett, 2012, Saad et al., 2012]), which are more amenable to the multiple changes needed for speculative execution. But, exactly because they are simpler, and have a greater focus on easy experimentation and modifiability, their performance is far from being comparable to HotSpot or other complex production VMs. This leads systems based on them to not be practical (as defined in Section 1.1): any performance gains from parallelization still have to offset the difference between the simpler VM and HotSpot before being of use to end-users.

As an alternative, I propose in [Anjo and Cachopo, 2011b] to work atop the VM as much as possible. This turned out to be mostly possible, except for the JaSPEx-MLS framework's reliance on first-class continuations, support for which needed to be implemented inside the VM so as to not incur enormous performance penalties. This implementation was based on earlier work by [Yamauchi, 2010] and is described in Chapter 5.

As a result of this strategy, as we shall see in Chapter 8, JaSPEx-MLS is able to take advantage of HotSpot's features to minimize the impact (overhead) of a number of changes done to client applications, and reach practical speedup for a number of benchmarks.

[2]http://openjdk.java.net/
[3]http://openjdk.java.net/groups/hotspot/

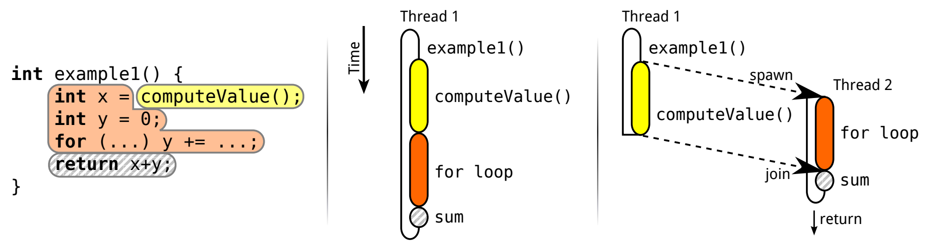


Figure 2.2: Execution of the `example1()` method (left) when run normally (center) and parallelized with MLS (right). Note that `computeValue()` is executed in the normal program order (at the start of `example1()`), whereas the `for` loop is executed speculatively.

2.4.4 Method-Level Speculation

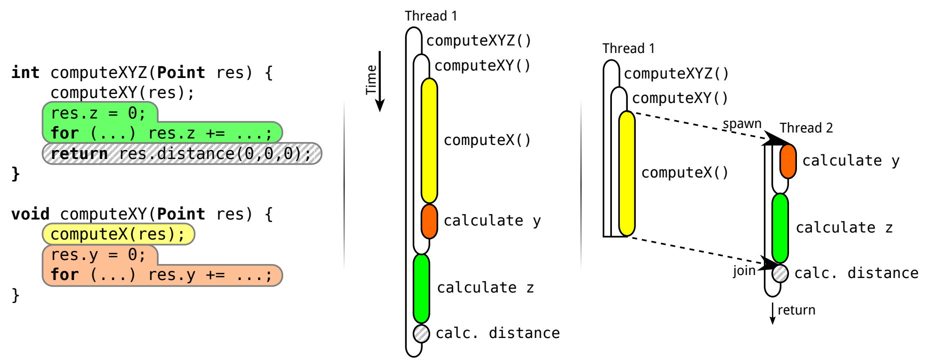
The first step in parallelizing an application is breaking it up into a number of tasks. How task selection is made depends on the type of application, on the expected independence of each part of its code, on the overheads and limits for task execution, and also on restrictions imposed by the execution platform itself.

Method-Level Speculation (MLS) is a speculative parallelization strategy first discussed in the context of Java by Chen and Olukotun [Chen and Olukotun, 1998], and shown to be a promising source for parallelism by several researchers (e.g. [Oplinger et al., 1999, Warg and Stenström, 2001, Whaley and Kozyrakis, 2005, Pickett, 2012]). This technique works by using method calls as speculative task spawn points — speculatively executing the code following the return of a method call in parallel with the method call itself.

MLS shares a number of similarities with the Fork/Join model, which was introduced into the Java 7 platform via the Java Fork/Join framework [Lea, 2000]. The MLS spawn operation works similarly to the fork operation, but with an important distinction. Whereas in the MLS model the new task spawned (forked) starts executing the code following the spawn (fork) point and the code in the method call (fork/join task) is executed as part of the previously-existing task, in the Fork/Join model the reverse happens: A new task is created to execute the code being forked and the code following the fork is executed as part of the previously-existing task. This distinction is in the models themselves, regardless of the run-time strategies chosen for task execution. For both Fork/Join and MLS, the join operation is similar, and serves to synchronize a pair of tasks where one needs to obtain the result of another's computation.

The main difference between the two models is that Fork/Join does not include any support for speculation, and is targeted at parallel algorithms where all tasks need to be executed and tasks are assumed to not interfere, making task execution order irrelevant. In an MLS system, tasks may be created based on invalid assumptions, so their entire computation will have to be undone. MLS tasks must also follow the strict ordering semantics imposed by the original sequential code — so that the outputs from MLS-parallelized code match the original application's outputs.

An example of Method-Level Speculation is shown in Figure 2.2. When the `computeValue()` method call is reached, Thread 1 (hosting the current task) begins executing it, while at the same time triggering the spawn of the speculative execution in a new task (by Thread 2) of the code following the return of that method. Thus, both the `computeValue()` method and the `for` loop execute concurrently.



```
int computeXYZ(Point res) {
    computeXY(res);
    res.z = 0;
    for (...) res.z += ...;
    return res.distance(0,0,0);
}

void computeXY(Point res) {
    computeX(res);
    res.y = 0;
    for (...) res.y += ...;
}
```

Figure 2.3: Execution of the `computeXYZ()` method (left) when run normally (center) and parallelized with MLS (right). In this example, a new speculative task is spawned to execute the code following the call to `computeX()` and speculative execution proceeds past the return of `computeXY()`, continuing into `computeXYZ()`.

To distinguish clearly between the two tasks, I refer to the existing task that goes on to run the method (`computeValue()` in the example) as the *parent* task, whereas the newly created task that runs the continuation of the method (the `for` loop in the example) is referred to as the *child* task. This parent/child relation implicitly imposes a global order on all tasks on the system that mirrors the code execution order of the original sequential application.

In the presented example, both the parent task and the speculative child task have to join to produce the result of the method. If, instead, the value of the variable `x` was never used, it would be possible to speculate past the return of `example1()`, and continue executing `example1()`'s caller method. Figure 2.3 shows one such example: Speculation starts inside `computeXY()` triggered by the call to `computeX()`, and continues past the return of that method and into `computeXYZ()`. Notice that this is made possible because the entire execution stack is moved from Thread 1 to Thread 2.

Alternatively, whenever a value is needed to proceed with the execution — as is the case with `x` from the previous example in Figure 2.2 — we can employ return value prediction to guess a probable value and continue speculation using this assumption, as discussed in Section 7.4.5.

2.4.5 Threads and Tasks

JaSPEx-MLS splits user applications into tasks. Any application code that runs, always runs as part of a given task. As we shall see in Section 7.4, tasks may be executing speculatively or non-speculatively (in program-order).

Tasks are assigned to be executed by threads. Internally, JaSPEx-MLS uses a thread pool, allowing threads to be reused to run multiple tasks. Threads pick up tasks for execution, and only after finishing them return to the pool. While describing the system, I refer to threads interchangeably with tasks, whenever it is clear from the context that I am referring to the thread that is hosting a given task.

2.5 Summary

In this chapter, I expand on the need for automatic parallelization systems as a bridge between current high-frequency low-core-count multicores and future manycores. I propose the creation of one such system, targeted at object-oriented applications.

I then introduce a number of characteristics that set object-oriented applications apart from other application domains (such as scientific applications) where automatic parallelization is successfully used.

I describe my choice of Java as a practical target for my techniques, and a number of challenges that speculative parallelization on the Java Platform poses.

I finish with an overview of the JaSPEx-MLS speculative parallelization framework, which will be described in detail in the following chapters.

Chapter 3

Related Work

In previous chapters, I argued for the importance of automatic parallelization approaches as a bridge that would allow chip designers to transition from multicores to manycores, and proposed to build a software-based automatic parallelization system aimed at object-oriented applications.

The idea of parallel-friendly and automatically parallelizing compilers and frameworks is not new — In this chapter I overview such research works, with the goal of both exploring alternatives similar to my proposal and also of identifying and understanding other contrasting approaches — with special focus on Java-based solutions.

I start by looking into notable parallelizing compilers in Section 3.1. Thread-Level Speculation (TLS) systems, examined in Section 3.2, attempt to extract more concurrency from applications by combining automatic parallelization with speculative execution. One TLS parallelization strategy in particular — Method-Level Speculation — is explored in detail in Section 3.3. Section 3.4 looks into hardware-based transactional systems for Java, and Section 3.5 discusses concurrent programming on the Java Platform. After that, in Section 3.6, I describe research work on speculative execution issues, and on uncovering more parallelism.

As I introduce the related work, I briefly comment on each proposed system, and in Section 3.7 I conclude with an expanded discussion where I relate the presented works.

3.1 Parallelizing Compilers

A number of earlier automatic parallelization works focused on Fortran, C, or C-style non–object-oriented C++ loops that can be analyzed statically and with few or no cross-iteration dependences. Many scientific-style applications are amenable for this kind of parallelization, which, due to its highly-static approach, usually presents very low execution overheads.

SUIF [Wilson et al., 1994] is an infrastructure for experimental research on parallelizing and optimizing compiler technology. It includes C and Fortran front-ends, and an optimizing MIPS backend. The SUIF Kernel defines an intermediate representation of programs, an API to access and manipulate them, and structures the protocol between compiler passes. Parallelization is aimed at loops, which are modified in multiple passes: First, SUIF applies scalar optimizations to help expose parallelism, such as constant propagation, forward propagation, induction variable detection, constant folding, and scalar privatization analysis. Afterwards, it performs an array dependence analysis, using its results to restructure the code for better locality and parallelism. Finally, the parallel code generator inserts the calls to the parallel run-time library. The included loop analyzer can recognize common reduction patterns such as sum, product, and minimum/maximum, even when they span across multiple nested loops.

A similar approach is used by the Polaris compiler. Polaris [Blume et al., 1996] is a prototype parallelizing compiler for Fortran that aims to overcome the limitations of earlier parallelizing compilers by combining multiple techniques. Interprocedural analysis is done by inline expansion, where the compiler starts with a top-level program unit and repeatedly expands subroutine and function calls. Induction variable substitution and reduction recognition try to break data dependences between different iterations of loops, so that they may execute in parallel. Symbolic dependence analysis determines which statements and loops can be safely executed in parallel. Scalar and array privatization passes identify scalars and arrays that are used as temporary work space by a loop iteration, and allocate local copies of them.

The Intel C++ and Fortran compilers include an automatic parallelization mode targeted at applications “where most of the computation is carried out in simple loops” [Intel Corporation, 2010]. When this mode is enabled, the compiler tries to add OpenMP parallel execution pragmas to loops that can be safely executed in parallel. To determine if a loop can be safely modified to work concurrently, the compiler relies on dataflow analysis. Unfortunately, this mode is very limited, and needs that many details about the loop be determined statically: The number of loop iterations must be known in advance, there can be no jumps out of the loop, and iterations must be independent. Other issues impeding the automatic parallelization include pointer aliasing and calls to external functions. As a fallback, Intel’s documentation suggests that the programmer can instead manually parallelize his application using OpenMP.

While a mostly-static approach is very successful for some classes of applications, modern object-oriented applications are usually very dynamic and as such these approaches are not effective in extracting parallelism from them.

3.2 Thread-Level Speculation

Thread-Level Speculation (TLS) systems extend classic parallelizing compilers by speculatively choosing parts of an application to run in parallel even when they cannot statically prove that the result will still be correct. At run-time, changes performed by code running speculatively are buffered or protected in some way, so that they can be undone in case the speculative execution fails because of capacity or dependence violations; the mechanisms for these are similar to those employed by Transactional Memory implementations, which I shall later describe in Section 4.1.

Multiple strategies on how to identify parts of a program to be run in parallel have been studied: In Section 3.2.1 and Section 3.2.2 I explore works mainly employing loop-based parallelism, deferring discussion on method-based parallelism to Section 3.3.

3.2.1 Hardware-based TLS Approaches

In this section, I describe TLS approaches that rely on changes to multiprocessor hardware, such as the addition of transactional buffers, or the introduction of changes to cache coherency protocols. While commercial TLS hardware is still mostly non-existent, these works allow the comparison and experimentation with different interfaces that could be used in future implementations of such hardware.

[Steffan et al., 2000] describe a hardware implementation of TLS that is built directly into the cache coherency protocols of participating processors. This allows speculation to scale beyond a single chip-multiprocessor, onto a machine with multiple such chips. The authors test their proposed architecture by manually adding TLS primitives into loops that are are not able to be proven parallel by the SUIF compiler [Wilson et al., 1994], and by simulating execution with varying node and architecture configurations.

Jrpm [Chen and Olukotun, 2003], the Java runtime parallelizing machine is a Java VM that does speculative loop parallelization on a multiprocessor that includes a special coprocessor providing hardware support for profiling and speculative execution. At run-time, applications are profiled to find speculative buffer requirements and inter-thread dependences; once sufficient data is collected, the chosen loops are JIT-recompiled to run in parallel. Due to the usage of a virtual machine, no changes to an application prior to execution are needed.

POSH [Liu et al., 2006] is a Thread-Level Speculation (TLS) infrastructure on top of the GNU Compiler Collection (GCC) that aims to perform loop parallelization and targets a multiprocessor architecture with hardware support for speculative transactions. The POSH framework is composed of a compiler and a profiler. The compiler has three main stages: task selection, spawn hoisting, and task refinement. Task selection uses a variety of heuristics to identify different tasks, which can still have dependences — the hardware ultimately guarantees correct execution; spawn hoisting tries to place the transaction spawn instruction as early as possible; and in the task refinement phase the final set of tasks to be parallelized is selected. The profiler is able to run applications with a training input set, afterwards providing the compiler with a list of tasks that are beneficial for performance, allowing for non-beneficial tasks to be eliminated. The authors argue that even when a speculative execution aborts — when a task is *squashed* — it may still be beneficial for the program execution, as when the task is restarted some of the values needed for its re-execution will already be in the processor's cache, and thus a squashed task also works as a prefetching mechanism; an estimated 26% of their obtained TLS speedup is a result of this effect. Unlike other presented works, POSH focuses on the framework needed to analyze and produce code for the simulated processor with hardware support for TLS, rather than on the processor design itself, showing the importance of program analysis and good task selection even when supported by hardware TLS.

IBM's Blue Gene/Q processor [Ohmacht and IBM Blue Gene/Q Team, 2011, Wang et al., 2012] is a multiprocessor chip designed for supercomputer clusters. It supports state snapshotting, Hardware Transactional Memory, and Thread-Level Speculation, in the form of a multi-versioned L2 cache. The programmer can specify the use of speculation via OpenMP-like pragmas — `#pragma speculative for` — on the application loops. When executing in TLS mode, the processor keeps track and enforces the correct ordering for speculative transactions; this ordering information is also used to allow speculative results from older speculative threads to be automatically forwarded to younger ones. The hardware supports a limited number (128) of transaction IDs that can be divided among multiple groups, allowing the transactional resources to be used concurrently — in different processor cores — for both Transactional Memory and speculative execution. The granularity for tracking memory read/write accesses is dynamic, and current L2 sizes allow the processor to support 20MB of speculative state.

When conflicts are detected, transactions can either be invalidated by the hardware, or they can be handed over to a software handler. Blue Gene/Q is one of the few commercially-available processors with TLS support, and its design fusing support for Hardware Transactional Memory and Thread-Level Speculation allows the creation of applications that take advantage of both models. Nevertheless, while the hardware support is available, currently applications still have to be manually changed to take advantage of its features. In the future, it would be very interesting to explore how an automatic parallelization framework could be combined with the provided hardware TLS to reduce speculation overheads and uncover additional parallelism.

3.2.2 Software-based TLS Approaches

Software-based TLS approaches dismiss the need for introducing extensions to the processor hardware's architecture by instead changing (usually automatically) the application to use additional indirections and to interface with thread spawning and coordination runtimes.

Oancea, Mycroft and Harris [Oancea et al., 2009] present SpLIP, a software speculation system that targets mostly-parallel loops. The authors concentrated on avoiding performance pitfalls present on many other software-based TLS proposals, which limit performance and scalability: SpLIP speculations commit their work in parallel, and updates to memory locations during speculation are applied in-place. The downside of this technique is that because threads perform in-place writes to memory, the penalty for bad speculation decisions becomes much heavier, involving costly rollback operations; this prompts a very careful analysis when choosing loops to parallelize. SpLIP also exploits the memory ordering features present on Intel's IA-32 architecture to avoid memory barriers and atomic operations when reading and writing to speculative locations. When present, non-transactional operations and their dependent instructions are executed serially. Results from a test suite composed of applications from SciMark2, BYTEmark, and JOlden show that the presented system reaches, on average, 77% of the speedup of hand-parallelized versions of the same benchmarks, on a machine capable of concurrently executing 8 hardware threads.

Fastpath [Spear et al., 2010] is also aimed at extracting parallelism from loops using speculation. This system distinguishes between the thread running in program-order (the *lead* thread), and other speculative threads: The lead thread always commits its work, and has minimal overhead, whereas speculative threads suffer from higher overheads and may abort. The authors propose two different Software Transactional Memory-inspired algorithms for conflict detection: value and signature-based. When using the value-based algorithm, each speculative thread keeps the values it reads on a read-set, which needs to be validated before a thread can commit its results. In this mode, the lead thread needs no per-access memory instrumentation. The signature-based conflict detection algorithm employs both a global array of write signatures and thread-local private read signatures — these signatures work similarly to bloom filters. Whenever a speculative thread wants to commit, it needs to intersect its own read-set with the write-sets of any transactions that have committed since it started. In this mode, the lead thread needs to compute and update its signature on every write, but no instrumentation is done for reads. Signature-based conflict detection trades off reduced validation costs on speculative threads for extra work from the lead thread; it can also lead to false conflicts and disallows data forwarding. In both conflict detection algorithms, the lead thread is always allowed to change memory locations in-place; speculative threads keep their own write-sets, which are applied only on commit. The Fastpath system as presented did not yet support automatic parallelization; results from a hand-instrumented benchmark showed that the value-based algorithm presented the best results, reaching a speedup of 2.10x with 8 threads. In contrast with more generic in-memory transactional models, speculative parallelization of sequential applications presents ordering characteristics normally not present on regular applications, creating the opportunity for works like Fastpath to provide more efficient speculation-specific (or speculation-aware) transactional models.

In [Mehrara et al., 2009] the authors propose STMlite, a lighter Software Transactional Memory model targeting loop parallelization; the presented implementation works as part of the LLVM C/C++ compiler. STMlite aims at using a small number (2-8) of speculative threads to extract parallelism from loops, avoiding the need to transactify the whole program or to guarantee strong atomicity. During execution, transactional read and write operations are encoded using hash-based signatures that are then checked by a central bookkeeping and commit thread — the Transaction Commit Manager (TCM). By centralizing most of the work on the single TCM thread, synchronization is simplified because there are no concurrent updates to shared state — only the TCM thread is allowed to write to memory. As an additional simplification, only one parallel loop is allowed to be running at a time: A barrier is issued at the end of each parallel loop. Loops with non-transactional operations, or complex operations such as accessing dynamic data structures are not chosen for parallelization. The presented results show that STMlite has smaller overheads and delivers better performance than the TL2 STM algorithm, allowing the parallelization system to achieve speedups over the original sequential applications on some of the chosen benchmarks. Like Fastpath above, STMlite explores characteristics of the speculative parallelization model to design a better STM model for speculation. STMlite is optimized for commonly-available multicore machines instead of bigger manycore machines, trading off scalability on those larger machines for lower overheads on commonly-sized multicores.

Software multi-threaded transactions [Raman et al., 2010] aim at parallelizing loops by combining speculative execution with parallel loop pipelining. Parallel pipelining splits up and executes parts of a loop body in parallel, with each thread being assigned a slice of the work for each iteration. When adding speculation to such a system, and because a single loop iteration is executed across multiple threads, the need for a multi-threaded transaction that is able to migrate between threads as they each execute a part of the same iteration emerges; most TLS and TM systems do not support this paradigm. The proposed implementation differs considerably from other TLS systems: Separation between executing threads is implemented by each being in a separate operating-system–level process. Shared state between processes is managed with the operating system's shared and virtual memory mechanisms, with copy-on-write semantics for memory pages that are changed, dismissing the need for read and write sets — when a shared memory page is to be modified, a copy of it is created, and mapped to the same location as the old one. Any further reads from the same thread will then be able to see the modifications. When a transaction is to be migrated to a different process, the new process must also copy these new memory pages to its address space, so as to have access to the writes done so far. Communication between processes is implemented with single-producer/single-consumer shared memory queues. To simplify commit operations, only one of the processes is allowed to validate and propagate the changes done to the shared memory state, resulting in a commit scheme similar to the one proposed by STMlite.

In [Wilkinson and Watson, 2008] the authors introduce an object-driven approach to speculative parallelization. Under the proposed model, each object instance in an object-oriented application is extended with a header that records the history of methods called for that instance; a per-class least-recently-used (LRU) list of past object history headers is also kept. When a method is invoked on an object, the object predicts which of his own methods is expected to be invoked next, and queues it for speculative execution. The authors present some preliminary data on the accuracy of predictions obtained by this method, but no benchmark results. While most other speculation models work by speculating on the system state for a given operation — e.g. most of the times it is known that an operation or loop iteration will be executed, and it is the system state that is used as a basis for that execution that is speculative — the proposed model mainly speculates on the possibility that a given operation is going to be executed after some other, but most of the needed state will already be available inside the object.

ParaScript [Mehrara et al., 2011] is a TLS framework aimed at loops in JavaScript applications that is built on top of Mozilla Firefox's TraceMonkey script engine. Because JavaScript is a dynamically typed language, most static analysis techniques are ineffective: The proposed framework dynamically identifies loops to be parallelized at run-time, afterwards generating a parallel version of the application. Speculation is supported by both a reference counting scheme to detect potential conflicts, and a checkpointing mechanism. When a hot loop is first encountered, a cross-iteration dependence analysis is performed. If no dependences are detected, a speculative parallel version of the loop is generated and used from then on. If later a cross-iteration dependence is detected, the parallel execution is aborted, the heap state is rolled back to an earlier snapshot, and the loop is blacklisted, its code being reverted back to the sequential version. Checkpointing works by keeping a deep-copy of any objects in the global and local namespaces that analysis reveals as being potentially modified inside a parallel loop. Reference counting is employed during checkpointing to reveal when multiple references to the same object might be present on an array, which may lead to dependence violations. Results obtained with a subset of the SunSpider benchmarks show an average speedup of 2.55x on an 8-core machine. As JavaScript is dynamic and hard to statically analyze, ParaScript ends up needing to take a pessimistic approach to parallelization, with a single dependence violation on a usually-parallel loop leading to the loop being blacklisted and executed sequentially. Nevertheless, this work shows that TLS is a viable approach even for highly-dynamic codes on top of a JIT-compiling script engine.

TLSTM [Barreto et al., 2012] is a middleware combining both STM and TLS. The starting point for this system is a user application which has been hand-parallelized and uses STM. At run-time, each STM transaction is further broken down into a number of subtasks, that are then executed speculatively in parallel. TLSTM builds upon SwissTM [Dragojević et al., 2009a], by adding mechanisms that allow multiple threads to share the same STM transaction, and to correctly order and validate the multiple speculative tasks that make up a single user transaction. Results from hand-parallelized benchmarks show that this approach is able to extract finer-grained parallelism from already-parallel applications, although it is dependent on the correct selection of decompositions for each transaction. The use of TLS in combination with STM allows finer-grained parallelism to be extracted from each single-threaded transaction, allowing larger transactions to avoid conflicts by finishing faster. In addition, using an already-concurrent STM application as a starting point allows for more efficient speculation that builds atop existing STM structures, and also relaxes some of the ordering constraints normally present on speculative parallelization systems, as the TLS system only needs to provide semantics equivalent to a concurrent STM execution instead of more restrictive as-if-sequential semantics.

3.3 Method-Level Speculation

Method-Level Speculation — previously introduced in Section 2.4.4 — is a speculative parallelization strategy first discussed in the context of Java by Chen and Olukotun [Chen and Olukotun, 1998], and shown to be a promising source for parallelism by several researchers (e.g. [Oplinger et al., 1999, Warg and Stenström, 2001, Whaley and Kozyrakis, 2005, Pickett, 2012]). This technique works by using method calls as speculative task spawn points — speculatively executing the code following the return of a method call in parallel with the method call itself.

In the following sections, I present an overview of current MLS proposals, describing how they implement the model, and relevant optimizations and limitations.

[Oplinger et al., 1999] investigated the problem of how to find and exploit speculative thread-level parallelism. The authors found that it is inadequate to exploit only loop-level parallelism, as many systems do, and that procedure calls provide important opportunities for parallelism, so they propose the use of speculative procedure execution, coupled with return value prediction. The results presented were obtained by simulating the execution of applications on variations of an optimal speculative thread-level parallelism machine, and show that the combination of loop and procedural speculation is able to obtain speedups over the original sequential versions of six out of the eight SPECint95 benchmarks tested.

[Warg and Stenström, 2001] expand on [Oplinger et al., 1999], also suggesting the usage of methods as a natural boundary for identifying tasks to be speculatively parallelized. The authors analyzed how much parallelism exists in a set of both imperative and object-oriented C and Java applications, and present a study of their execution on top of an idealized machine that supports perfect return value and memory access prediction, and imposes no overhead on thread management and communication. Under this model, the authors were able to obtain a speedup of 3.5x on the examined benchmarks.

3.3.1 Hardware-based MLS approaches

Similarly to the hardware-based TLS approaches previously described in Section 3.2.1, commercial hardware for MLS is mostly non-existent, with current works providing a basis for experimentation on the trade-offs involved in hardware implementations and on their interface with applications and runtime frameworks.

[Whaley and Kozyrakis, 2005] propose and evaluate a number of heuristics for — together with a profiler — identifying profitable methods for MLS parallelization. Heuristics are categorized into three groups: *simple*, *single-pass* and *multi-pass*. Simple heuristics rely on very direct metrics such as method run time and number of memory operations; single-pass heuristics perform a single pass over the profiling data and attempt to estimate the potential speedup or savings from each method; multi-pass heuristics perform multiple passes and attempt to adjust method selection by additionally considering nested speculation. The proposed heuristics were evaluated with Java benchmarks from the SpecJVM98 and SPLASH-2 benchmark suites on a simulated processor with support for speculative execution, leading to speedups of 1.4-1.8x. The authors concluded that multi-pass heuristics are too pessimistic, leading to under-speculation and overly complex spawning decisions, and that the most cycles saved heuristic (SP-CS), which attempts to speculate on methods that are predicted to save the largest number of cycles, was the most successful for the chosen benchmarks.

Helper Transactions [Yoo and Lee, 2008] rely on hardware support for transactions to perform Method-Level Speculation, which the authors refer to as procedure fall-through speculation. MLS is performed as described in Section 2.4.4, including support for nesting speculative tasks, if further methods are called from inside an active speculation. When nested transactions fail, partial aborts allow only the nested transaction to be retried, instead of the entire top-level transaction. The authors propose a new simple scheme to maintain the order in which speculative transactions are allowed to commit, based on a binary tree. In addition, the concept of implicit commit is introduced, allowing a thread that finished working, and after committing its own speculative transaction, to signal its sibling speculation to commit. This sibling speculation — the one that is speculatively running the code that would follow in normal program-order the one that just finished — should then validate itself, commit its current work and continue executing non-speculatively. The advantage of this approach is that the system automatically switches off speculative execution for selected threads as soon as possible, lowering the impact of speculative execution overheads. The authors of this work also go into detail on how the chosen processor architecture with Hardware Transactional Memory support can be expanded to support the features needed for speculative execution. Unfortunately, hardware with the very complex transactional support required is non-existent, and no results are included.

3.3.2 Software-based MLS approaches

Software-based MLS approaches speculate using methods as a boundary for task creation, but unlike hardware-based approaches they do not rely on speculative extensions to multiprocessor hardware.

In their work on safe futures for Java, [Welc et al., 2005] propose extending the Java programming model with support for safe futures that are guaranteed to respect serial execution semantics. Safe futures can be thought of as semantically transparent annotations on methods, designating opportunities for concurrency: Execution of a method can be replaced with the execution of a safe future, with the safe future model guaranteeing that sequential execution semantics is respected and observed behavior of serial and concurrent tasks is unchanged. Compared to other parallelization approaches, programmers have to change their code explicitly to employ futures. In cases where the return value from a method call is used immediately, programmers must additionally refactor their code to delay such use. At run-time, the proposed transactional system helps to avoid typical parallel programming pitfalls when multiple tasks concurrently access and mutate shared data. The model presented is similar to a Software Transactional Memory that has been augmented with constraints on the order in which transactions are allowed to commit, and also with additional features to deal with uncaught exceptions. The authors implemented the proposed model on top of the Jikes Virtual Machine [Alpern et al., 2000]; testing with the Java Grande and OO7 benchmarks with varying workloads revealed speedups ranging from 1.33x to 4x over the original sequential version of the applications running on the same VM on a quad-core machine. Zhang and Krintz's Safe DBLFutures [Zhang and Krintz, 2009] also propose a similar approach, extending it to support the safe handling of exceptions while respecting sequential semantics.

[Hu et al., 2003] studied the importance of return value prediction (RVP) to MLS and similar speculative schemes, showing that RVP could provide clear performance advantages by simulating the execution of multiple benchmarks on a specially modified Java VM. [Pickett and Verbrugge, 2004] also studied multiple prediction strategies and proposed a hybrid design that attempts to choose dynamically the best predictors for each call site.

SableSpMT [Pickett, 2012] is a Java software TLS runtime and framework that employs Method-Level Speculation and includes support for performing return value prediction. Nested speculation is not allowed, limiting some of the achievable parallelism: Although the main thread is allowed to spawn multiple speculative tasks, the tasks themselves cannot spawn further speculative tasks. The author also presents an analysis of which Java bytecode instructions need special handling to support speculation, and what side-effects their execution may cause. Before running an application, SableSpMT performs a static analysis and modification pass on the input application: This pass inserts spawn points into the application bytecode, and gathers information to be used by the return value predictor. In addition, before being used, each of the application's methods is duplicated and modified to create a speculation-safe version; the original version is kept around to be used when code is running non-speculatively. SableSpMT is based on a modified SableVM virtual machine, which unfortunately includes only an interpreter and a very simple garbage collection algorithm. The system was benchmarked using the SPECjvm98 benchmark suite, but no speedup was achieved over the original application run times due to the added overheads. In further testing with fork and join overheads factored out by instead considering a baseline execution where every speculation fails to commit at the end, SableSpMT was able to achieve a mean relative speedup of 1.3x.

HotSpec [Schätti, 2008] was an attempt of continuing the work started in [Costanza, 2007] of implementing Method-Level Speculation on top of a state-of-the-art Java Virtual Machine. The chosen VM for this work was the OpenJDK HotSpot JVM. This work describes many of the modifications needed to the VM, and also how to deal with bytecode rewriting, non-transactional operations, and synchronization between speculative threads; it also generically documents a lot of the inner workings of the VM itself, the object and frame layouts, how it starts up, and how the interpreter interacts with the just-in-time compiler. Unfortunately, owing to HotSpot's complexity and even though it was aiming at a simple implementation with several limitations — only using the interpreter, no object nor array allocation during speculation, no nested speculations and very simple return value prediction — the work was not completed, and no results from benchmarking were presented. In the implementation I propose in this dissertation I also employ the OpenJDK HotSpot JVM, but, as was previously introduced in Section 2.4.3, to avoid the same issues faced by HotSpec I instead work atop the HotSpot JVM, with minimal modifications performed to the VM itself.

HydraVM [Saad et al., 2012] is a custom Java VM built on top of the Jikes JVM's [Alpern et al., 2000] baseline compiler that extracts parallelism from Java applications by splitting their code into parallel semi-independent blocks called superblocks that are then executed with support from STM. Superblock identification can be done either online or offline, and it works by modifying the VM to identify basic blocks and their accessed variables, and by tracing the execution of those basic blocks to form a program execution graph. After an execution graph is identified, it is factored into independent superblocks, representing subsets of the graph. While the superblock approach has the advantage of capturing both loops and method invocations, it is not clear how polymorphism and other control abstractions are dealt with by the system. Speculative code execution is supported by an STM that was integrated into the VM itself. The obtained results are very positive, with speedups of between 2x-5x on the chosen benchmarks. Unfortunately, the Jikes baseline compiler chosen for this work is the simplest of Jikes' compilers, performing no optimizations, and its resulting code is documented as executing only "somewhat faster"[1] than an interpreter. As such, the baseline for comparison used on this work is much lower and less realistic than if a production VM/compiler was chosen, as I explore in Chapter 8.

3.4 Hardware-based Transactional Java Execution

In this section, I explore proposals which build upon (simulated) hardware support for transactional code execution, applying it to the Java Platform. While different from software-based parallelization approaches, these works share some of the same stumbling blocks that a software-based Java parallelization framework faces, such as dealing with non-transactional code and inter-thread result communication.

[Carlstrom et al., 2005] investigate the implications of using hardware transactional support to execute existing parallel Java applications. The general approach proposed is the transformation of `synchronized` blocks into atomic transactions, similarly to speculative lock elision [Rajwar and Goodman, 2001]. The authors argue that strong transactional atomicity semantics is a natural replacement for the critical sections defined by `synchronized`. Also discussed are the problems underlying calls to native machine code through the Java Native Interface (JNI), and of non-transactional operations. The authors conclude that by building atop hardware support for transactions, existing parallel Java applications can be run transactionally with minimal changes, and that a continuous transactional system — one where each thread consists of a sequence of transactions, and there is no execution outside of transactions — can deliver performance equal to or better than a pessimistic lock-based implementation of the same application.

[Carlstrom et al., 2006] present the Atomos programming language, a programming language with implicit transactions, strong atomicity, and a scalable multiprocessor implementation. Atomos is derived from Java, replacing Java synchronization and conditional waiting with transactional alternatives. The current implementation is based on the Jikes JVM [Alpern et al., 2000], and on the Transactional Coherence and Consistency (TCC) Hardware Transactional Memory model [McDonald et al., 2005]. In Atomos, transactions are defined by an `atomic` statement that conceptually replaces the use of `synchronized` statements. The `watch` statement allows programmers to specify fine-grained watch sets, that are used with the `retry` conditional waiting statement for transactional conflict-driven wakeup; this is similar to the approach introduced by [Harris et al., 2005]. Also included are open-nested transactions [Moss and Hosking, 2006], using the `open` statement, which are nested transactions that can be committed and their results seen by other transactions, even while the parent transaction is still active; this allows threads to communicate between transactions, similar to `volatile` variables, while

[1] *Jikes RVM's compilers*, http://jikesrvm.org/Jikes+RVM's+compilers

```
Result solveProblem(Problem) {
    if (problem size is small enough) {
        solve problem
    } else {
        split problem into subproblems
        execute subproblems in parallel // fork
        obtain results from all subproblems // join
        compose result from subresults
    }
}
```

Listing 3.1: General pseudo-code form of a fork/join algorithm.

minimizing the risk of data dependence violations, by limiting violation rollbacks to the open-nested transaction. Transaction commit and abort handlers that run on transaction commit or abort are supported, along with violation handlers that allow programs to recover from data dependence violations without rolling back. Benchmarks were done on a PowerPC CMP system simulator with support for the TCC HTM, with Atomos versions of the tested benchmarks being able to surpass their Java counterparts. Like other emerging parallel-friendly languages, Atomos facilitates the creation of new concurrent applications; in addition, its Java-derived design facilitates the porting of existing parallel Java applications to the new framework, but still requires access to and careful examination of the code base by an expert programmer.

3.5 Java Concurrent Programming

In this section, I examine research works that aim to provide better abstractions for concurrent programming on top of the Java Platform. While their usage depends on manual program modification and is thus not directly comparable to an automatic parallelization framework, their implementation provides valuable insights for any concurrent framework that works atop the JVM, such as the one I propose to build. In addition, and as introduced in Section 2.4.4, the Method-Level Speculation model I propose to use shares a number of similarities with the Fork/Join and similar models presented in this section.

FJTask [Lea, 2000] is a Java framework added to Java 7 that supports Fork/Join parallelism, a style of parallel programming where problems are solved by recursively splitting them into subtasks, which can then be executed in parallel, as shown in Listing 3.1. The framework uses a pool of worker threads that are reused for multiple tasks; tasks themselves are lightweight objects. A special queuing and scheduling discipline is used to manage tasks and map them to the worker threads: each thread maintains a double-ended queue of tasks, which are processed in LIFO order; when their own queue is empty, threads try to obtain tasks by work-stealing [Blumofe and Leiserson, 1999] from other threads. The combination of lightweight objects and a tuned scheduler helps FJTask avoid the pitfalls of standard thread frameworks, which the author argues are too heavy to support this style of programming, imposing unneeded overheads and depending on the operating system's generic scheduling algorithms.

JCilk [Danaher et al., 2005, Danaher, 2005] is a Java-based language for parallel programming that provides call-return semantics for multithreading, allowing a programming style very similar to Fork/Join. It extends Java with three new keywords: `cilk` is used to declare methods that can be spawned to execute in parallel, `spawn` is used to spawn a child method call, and `sync` acts as a barrier, waiting for all spawned computations to complete before continuing. These keywords can be seen as transparent, as their elision results in a correct Java sequential application. JCilk includes very detailed and strict semantics for exception handling, aborting of side computations, and other interactions between threads that try to minimize the complexity of reasoning about them. Similarly to the work by [Welc et al., 2005], programmers need to manually modify (and possibly refactor) their program for executing using JCilk. JCilk itself is implemented via a combination of a compiler which transforms JCilk code first into a version of Java extended with `goto` statements, and afterwards into Java bytecode, and a Java-based runtime library. This combination works atop a normal Java VM, dismissing the need for special support, although the authors document a number of limitations and possible workarounds for their approach.

3.6 Enabling More Parallelism

In this section, I examine works that focus on some of the challenges of extracting speculative parallelism from sequential applications.

In [Baugh and Zilles, 2008] the authors present a study of the implications of non-transactional operations (or side-effects) for Transactional Memory. Such operations are defined as those that, while having effects on the system in which they are running, the TM is not able to provide either conflict detection or rollback — such as for system calls and input/output operations. The authors draw on previous work [Harris et al., 2005, Harris, 2005, Blundell et al., 2006b, Moss and Hosking, 2006, Zilles and Baugh, 2006] on how to handle side-effects in transactions — outlaw, defer, go non-speculative, or compensate — and study their suitability in the context of critical sections in two real-world applications: the MySQL RDBMS and the Mozilla Firefox web browser. In addition, they also analyze both how often and when these operations are executed inside critical sections. The authors conclude that most of the system calls used in the studied applications can be made transactional via the addition of both compensation code and a transactional file system; and also that the technique of deferring input/output operations until the commit of a transaction is of limited usefulness, as many of the results from system calls performed while in critical sections are used within those same critical sections.

[Rountev et al., 2010] studied the parallelism available on multiple Java sequential benchmarks, and propose that parallelization be broken into two steps: (1) the modification of a sequential program into a still sequential but concurrency-friendly program; and (2) the parallelization itself. The authors also introduce a new technique to help identify parallelism-inhibiting memory accesses, which would be used as part of a tool aiming to solve the first step of the parallelization effort. In this dissertation, my focus is on the second part of the parallelization process, but the creation of tools that aid in the creation of concurrency-friendly applications is invaluable in widening the applicability of automatic parallelization.

Similarly, [Cachopo, 2010] raises the question of whether application developers should aim at producing parallelizable applications — applications that are amenable to automatic parallelization — instead of directly developing parallel applications. Because automatic parallelization is at odds with already concurrent applications due to a number of issues, programmers nowadays end up with manually-parallelized versions of their applications, missing out on techniques employed by automatic parallelization runtimes that are hard to manually (and statically) apply.

[Baptista, 2011] exposed the issues with contention management when used in the context of a transactional system that does speculation, and instead proposed the concept of a conflict-aware task scheduler. Such a task scheduler collects information of conflicts between speculative tasks, later using the information to avoid running two conflict-prone speculative tasks concurrently, freeing up system resources for other tasks. Early benchmarking results from a proof-of-concept implementation of the proposed task scheduler on top of the JaSPEx framework [Anjo, 2009] show a simple *No-Conflicts* policy scheduler attaining speedup when compared to speculative executions with no scheduler.

3.7 Discussion

In this chapter I described multiple approaches to automatic parallelization of applications and also to code execution with (sometimes partial) transactional semantics.

Different strategies for parallelization are suited to different applications, and so far no single strategy seems to subsume all of them. This happens because the overheads introduced by the machinery needed for execution and the strategy used influence each application differently in terms of obtainable speedup — this is specially true for systems that employ speculation. Each presented solution balances the involved trade-offs differently, and as a result is mostly suited to a subset of applications that can take advantage of the chosen features. This happens for both hardware and software-based proposals.

Most hardware-based proposals are not available on any shipping processor, and were only evaluated using simulators [Steffan et al., 2000, Chen and Olukotun, 2003, Liu et al., 2006, Yoo and Lee, 2008]. Nevertheless, these works allow experimentation on the overheads of such implementations, and on the trade-offs between implementing a given feature fully in hardware, using a mix of hardware and software, or using only software. These systems also allow the exploration of the interface between the software runtime system and the hardware.

IBM's Blue Gene/Q [Ohmacht and IBM Blue Gene/Q Team, 2011, Wang et al., 2012] includes, in addition to HTM, support for speculative execution: the processor is able to keep track of transaction ordering and even forwards values from older to newer speculative tasks, making it the most interesting commercially-available solution. Unfortunately, it is only available in large quantities as a building block for massive supercomputers. In the future, it would very interesting to explore how an automatic parallelization framework could be combined with the provided hardware TLS to reduce speculation overheads, resulting in a more efficient execution and bigger speedups for parallelized code.

When doing speculative parallelization, support for loop speculation tends to use very small and lightweight custom-built transaction implementations, whereas support for whole-program transactional execution as needed for Method-Level Speculation tends to be very similar to Software Transactional Memory implementations [Welc et al., 2005, Mehrara et al., 2009, Spear et al., 2010, Pickett, 2012, Barreto et al., 2012]. Many proposals already show some results of cross-breeding ideas from the two fields; nevertheless, issues such as nested transactions and speculation still seem to have yet to be uncovered potential [Diegues and Cachopo, 2012].

Strategies that employ Method-Level Speculation [Oplinger et al., 1999, Warg and Stenström, 2001, Whaley and Kozyrakis, 2005, Pickett, 2012] are promising for exploiting parallelism beyond loops, or in applications where loops are few and complex. Yet, current MLS implementations, especially those targeted at Java [Welc et al., 2005, Whaley and Kozyrakis, 2005, Schätti, 2008, Pickett, 2012, Saad et al., 2012] have a number of limitations, such as needing hardware support, programmer input, or having low practical performance.

For new applications, there are a number of proposals of concurrent-friendly Java frameworks [Lea, 2000, Danaher et al., 2005, Carlstrom et al., 2006] that help developers to structure and divide their application, that simplify the resulting semantics, and also provide runtime engines that automatically manage the parallel execution of program tasks.

Approaches to help a programmer modify [Rountev et al., 2010, Pina and Cachopo, 2011] or even write from scratch [Cachopo, 2010] an application that takes advantage of transactional execution and automatic parallelization, where the developer is aware of the parallelization framework — and thus can give hints and refactor harder-to-parallelize parts of an application — are also promising avenues in concurrency research. While having a different goal, they share some of the same challenges that automatic parallelization approaches face: efficient speculative execution, task identification and management, handling of non-transactional operations, etc.

3.8 Summary

In this chapter, I examine works that target application parallelization, with a big focus on those that are either automatic and/or speculative. I also look into a number of parallel frameworks that work atop the Java Platform, and how they solve a number of issues relating to parallel and speculative execution.

I start with parallelizing compilers, then moving on to Thread-Level Speculation, and finally into Method-Level Speculation. For both TLS and MLS, I examine proposed systems and categorize them into hardware and software-based.

Next, I analyze solutions and frameworks for creating parallel and transactional Java and JVM applications. Afterwards, I describe research work that focuses on how to expose and extract more latent parallelism in sequential applications.

Finally, I end by presenting an overall discussion of the current state-of-the-art for automatically-parallelized and transactional applications where I relate the presented works.

Chapter 4

Building Blocks

In Chapter 3, I described research works that targeted application parallelization, with a heavy focus on both speculative parallelization and Java-based solutions.

The objectives of the current chapter are twofold. In the first part of this chapter, (Section 4.1) I introduce and discuss Transactional Memory, an essential component on which I shall build and extend upon to provide speculative code execution for my proposed automatic parallelization framework. I describe both hardware and software-based implementations of Transactional Memory, again with a special focus on Java-based solutions.

In the second part of this chapter, I provide an overview of important features and characteristics of the Java Platform which are relevant in understanding the design choices for my proposed solution. I start with a generic introduction to the Java Platform in Section 4.2, followed by a more specific introduction to the `sun.misc.Unsafe` API in Section 4.3.

4.1 Transactional Memory

Transactional Memory (TM) [Herlihy and Moss, 1993] was initially proposed as a multi-processor architecture capable of making lock-free synchronization as efficient and easy to use as mutual exclusion.

Transactional Memory provides a new synchronization primitive that allows an application thread to group a number of read and write operations to one or more memory locations as a single in-memory *transaction*.

Transactions are *atomic*: Their effects are seen by other threads in the system as occurring in a single step, and no intermediate states are visible. A transaction ends with either a *commit*, in which case the operations performed by the transaction become visible to the rest of the system, all at the same time, or with an *abort*, in which any changes done during the transaction are discarded, never becoming visible to the rest of the system. Data structures concurrently accessed and mutated while protected by transactions are always observed to be in a consistent state, and atomically move from one consistent state to another during program execution.

Transactional Memories can be categorized into three groups, depending on the requirements for their implementation: Software-based (Section 4.1.1), Hardware-based (Section 4.1.2), or Hybrid (Section 4.1.3).

4.1.1 Software Transactional Memory

Software Transactional Memory (STM) [Shavit and Touitou, 1997] was initially proposed as a software implementation of Transactional Memory targeted at mainstream processors. In turn [Herlihy et al., 2003] introduced support for unbounded transactions, allowing dynamic-sized data structures to be used.

In most STMs, it is the job of the application developer to delimit a transaction, with common options being the explicit use of `begin()/commit()/abort()` API calls, or by using method/block annotations such as `@Atomic` to inform the runtime or compiler that a block of code should be executed with transactional semantics.

STMs can be integrated into an application in different ways. The process of preparing an application so that it can run with transactional semantics under the control of an STM is called *transactification*. Many compiler-based STMs perform automatic transactification of code being compiled [Ni et al., 2008, Mehrara et al., 2009, Korland et al., 2010], while other library-based STMs [Herlihy et al., 2006, Cachopo, 2007] need manual changes to the application's source code.

When using an STM, apart from correctly identifying and delimiting atomic blocks, the application developer no longer has to worry about the correctness nor liveness of the application when multiple threads are interacting and mutating the same shared data: This concern is moved into the domain of the STM developer. This also means that the programmer should not concern himself with synchronization via the use of locks and monitors, as it too becomes the job of the STM.

Many STMs respect *opacity* [Guerraoui and Kapałka, 2008] as a correctness criteria. Under opacity, each committed transaction appears to execute atomically at some point in time between when it is started and when it commits, no operation performed by an aborted transaction is ever visible to any other transaction, and every transaction observes a consistent state during its execution, even if the transaction later ends up being aborted.

Most STMs execute transactions optimistically, but many preserve the option to fallback to pessimistic mode, usually when a transaction has been aborted and retried too many times in optimistic mode. For synchronization, both non-blocking approaches (e.g. [Herlihy et al., 2003, 2006, Fernandes and Cachopo, 2011]), and lock-based STM algorithms (e.g. [Riegel et al., 2006, Dice et al., 2006, Saha et al., 2006, Cachopo, 2007, Ni et al., 2008]) have been proposed.

Since Shavit and Herlihy's initial works, many STM implementations have been proposed, exploring various features and design trade-offs. In the rest of this section, I analyze a subset that I consider most relevant to my work; a more general review of Transactional Memory research can be found on [Marathe et al., 2004, Harris et al., 2010].

In [Ni et al., 2008] the authors describe the design of a compiler-integrated STM system for C and C++ that introduces new first-class language constructs to support Transactional Memory. Within an atomic block, the compiler instruments each shared-memory access to be controlled by the TM, so there is no need for the programmer to change his application manually. To support this, methods that are used inside transactions are duplicated, yielding versions with and without instrumentation. Hints to the compiler may be provided via annotations, such as marking functions (or even entire classes) that do not need to be instrumented for safe execution, or marking those that will only be called from transactional contexts. Whenever a transaction needs to perform a non-transactional operation (such as doing I/O, or calling code outside the compiler's control), the system switches to a serial execution mode that entirely forbids transaction concurrency. The system was implemented on top of a high-performance production compiler, and benchmark testing shows the STM performing similarly to the fine-grained locking implementations of the same benchmarks.

Partly as a follow-up to the previous work, as of version 4.7[1] the GNU Compiler Collection (GCC) includes support for Transactional Memory constructs for C and C++. These are based on a draft specification by Intel [Adl-Tabatabai et al., 2012]. The specification introduces a number of keywords and attributes that can be used to tag regions of code for atomic execution, and defines the expected semantics for their execution, including interactions with C++11's memory model and exception handling. The STM implementation itself is delegated to a library, `libitm`, which follows the specification outlined in [Intel Corporation, 2009]; during code generation, GCC emits the appropriate calls to `libitm` for code that will be executed inside atomic regions.

DSTM2 [Herlihy et al., 2006] is a framework to host pluggable Software Transactional Memory implementations for Java that are supplied as *transactional factories*. To use the DSTM2, programmers provide an interface definition for their desired transactional object, and the framework provides a factory capable of creating instances of transactional objects that implement the supplied interface — this factory and its generated objects are dynamically created via a combination of reflection and bytecode generation. Arrays are supported via a special `AtomicArray` class that includes its own internal synchronization and recovery. The framework includes three STM factories: an obstruction-free algorithm based on DSTM [Herlihy et al., 2003] in both invisible-reads and visible-reads versions, and additionally a factory that employs locking.

The Java Versioned Software Transactional Memory (JVSTM) is a pure Java library implementing a Software Transactional Memory [Cachopo and Rito-Silva, 2006, Cachopo, 2007]. The JVSTM introduces the concept of versioned boxes, or *vboxes*, which are transactional locations that may be read and written during transactions, much in the same way of other STMs, except that along with the most recent value for a given memory location, they also keep a history of past values written to them by older committed transactions. Because the JVSTM keeps all the old versions that may be needed by any of the active transactions on the system, a transaction that got delayed for some reason can still access a previous version of the box, ensuring that it will always perform consistent reads — allowing the JVSTM to satisfy the opacity correctness criteria [Guerraoui and Kapałka, 2008].

Keeping multiple versions also allows read-only transactions in the JVSTM to be lock-free and always guaranteed to commit. Because read-only operations do not need to be validated, no read-set needs to be kept during transaction execution, and thus commit operations for read-only transactions do not need to perform any work, making them very lightweight. The combination of multi-versioning and lightweight read-only transactions makes the JVSTM especially suited for applications that have a high read/write transaction ratio.

There are two versions of the JVSTM that differ on their commit algorithm: The original version of the JVSTM uses a lock-based commit algorithm, whereas more recently [Fernandes

[1] *GCC 4.7 Release Series — Changes, New Features, and Fixes*, https://gnu.org/software/gcc/gcc-4.7/changes.html (2012)

and Cachopo, 2011] described a lock-free commit algorithm for the JVSTM. Because the JVSTM is a normal Java library that exports the vbox and transaction APIs, applications have to be manually modified to interface with the STM when accessing Transactional Memory positions. Also included as part of the JVSTM are specialized transactional arrays [Anjo and Cachopo, 2011a] that have reduced access and memory overheads for read-dominated workloads.

The Deuce [Korland et al., 2010] Java STM framework does automatic transactification of classes by replacing instructions to read/write fields and arrays with calls to the framework. Deuce relies on the `sun.misc.Unsafe` internal HotSpot JVM API that provides very low overhead when writing to, or reading from any memory position.

The automatic transactification part of Deuce changes the application to depend on the `Context` interface. Multiple STM algorithms can then be used with Deuce by swapping implementations of the interface. For each Java class to be transactified, Deuce creates a companion *fields holder* class that, upon loading on the virtual machine, obtains the internal offset of each field of the original class.[2] Each method from the class is then duplicated, with the new version receiving an instance of a `Context` object as an additional parameter; the method's code is also changed, so that any access to a field is replaced with a call to the context API, passing as parameters the target object and the internal offset of the field. These parameters can then be used with the `sun.misc.Unsafe` API (Section 4.3) to read from and write to objects directly, bypassing the need to use costly reflection-based mechanisms. This technique also works with Java arrays: The position being accessed is used to calculate the correct internal offset inside the array instance. Finally, each method annotated with `@Atomic` is modified to start a transaction, invoke the original code of the method, and to, upon return, attempt to commit the transaction. Automatic transactification of the base JDK libraries is supported, but must be done offline before starting the Virtual Machine.

Deuce's automatic transactification process is concerned only with changing field and array accesses: It is the job of the programmer to fix his application so that non-transactional operations such as calling native code or performing I/O are correctly handled inside a transactional context, or to make sure that they are not invoked. In addition, Deuce provides only weak atomicity [Blundell et al., 2006a]: When operating outside an atomic context, the application accesses its fields and arrays directly, which may interfere with the work of the STM and cause data races. Deuce is shipped with implementations of the TL2 [Dice et al., 2006] and LSA [Riegel et al., 2006] STM algorithms.

4.1.2 Hardware Transactional Memory

The rise in interest in recent years for Hardware Transactional Memories has culminated in major vendors such as Intel and IBM shipping commercial processor designs with support for best-effort limited HTM. Early experiments show that while these extensions pose great promise, new techniques still need to be developed for them to better integrate with Software Transactional Memories as a fallback, allowing them to be used for bigger transactions. Current progress guarantees are also very weak, suggesting that new algorithms will need to be developed to take advantage of the characteristics of these new hardware extensions.

[2]In some cases, the fields holder class can also be inlined into the original class.

[Dice et al., 2009] document the results of experimenting with the Hardware Transactional Memory support in early revisions of the Sun Rock microprocessor. Rock supports best-effort HTM: Transactions larger than those supported by most bounded HTM implementations are possible, but are not guaranteed to ever be able to successfully commit. The ramifications of this design decision are discussed in detail, for instance while trying to implement small operations (such as double compare-and-swap) in which it would be desirable to have progress guarantees. The authors explore the usage of a compiler and library combination to support Hybrid Transactional Memory on top of the provided HTM: In a hash table benchmark Rock's HTM is able to outperform both coarse-grained locks and STM, and is able to match a fine-grained locking implementation. When testing with a red-black tree, the results are not so clear, with STM sometimes outperforming HTM, depending on the test configuration. The authors then proposed the use of speculation barriers to avoid issues they faced with failing transactions due to bad interactions with Rock's internal out-of-order processing. Finally, the authors explored the idea of speeding up Java locks with transactional lock elision on a modified Virtual Machine: While micro-benchmarks showed very positive improvement, testing with real benchmarks suggests that Rock's HTM is currently too limited for it to replace most synchronized regions. The authors end by detailing a number of improvements and ideas for future Rock versions and other HTM implementations. By suggesting that, in a real-world implementation, HTM may not always outperform STM, and that in many cases the provided buffers are not enough for larger transactions, this work shows that even with more ubiquitous support for HTM, there will still be — for the foreseeable future — a need for software-based speculation mechanisms. This work also shows that the integration of HTM with a managed runtime such as the Java Virtual Machine is far from a trivial affair. Unfortunately, the Sun Rock processor design was scrapped shortly after Oracle's acquisition of Sun Microsystems in 2009, and there are no plans for its revival.

[Jacobi et al., 2012] describe the implementation of Hardware Transactional Memory included in IBM's zEnterprise EC12 processor family, which has been shipping since late 2012. In the zEC12 processor, HTM support is built by extending the cache structure: extra *tx-read* and *tx-dirty* bits are added to each L1 cache line, and a special LRU extension vector tracks cache lines that are evicted from the L1 and into the L2 cache. As a consequence of this design, the maximum read-set size for a transaction closely tracks the capacity of the L2 cache (1MB in the described implementation). Transactional writes are written to a special *Gathering Store* cache, limiting the write-set size to 8KB. Hardware transactions may execute in one of two modes: constrained or unconstrained. In constrained mode, tight limitations are imposed on the amount of code and instructions allowed inside an atomic block — 32 instructions, no loops or subroutine calls, access to at most 32B of memory, no floating-point operations, and no transaction nesting — in exchange for the assurance that these transactions always eventually succeed. In unconstrained mode, larger transactions can be performed, with more complex operations, but these are not guaranteed to ever succeed. Because of this, and unlike in the constrained mode, programmers are required to provide their own fallback alternatives for unconstrained transactions, using, for instance, locking.

The Intel Haswell family of processors includes support for Restricted Transactional Memory (RTM) [Reinders, 2012]. RTM in Haswell is best-effort: The hardware provides no guarantees that an RTM transaction will ever successfully commit, but a number of guidelines are provided in the developer documentation on how to maximize the chances of success. Also like previous best-effort approaches, programmers are required to specify a fallback path that is executed when an RTM transaction fails. Haswell in addition includes support for Hardware Lock Elision (HLE) in the form of legacy-compatible hints: HLE allows a processor to enter a critical section by starting a transaction and placing the lock for that section in its read-set instead of acquiring it. Afterwards, the instructions from the critical section are executed, and the processor attempts to commit when the instruction to release the lock is reached. This arrangement allows multiple threads to execute the same critical section in parallel as long as they do not perform conflicting operations on each other's data. Whenever an HLE transaction fails, the processor falls back to acquiring and releasing the locks as normal, which leads to HLE transactions having the same progress guarantees as normal locks. A big advantage of HLE is that, due to its implementation as hints, HLE transactions are simply ignored on legacy x86 processors, which fall back to normal locking, allowing the same HLE-enabled program binaries to be used on all x86 machines. While Intel has not published details on many of the implementation internals and specific limits for transactions, [Wang et al., 2014] reports on experiments to characterize some of these limitations.

Unfortunately, as of August 2014, Intel has published erratum microcode firmware updates that disabled both RTM and HLE on all shipping Haswell processors, and also on future Haswell-E/EP and Broadwell-Y, some which have yet to be officially launched.[3] This unfortunate setback pushes back wide availability of transactional hardware for at least a year, especially for end-user machines, as reportedly Intel's interest on HTM is only geared toward servers and workstations.

[Cain et al., 2013] describes the architectural support for Transactional Memory that is expected to be added to a future version of the Power architecture. The proposed implementation targets all levels of the system stack: hypervisors, operating systems, libraries/compilers/runtimes and user applications. The Power HTM provides strong atomicity, but, unlike other implementations, it also includes support for transactional suspension, both implicitly when interrupts are handled, and explicitly by using suspend/resume instructions. Also supported are Rollback-Only Transactions (ROTs), that are still speculative but only partially isolated: ROTs are allowed to observe concurrent changes and commits from other threads without conflicting, while their own changes are kept isolated from the rest of the system — other concurrent threads do not observe tentative changes done by a ROT. The addition of reduced-isolation modes such as ROTs to HTM implementations is very interesting for speculative parallelization: ROTs could be used to isolate the changes performed by speculative threads from a non-speculative program-order thread while at the same time not doing the reverse, enabling value forwarding from the non-speculative thread to the speculative threads. Furthermore, ROTs reduce the amount of meta-data that has to be tracked by the processor.

While commercially-available hardware support for HTM marks a huge leap forward for Transactional Memory as an accepted tool for concurrency, the availability and limitations of such hardware still make a very good case for the need for software-based speculation. In my work, I aim to provide a speculative parallelization solution that does not depend on any of these mechanisms, but of course very relevant future work would be to allow the opportunistic usage of hardware-based mechanisms.

[3]*Intel Disables TSX Instructions: Erratum Found in Haswell, Haswell-E/EP, Broadwell-Y*, http://www.anandtech.com/show/8376/intel-disables-tsx-instructions-erratum-found-in-haswell-haswelleep-broadwelly (2014)

4.1.3 Hybrid Transactional Memory

Hybrid approaches combine both HTM and STM, allowing applications to use unbounded dynamic transactions while taking advantage of HTM for smaller transactions. More importantly, they tackle the challenge of how to efficiently coordinate the concurrent execution of both hardware and software transactions, providing important insights for other software-only systems to follow in their own transition towards a hybrid model.

Hybrid Transactional Memory (HyTM) [Damron et al., 2006] is an approach to implementing a Transactional Memory in software that can use best-effort hardware support to boost performance, but does not depend on it. As such, it works on all computers, with or without hardware support for transactions. When hardware support is available, transactions try to run with this support; if they reach hardware limitations they abort and restart as software transactions. The system is implemented as a prototype compiler based on the Sun Studio C/C++ Compiler and an STM library; the compiler produces code for executing transactions either using HTM or the STM library. An important part of this work is reconciling concurrent hardware and software transactions, especially detection of conflicts between them. The authors hope with this hybrid approach to allow for an incremental and transparent transition between pure Software TM, and Hardware TM, allowing chip designers to gradually introduce transactional support in their chips, without having to commit to unbounded (dynamic) hardware solutions from the start. The authors also included support for contention managers as proposed by [Herlihy et al., 2003].

Rajwar, Herlihy and Lai [Rajwar et al., 2005] propose the Virtual Transactional Memory (VTM) system that, like the HyTM, combines hardware and software approaches. The authors argue that like virtual memory shields the programmer from platform-specific limits of physical memory, so must Virtual Transactional Memory shield programmers from HTM limitations such as transactional buffer sizes, scheduling quanta, and page faults. But, unlike the HyTM, VTM also needs hardware machinery to handle the transition from working in hardware-only mode (HTM) to the unbounded software-based transaction mode.

4.2 Java Platform Introduction

As introduced in Section 2.2, I selected the Java Platform as a case study for implementing the ideas in this dissertation.

Before delving into speculative parallelization of Java applications, in the following sections I introduce important details and characteristics of the Java Platform which influence the practical implementation of the techniques proposed in this dissertation.

4.2.1 Java Bytecode

The Java Platform bytecode instruction set targets an abstract machine with carefully defined semantics. At run-time, the Java Virtual Machine simulates this abstract machine on top of a host platform, with bytecodes being either interpreted or converted into native instructions through the usage of a JIT compiler.

The abstract Java machine itself is a stack-based machine employing 32-bit signed words. This does not mean that the underlying JVM implementation follows this (it usually does not), but the Java bytecodes emitted by the various language compilers — including `javac` — target this architecture.

There are 8 primitive data types in Java: `boolean`, `byte`, `char`, `short`, `int`, `float`, `long` and `double`. The first six are represented on the stack with a single word, leaving `long` and `double` (which are defined as always being 64-bit) to be represented using two words. All types are signed — there are no unsigned variants of primitive types.

In addition to primitive types, we have, of course, objects. Any object instance has an associated class, which must be `java.lang.Object` or one of its subclasses. `Object` references also occupy a single word on the stack — note that this is one case where in practice the JVM will probably use a different representation, such as employing 64-bit pointers on modern machines,[4] making these references in practice as wide as a `long` and `double`, while observably at the level of bytecode still only using a single stack word.

Arrays of objects are also objects, and are direct subclasses of `Object`. Single and multidimensional arrays of primitive types (such as `int[]`, `float[][]`, etc) are also objects with their associated classes being internally generated and managed by the VM.

Java bytecode is usually explicitly typed. In many cases there are different instructions for each of the primitive data types (or sometimes just a subset of them) and for `Object`. This means that many low-level changes must take this into account, and sometimes it becomes necessary to provide individual replacements for each of the low-level instructions. For instance, there are eight different instructions for reading from an array (and here `byte` and `boolean` share an instruction), and eight more for writing. When on the stack, the primitive types `boolean`, `byte`, `char`, `short` and `int` all share the same instructions, and all are represented and operated on as `ints` — this conversion is implicit; the remaining types — `float`, `double`, `long` and `Object` — have separate instructions and representation.

Two simple Java methods are shown in Listing 4.1 and their compiled bytecode can be seen in Listing 4.2. Because we are dealing with primitive types, the same Java code with different types leads to different Java bytecodes being emitted by the compiler: `baload` (line 12) changes into `iaload` (line 29) as we read either from a `byte` or an `int` array; and also `iastore` (line 14) changes into `lastore` (line 33) for similar reasons. In line 13 we can see that as a result from reading a `byte` array an `int` is placed on the stack, due to the aforementioned rule that all types shorter than `int` are implicitly converted to and represented as `int` on the stack. Finally, in line 31 we see an explicit conversion from an `int` into a `long`, which needs two stack words to be stored.

Two things not present in the bytecode example are array boundary checks and null pointer checks. These safety checks are implicitly performed by the Virtual Machine, and are thus not present on the bytecode stream — nor can they be disabled.

The Java stack is not explicitly typed. That is, while the data types on the stack after each instruction (as shown in comments on Listing 4.2) must always be computable , and consistent when different code paths are followed to reach the same instruction, the data types on each stack position do not have to be explicitly declared or kept, and can obviously change during method execution.

[4]Interestingly, Oracle's JVM can optionally employ 32-bit pointers on 64-bit machines using heaps smaller than 32GB, a feature called "compressed oops", as introduced in http://docs.oracle.com/javase/7/docs/technotes/guides/vm/performance-enhancements-7.html

```java
class BytecodeExample {
    public static void copyFirst(byte[] from, int[] to) {
        to[0] = from[0];
    }

    public static void copyFirst(int[] from, long[] to) {
        to[0] = from[0];
    }
}
```

Listing 4.1: Java methods that copy the first element of an array into another array.

```
class BytecodeExample {
    public static void copyFirst(byte[] /*from*/, int[] /*to*/) {
                    // stack: empty
        aload_1     // place reference to 2nd method argument (to) on stack
                    // stack: int[]
        iconst_0    // place integer constant 0 on stack
                    // stack: int[], int
        aload_0     // place reference to 1st method argument (from) on stack
                    // stack: int[], int, byte[]
        iconst_0    // place integer constant 0 on stack
                    // stack: int[], int, byte[], int
        baload      // read from byte array, placing result on stack
                    // stack: int[], int, int
        iastore     // store top of stack into int array
                    // stack: empty
        return
    }

    public static void copyFirst(int[] /*from*/, long[] /*to*/) {
                    // stack: empty
        aload_1     // place reference to 2nd method argument (to) on stack
                    // stack: long[]
        iconst_0    // place integer constant 0 on stack
                    // stack: long[], int
        aload_0     // place reference to 1st method argument (from) on stack
                    // stack: long[], int, int[]
        iconst_0    // place integer constant 0 on stack
                    // stack: long[], int, int[], int
        iaload      // read from int array, placing result on stack
                    // stack: long[], int, int
        i2l         // extend top of stack from int to long
                    // stack: long[], int, long word2, long word1
        lastore     // store top of stack into long array
                    // stack: empty
        return
    }
}
```

Listing 4.2: Bytecode resulting from the compilation of the Java source code from Listing 4.1. Each bytecode instruction is annotated with its effect and the state of the stack after its execution.

```java
int localVar;
localVar = 0;
localVar = "Hello World";
localVar = new Object();
localVar = 3.14f;
localVar = false;
```

Listing 4.3: Pseudo-Java exemplifying the untyped nature of local variables in the JVM, and how they can change during method execution. While this pseudo-code is not accepted by the Java compiler, its bytecode equivalent — saving different data types to the same local variable — is valid and accepted by the JVM.

To simplify presentation, whenever there is need for multiple similar versions of the same code to deal with each primitive type, I may present only a subset — for instance by presenting only one of the methods from Listing 4.1.

Local variables in the JVM are saved into numbered slots. The number of slots for a given method is known in advance: During bytecode generation, a compiler targeting the JVM Platform must determine how many local variable slots will be needed, and save that information as part of the Java class file. An important detail for this work (that will be later used in Section 6.2) is that like the stack, local variables in Java are both untyped and the type contained in a local variable can change during the execution of a method. As an example, the bytecode equivalent of the Java pseudo-code shown in Listing 4.3, even though invalid in Java, is valid in the JVM: `localVar` is assigned a given slot number, and it is possible to write into this slot changing its type to an `int`, a `String`, an `Object`, a `float` and a `boolean` — the values are being written directly, no boxing is happening.

Local variables, method arguments, and the `this` reference all share the same numbered slots, starting at 0. This means that `this` (for non-static methods) will be in the local variable slot 0, followed by each method argument, followed by any local variables employed. Listing 4.4 shows an example method `m()` and Figure 4.1 shows the local variable allocations for `m()`. As `m()` is not static, slot 0 contains the reference to `this`, slots 1-3 contain the method arguments (two slots are needed to represent a `long`, similarly to the stack), then slots 4-6 contain the local variables — a `double` (which, like a `long`, requires two slots) and an `int`. Because this allocation is not fixed, later bytecode instructions are free to overwrite any of these slots with any other values (of any other data types), the only restriction being that `long`/`double` values cannot be broken up and then later read.

The Java Platform also has support for exceptions, allowing for limited non-local control transfers. An instance of `java.lang.Throwable` or any of its subclasses may be thrown at any time, in which case the JVM performs a special control transfer to the corresponding exception handler block. Note that although the Java language distinguishes between *checked* and *unchecked* exceptions, there is no such distinction in the JVM — the JVM specification allows any method to throw any exception at any time. The distinction between checked and unchecked exceptions is actually implemented and enforced by the Java compiler, based on class metadata.

```
public void m(short s, long l) {
    double d = s;
    int i = s;
    ...
}
```

Listing 4.4: Example method used to illustrate local variable slot allocation. This method receives as arguments a `short` and a `long`, and additionally uses a `double` and an `int` in its body.

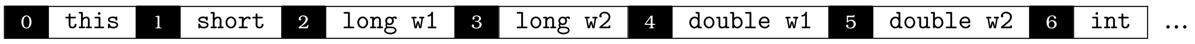


Figure 4.1: Allocation of local variable slots for method `m()` from Listing 4.4. Slot 0 contains the `this` reference, slots 1-3 contain the method arguments, and slots 4-6 contain the local variables.

4.2.2 Bytecode Verification

The platform's restrictions listed in the previous sections (and many others that are included in the Java VM specification [Lindholm et al., 2013]) are enforced by the VM's bytecode verification component, as part of the class loading procedure.

The imposed restrictions can be useful for a tool that needs to understand and manipulate Java bytecode, as they disallow a lot of potentially-complex corner cases. A number of important restrictions are:

- *Type safety.* The VM validates that the types used as part of stack operations, received as arguments, and used for method calling are always correct and without any undefined types — in some cases by mandating that bytecode instructions for run-time testing are included. No unsafe casting is allowed: an object reference may only be casted into its own class, any of its superclasses, or any interface it or any superclass/superinterface implements. Casting between native types is implemented by bytecode type conversion instructions.

- *Integrity.* No memory can be accessed before being initialized (in some cases, there is an implicit 0 or null initialization by the VM), two-word types (long and double) cannot be split up and then used, protection and visibility limitations (private, protected, public or package) are enforced, and only valid instructions/instruction combinations are allowed. There is no way to directly access nor mutate the underlying representation for any native data type.

  Java bytecode cannot directly access VM-internal structures: Selective access and manipulation is provided only via platform APIs, that are internally implemented via native code calls. There is no direct access or any way to control the mapping that may be used between Java bytecode and native machine code; and, in addition there is no access to special purpose or machine-specific processor registers from the host machine.

- *Method Isolation.* Methods are isolated and self-contained: branches can only jump between method-local offsets, never between different methods. If a code block may be reached via several different execution paths, the state of the operand stack must be the same for all paths, and any needed local variables must also be consistent. No self-modifiable code is allowed — apart from using a VM-attached debugger, after a class is loaded its code is not allowed to change. Non-local transfers are only allowed as part of the exception mechanism, with precisely-defined semantics. Execution cannot "fall off" a method.

The combined restrictions and their enforcement allows the Java Platform to safely[5] execute and contain untrusted code. They also guard against a lot of possible bugs in tools that manipulate or generate bytecode, such as the `javac` compiler.

Unfortunately, some of the restrictions also work against the JaSPEx-MLS framework, as they disallow behaviors potentially very useful to the framework. For instance, because there is no access to VM-internal structures, I had to resort to adding special access, as part of the first-class continuations mechanism described in Chapter 5. Also, some of the changes described in Chapter 6 have to work around these restrictions, sometimes at the cost of extra overhead. Fortunately, as we shall see in Chapter 8 many of these overheads end up being mitigated by the VM's just-in-time optimizing compiler.

4.2.3 Bytecode Execution

The Java VM specification only defines the expected semantics for executing Java bytecode, and leaves up to the implementor the freedom of how to best implement them.

While early Java releases mainly relied on interpreted execution of bytecodes (with modern versions of Java bytecode still retaining characteristics that were intended as optimizations for interpreters), modern JVMs such as Oracle's HotSpot VM usually employ one or more just-in-time (JIT) compilers to translate Java bytecode into native host platform code in a technique called tiered compilation.

Tiered compilation works by starting with a very fast compiler (or even with an interpreter), and by gathering statistics for executed code. The goal of this technique is to reduce start-up times for Java applications. After a method is executed enough times, a better JIT compiler kicks in, performing more complex analysis and taking into account the gathered statistics, resulting in higher quality code being generated for often-used parts of the application.

Without the optimizations from the JIT compiler, Java execution can be very slow when compared to natively compiled code, and thus the JIT compiler performs a key role in Java's execution performance.

4.3 Lightweight Low-Level Reflection Using `sun.misc.Unsafe`

To allow for speculative execution of Java code, and as will be later described in Section 6.1.2, the JaSPEx-MLS framework will first need to intercept any accesses to accessible memory locations (fields or array elements), replacing them with calls to a custom STM API. Because this transformation has a big impact on most program code, it is very important that its overheads are minimized.

Internally, the JaSPEx-MLS framework's custom STM needs to be able to reference and access any given field or array, without having it hard coded as part of the STM codebase. To support such use cases, the JDK API includes the `java.lang.reflect.Field` and

[5]Pending any Java VM implementation bugs, of which unfortunately there have been many.

`java.lang.reflect.Array` reflection APIs. These APIs support safe restricted access to fields and arrays, and could potentially be used to implement the inner workings of the framework's STM.

The big issue with the Java reflection APIs is their performance. As documented by Oracle: "*Because reflection involves types that are dynamically resolved, certain Java virtual machine optimizations cannot be performed. Consequently, reflective operations have slower performance than their non-reflective counterparts, and should be avoided in sections of code which are called frequently in performance-sensitive applications*".[6] In addition, because the Java Platform is designed to be safe even in the presence of third-party untrusted code, many reflective operations additionally entail security verifications.

An alternative solution to this predicament is the transformation of fields into transactional objects, as employed by the JVSTM [Cachopo and Rito-Silva, 2006]. This solution was used by the earlier JaSPEx framework [Anjo, 2009] (of which JaSPEx-MLS is an evolution), but performance was also very low: tested benchmarks showed applications executing 10-38x slower; later (unpublished) versions were able to improve on this value, but the added overheads still noticeably impacted application execution performance.

To avoid this performance impact, JaSPEx-MLS instead relies on a special API — the `sun.misc.Unsafe` class — that provides access to fast unchecked low-level operations. This class is present on most modern JVMs, and is used by core JDK classes in special cases as a backdoor to obtain extended platform-level functionality not available otherwise, such as direct memory access and allocation, atomic memory operations, memory barriers, and unchecked exception throwing.

Unlike other JDK APIs, `Unsafe`, as its name implies, really is unsafe. Many API methods on `Unsafe` skip the normal Java verifications listed in Section 4.2.2 of type safety, null pointer checking, array bounds checking, type integrity, etc. It is up to the API client to perform any needed verification before calling the `Unsafe` API; misuse of this API easily leads to memory corruption and VM crashes.

Properly used, operations provided by `Unsafe` are faster and impose smaller overheads than using the Java reflection API or via bytecode manipulation. As such, the STM part of JaSPEx-MLS heavily relies on one very important set of operations provided by `Unsafe`: The ability to directly read from and write to arbitrary memory locations. Listing 4.5 shows the relevant `Unsafe` APIs for performing field and array reflection.

In the `Unsafe` API, an instance field can be referenced using a pair `<Instance, Offset>`, whereas a static field is referenced via a pair `<Class static field base, Offset>`. The `objectFieldOffset(Field)` method maps `java.lang.reflect.Field` instances representing instance fields to their corresponding offsets, and `staticFieldOffset(Field)` is its equivalent for static fields. There is also a method to obtain the static field base object for a given class: `staticFieldBase(Class)`; this static field base object is used as an instance placeholder to represent a class in the `Unsafe` reflection API. Listing 4.6 shows both how a field access is normally performed using the Java reflection API, and how it can be performed with the `Unsafe` API instead.

[6] *Trail: The Reflection API (The Java Tutorials)*, http://docs.oracle.com/javase/tutorial/reflect/index.html

```java
package sun.misc;

public final class Unsafe {
    // Read from a field or array
    public native Object getObject(Object o, long offset);
    public native int getInt(Object o, long offset);
    ...

    // Write to a field or array
    public native void putObject(Object o, long offset, Object x);
    public native void putInt(Object o, long offset, int x);
    ...

    // Map a given instance field to its offset
    public native long objectFieldOffset(Field f);
    // Map a given static field to its offset
    public native long staticFieldOffset(Field f);
    // Map a given class to its static field base object
    public Object staticFieldBase(Class c);

    // Offset of the first element for a given array type
    public native int arrayBaseOffset(Class arrayClass);
    // Scale factor for addressing elements of a given array type
    public native int arrayIndexScale(Class arrayClass);
}
```

Listing 4.5: `Unsafe` API for performing field and array reflection.

```java
public class UnsafeFieldExample {
    static class A {
        public int i;
    }

    public static int getUsingReflection(Object a) throws Exception {
        Class<?> c = a.getClass();
        Field f = c.getField("i");
        return (Integer) f.get(a);
    }

    public static int getUsingUnsafe(Object a) throws Exception {
        Class<?> c = a.getClass();
        Field f = c.getField("i");
        sun.misc.Unsafe unsafe = /* ... */;
        long offset = unsafe.objectFieldOffset(f);
        return unsafe.getInt(a, offset);
    }
}
```

Listing 4.6: Sample class that provides access to field `i` from class `A` using both the Java reflection API and the `Unsafe` API. Notice that the `Field.get()` operation from the reflection API needs to box the value of `i` inside an `Integer`, while `Unsafe` avoids the extra overhead from the boxing operation.

```java
public class UnsafeArrayExample {
    static final sun.misc.Unsafe UNSAFE = /* ... */;

    static final int INT_ARRAY_BASE = UNSAFE.arrayBaseOffset(int[].class);
    static final int INT_ARRAY_SHIFT = /* ... */;

    // Alternative implementation of java.lang.reflect.Array.getInt()
    // Returns array[index]
    public static int readArray(int[] array, int index) {
        return UNSAFE.getInt(array, ((long) index << INT_ARRAY_SHIFT) + INT_ARRAY_BASE);
    }
}
```

Listing 4.7: Example usage of the `Unsafe` API to access an array. The `readArray()` method provides the same functionality as the `java.lang.reflect.Array.getInt()` method on the reflection API — programmatic access to an array position.

Arrays can also be accessed using this API. To access arrays, the array itself is used as the instance argument. The offset can be calculated by shifting the target array position with a constant value called the *array shift*, that can be obtained with information from the API, and then by adding another constant, the *array base*. There are different shift and base values for each primitive type, and also for `Object` arrays. Listing 4.7 shows an array access example using the `Unsafe` API. While not immediately apparent, the advantage of using this API for accessing arrays instead of a normal access is that it both avoids the overhead from bounds checks (which are not performed by `Unsafe`) and it is exactly the same that is used for fields, allowing a simpler STM implementation.

4.4 Summary

In this chapter, I introduce important building blocks for the implementation of a Java software-based speculative parallelization framework.

I start by introducing the concept of Transactional Memory, and then move on to describing the common characteristics of its software-based implementations, with a special focus on Java-based solutions.

Afterwards, I look into emerging commercial Hardware Transactional Memory implementations, and also hybrid works that attempt to reconcile hardware and software transactional mechanisms.

Next, I present an introduction to the Java Platform, including important features and characteristics that will need to be considered as part of my implementation.

Finally, I quickly describe a special low-level API present on some Java virtual machines that allows lightweight reflection. By building atop this API, more efficient Java-based STM implementations can be developed.

Chapter 5

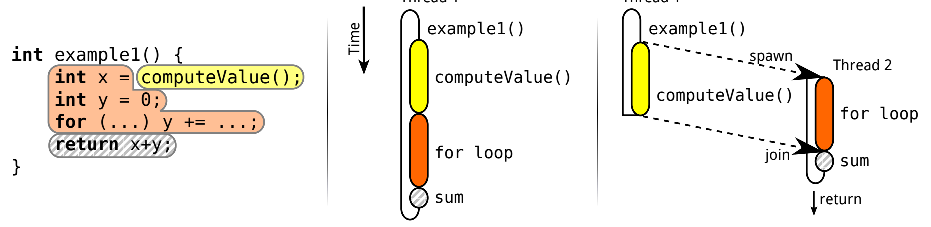
First-class Continuations on the JVM

Implementing Method-Level Speculation in Java poses a number of challenges, due to the unique way that programs are divided into tasks for speculative execution.

Consider my original example for Method-Level speculation, re-reproduced in Figure 5.1. The transformation of the method call to `computeValue()` into a speculative task spawn point begets some questions:

- How can Thread 2 jump to the middle of the `example1()` method, and start execution of the `for loop` without first invoking the `computeValue()` method?
- What about any possible local variables or method arguments? How can they be transferred between threads?
- What about pending method invocations? In the original, sequential version, the same thread entered and returned from `example1()`, but the same does not happen in the MLS-parallelized version: Thread 1 enters the method, but Thread 2 performs the return.
- What if it all goes wrong and the speculative execution fails? What if, as illustrated in Figure 2.3, the speculation spans across several methods and then fails?

All of these questions suggest the need for a mechanism that is both able to transfer program state between threads, and that also allows the snapshotting and later restoration of an earlier



```
int example1() {
    int x = computeValue();
    int y = 0;
    for (...) y += ...;
    return x+y;
}
```

Figure 5.1: Re-reproduction of Figure 2.2: Execution of the `example1()` method (left) when run normally (center) and parallelized with MLS (right). Note that `computeValue()` is executed in the normal program order (at the start of `example1()`), whereas the `for` loop is executed speculatively.

stack state whenever a speculative execution fails. To perform these operations, the JaSPEx-MLS framework needs to have privileged access to a thread's private state: its call stack, its method frames including local variables, and even the program counter.

This kind of mechanism is usually described under the concept of *first-class continuations* — a concept that is part of many high-level programming languages, such as Smalltalk, Scheme, and Haskell, and that can be used as a basis on which to implement programming patterns such as coroutines [Haynes et al., 1986] and generators [Tismer, 2000].

Unfortunately, Java and the Java VM are not among the list of languages and runtimes that natively support continuations. There have been multiple proposals for extending the Java Platform with continuations, which can be divided into two big groups: bytecode-based [RIFE Team, 2006, Mann, 2008, Ortega-Ruiz et al., 2010] and VM-based [Dragos et al., 2007, Stadler et al., 2009, Yamauchi, 2010].

Bytecode-based approaches work by modifying application bytecode to keep an alternate representation of a thread's state on the heap, allowing this state to be saved for later replay, and also by modifying methods so that the entire call stack can be rebuilt from the saved representation. Although these approaches have been proven to work successfully and are applicable to any JVM, they suffer from very large execution overheads: These overheads originate from the continuous need to maintain and update the alternate representation of a thread's state. Regrettably, these overheads are always present, even when an application never actually tries to capture or resume any continuations, as the bytecode transformations they require are performed before the bytecode is loaded onto the VM. In addition, the irregular (when compared to normal Java programs) and complex bytecode resulting from the transformations performed is harder for the VM's just-in-time compiler to optimize successfully.

VM-based continuations work by extending the Java VM, adding hooks that allow access to the VM's internal representation of threads. Because the process of capturing a continuation on a VM with native support for continuations directly accesses the VM's internal data structures, no extra information has to be kept and, thus, there is no impact on code execution speed — all the work is performed (and resulting overhead occurs only) during capture or resume operations. Because the JVM specification [Lindholm et al., 2013] does not include continuations, usage of VM-based continuation APIs means that the resulting applications are non-portable and VM-specific, and efforts to standardize on an interface have thus far failed.

5.1 VM-based Continuations on OpenJDK HotSpot

As there are no readily-available JVMs with support for VM-based continuations, I built upon an earlier experimental work by [Yamauchi, 2010] to create a JVM with support for continuations. The chosen VM for this work — as introduced in Section 2.4.3 — is the OpenJDK HotSpot JVM.

The original work by [Yamauchi, 2010] implemented continuations aimed at web servers, where the state of a web interaction was kept inside a continuation between each request/response pair from the same client. This meant that when a continuation was created at the end of each web interaction, the thread state that was saved would no longer be needed until the next request from that client, so the continuation implementation would also, during the capture operation, clear the previously-existing state and reset the thread to a clean state ready to serve the next client. It also had the restriction that each continuation could only be resumed at most once as it was expected that at the end of each interaction a new one would be created. For my custom JVM, I extended this work by removing its restrictions and by optimizing it for JaSPEx-MLS' use case.

To capture a continuation, the VM walks the thread's current stack, creating a byte array representing its state — local variables, method arguments, and pending methods. During this walk, if either a monitor is being held, or native code is found, an error is generated — in the first case to enforce the restriction that only a single thread may hold a lock, and to avoid deadlocks, and in the second case because native code cannot be safely saved and restored, as the VM has no control over it. In addition, because there may be object references that are only reachable via the references present on the thread stack, and to ensure that the garbage collector does not collect them, the VM also creates an object array containing each object that was referenced by the collected stack. This strategy allows the continuation implementation to be fully independent from the garbage collection algorithms employed by the VM: By keeping all the references from the collected stack on a normal Java array, the continuation implementation is able to ensure that there are always live GC-visible references to any objects that are referenced by the state byte array — which itself is opaque to the GC algorithm — avoiding their erroneous early collection. Both the state byte array and the object array are saved in a new continuation object. After collecting this information — and in contrast to the original implementation — execution is resumed at the instruction immediately following the capture method call, where the newly created continuation is returned.

As a safety measure, the continuation implementation disables some of the VM's optimizations when dealing with the special continuation-capturing methods, to avoid e.g. the unsafe inlining of the capture method.

To resume a continuation, the current thread's call stack is overwritten with the previously-created state byte array. The program counter is then reset to point at the end of the capture method, and execution of Java code is resumed — identically to the capture operation — at the instruction immediately following the capture call, where the now-resumed continuation is again returned. Because the continuation is not changed and is treated as an immutable object, it can be resumed multiple times with no issues. Note that the previously-created object array is never used by the resume operation, it is only kept around as a way of keeping alive all object references that are a part of the state byte array.

The continuation object and its internal state — the state byte array and the object array — are all normal Java objects. Whenever a continuation becomes unreachable, it can be normally collected by the GC, causing its internal state and any heap object that was only being kept alive by references on the object array to also become unreachable and eligible for collection.

To avoid a direct dependence on the custom JVM, I developed the `contlib` library, which provides a VM-agnostic API to use continuations, with the goal of allowing future portability to other VMs. As a test for this API, I also developed an experimental version of `contlib` that wraps the low-level coroutine APIs provided in [Stadler et al., 2009] (regrettably, in its current version the coroutine-providing VM is still unstable).

The `contlib` API is shown in Listing 5.1. Before a continuation can be captured, a new *continuation scope* must be started, by passing an instance of `Runnable` that then invokes the program code. Only inside this scope are the `capture()` and `resume()` operations available. The

```java
package contlib;

public class Continuation {
    /** Setup methods, to be used only outside a continuation scope: **/

    // Start a new continuation scope where continuations can be
    // captured/resumed, and either run the Runnable r or resume the
    // already-existing Continuation c
    public static void runWithContinuationSupport(Runnable r);
    public static void runWithContinuationSupport(Continuation c);

    /** Available inside a continuation scope: **/

    // Create a new Continuation containing the currently-active thread's state
    public static Continuation capture();

    // Resume a previously-captured Continuation
    public static void resume(Continuation c);

    /** Other: **/

    // Is this Continuation freshly-captured, or has it been resumed?
    public boolean isResumed();
}
```

Listing 5.1: `contlib` API for JVM continuations. The `runWithContinuationSupport()` methods start a continuation scope, inside of which the remaining methods can be used.

`isResumed()` method is needed because whenever a continuation is resumed, execution jumps to the return of the `capture()` operation again, and the application code may need to distinguish if this was the first time that `capture()` returned, or if, on the contrary, the continuation was just resumed.

An example showing the usage of this API is shown in Listing 5.2. In the example, a new continuation is captured and stored in `c`, `val` is decremented and printed, and the continuation is then resumed. As a result, the execution jumps back to the `capture()` operation, where the code repeats itself. Note that `val` is an instance variable of the `Runnable`, meaning that it is stored on the Java heap and as such is not saved inside the continuation object. If, instead, `val` was declared as being a local variable inside `run()`, and as a consequence was stored on the stack, it would lead to an infinite loop, as at each `resume()` its value would be reset to the one saved inside the continuation.

A limitation with the current continuation implementation is that the VM is not able to capture and restore the state of native code invocations. Unfortunately, HotSpot's own implementation of the Java reflection API is built using native code, making it impossible to combine reflection with continuation usage. As JaSPEx-MLS also relies on reflection, I have developed an alternative reflective invocation system, described in Section 5.2, that relies on run-time dynamic bytecode generation. This strategy allows the JaSPEx-MLS framework to work around the VM's native code capture limitation and combine the use of reflection with continuations.

Besides providing a simpler API on top of the low-level VM continuations, internally `contlib` is built to avoid memory leaks due to continuations accidentally referencing other continuations.

```
Continuation.runWithContinuationSupport(new Runnable() {
    int val = 5;

    public void run() {
        Continuation c = Continuation.capture();
        if (!c.isResumed()) System.out.println("Capture was successful!");
        System.out.print(--val + " ");
        if (val > 0) Continuation.resume(c);
        System.out.println("Finished!");
    }
});

// Output:
Capture was successful!
4 3 2 1 0
Finished!
```

Listing 5.2: Example usage of the `contlib` JVM continuations API to simulate a loop. At each step, `val` is decremented and printed; execution is then reset back to the end of the `capture()` instruction and the method is repeated until `val` becomes 0.

These memory leaks can very easily happen with the original low-level continuation API, as the resume operation can cause a reference to the resumed continuation to stay alive on the stack, causing a newly-captured continuation to refer to it; the cycle is then repeated if the newly-captured continuation is resumed and then a newer one is captured. This chain of references implicitly forms a linked list of continuations, disallowing the garbage collector from collecting older unused continuations (and all of their referenced objects), leading to increased memory usage and to the possible exhaustion of available memory. `contlib` avoids this issue by using a thread-local object to keep a temporary reference to the continuation being resumed; after the resume operation successfully completes, this temporary reference is cleared, leaving no unintended direct or indirect references to the newly-resumed continuation on the stack.

In addition to being used as a basis for the implementation of the JaSPEx-MLS framework, the extended VM and `contlib` have also been used to power speculative execution in the context of distributed Software Transactional Memories [Fernandes, 2011, Peluso et al., 2012].

5.2 Alternative Reflective Invocation Mechanism

A very important implementation detail of the continuation-supporting JVM described in the previous section is that the state of native method invocations cannot be saved as part of the continuation capture operation; This includes methods that are pending on a thread's stack, even if there are other normal methods on top of them.

This poses a challenge to JaSPEx-MLS: The framework needs to be able to dynamically invoke a given method — a feature that is used when a method call is transformed into a spawn point, as we shall see in Section 6.2 — but the implementation of `java.lang.reflect.Method.invoke()` on the HotSpot JVM works by dynamically generating native code that itself performs the invocation. This would normally mean that the usage of continuations could not be combined with reflective method invocation on this JVM.

Listing 5.3 shows `exampleMethod()` being invoked using both normal and reflective invocation using the Java reflection API. When the `capture()` operation is reached via the `invokeNormally()`

```
class Example {
    public void invokeNormally() {
        exampleMethod(10, new Object());
    }

    public void invokeWithReflection() throws Exception {
        java.lang.reflect.Method m =
          Example.class.getDeclaredMethod("exampleMethod", int.class, Object.class);
        m.invoke(this,  10, new Object());
    }

    private void exampleMethod(int arg1, Object arg2) {
        // ok via invokeNormally(), error via invokeWithReflection()
        Continuation.capture();
        System.out.println("Capture was successful!");
        // ...
    }
}

// Output of invokeNormally():
Capture was successful!

// Output of invokeWithReflection():
java.lang.reflect.InvocationTargetException
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:57)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethod...Impl.java:43)
    at java.lang.reflect.Method.invoke(Method.java:613)
    at Example.invokeWithReflection(Example.java:11)
    ...
Caused by: java.lang.IllegalThreadStateException
    at sun.misc.Continuation.save_cont(Native Method)
    at sun.misc.Continuation.save(Continuation.java:101)
    at contlib.Continuation.capture(Continuation.java:102)
    at Example.exampleMethod(Example.java:16)
```

Listing 5.3: Example of the failed combination of reflective invocation and continuation capture on the continuation-supporting JVM. When the `capture()` operation is reached via the `invokeNormally()` method the code works correctly, but when `capture()` is instead reached via `invokeWithReflection()`, the operation fails. This happens because reflection is implemented on HotSpot using `native` code — we can observe this on the above stack trace which references the `sun.reflect.NativeMethodAccessorImpl.invoke0()` `native` method.

```java
package jaspex.speculation.runtime.codegen.Codegen$0$Example;

public final class exampleMethod implements Callable {
    private final Example arg0;
    private final int arg1;
    private final Object arg2;

    public exampleMethod(Example arg0, int arg1, Object arg2) {
        this.arg0 = arg0;
        this.arg1 = arg1;
        this.arg2 = arg2;
    }

    public Object call() {
        arg0.exampleMethod(arg1, arg2); // invoke target method
        return null;
    }
}
```

Listing 5.4: Dynamically-generated `Callable` class for `Example.exampleMethod()`. The generated class includes private fields to store each of the target method's arguments (including the instance on which the method should be executed), a constructor that populates those fields, and a `call()` method that invokes the target method.

method it works correctly, but when `capture()` is instead reached via `invokeWithReflection()`, the operation fails due to the `sun.reflect.NativeMethodAccessorImpl.invoke0()` native method being on the thread stack.

To work around this issue, I designed an alternative reflective invocation mechanism that is implemented using only Java bytecode that is dynamically generated and loaded on-demand by JaSPEx-MLS's class loader. The alternative reflective invocation mechanism works by dynamically generating a new class for each method that is to be invoked. The generated `Callable` class for invoking `Example.exampleMethod()` is shown in Listing 5.4. The generated class includes private fields to store each of the target method's arguments (including the instance on which the method should be executed), a constructor that populates those fields, and a `call()` method that invokes the target method.

Using the generated class we can rewrite the `invokeWithReflection()` method, as shown in Listing 5.5. For the use cases of the JaSPEx-MLS framework, the direct invocation case suffices — the framework knows in advance which methods will need to be invoked dynamically, it only needs the means to do so. The second version, which still partially uses reflection can be used to fully replace the original Java reflective method invocation API: This version uses reflection only to obtain a reference to the dynamically generated `Callable` class and to instantiate it, but the invocation itself which will stay pending on the stack is implemented with normal Java bytecode; the `Callable` does not depend on any native methods (specifically, it not longer depends on the `sun.reflect.NativeMethodAccessorImpl.invoke0()` method seen in Listing 5.3).

At run-time, when combined with the JaSPEx-MLS framework's custom `ClassLoader`, a program can request the `Callable` class corresponding to any method, triggering the generation of the class, if needed: Whenever the JVM asks for a class in the package `jaspex.speculation.runtime.codegen` to be loaded, this request is routed to a class that, based on the name of the class being requested — in the example above, class `exampleMethod` in

```java
// Direct invocation:
public void invokeWithReflection() throws Exception {
    Callable c = new jaspex.speculation.runtime.codegen.
      Codegen$0$Example.exampleMethod(this, 10, new Object());
    Object ret = c.call();
}

// Or using reflection to instantiate the caller:
public void invokeWithReflection() throws Exception {
    Callable c =
      (Callable) Class
        .forName("jaspex.speculation.runtime.codegen.Codegen$0$Example.exampleMethod")
        .getConstructor(Example.class, int.class, Object.class)
        .newInstance(this, 10, new Object());
    Object ret = c.call();
}
```

Listing 5.5: Implementing the `invokeWithReflection()` method from Listing 5.3 using JaSPEx-MLS's alternative reflective invocation mechanism. The direct invocation example hard codes the reference to the `Callable` class, while the second example uses reflection to obtain and instantiate the `Callable`. In either case, as the `Callable` class is normal Java bytecode, no native methods will remain on the Java thread stack.

package `Codegen$0$Example`[1] — identifies the target method, analyses it, and generates the corresponding `Callable` class, which is then loaded into the VM.

An additional challenge for this mechanism is caused by the protection modifiers of the target method. If a target method is `private` or `protected`, the `Callable` class is not allowed by the Java specification to invoke it. At first glance it may appear that simply changing the modifier to `public` during bytecode preparation would be enough, but this modification is erroneous. Consider for instance the example shown in Listing 5.6: Executing class `B` would yield "`Right: A.m()`", because `runM()` from `A` directly invokes its own local `m()`, as it is a private method, and the Java semantics specify that private methods are not overridable. If instead `A.m()` is changed into a public method, executing class `B` would now yield "`Wrong: B.m()`", indicating that the program semantics has been changed.

To fix this, whenever the framework finds a private method, in addition to changing it from `private` into `public`, JaSPEx-MLS applies name mangling so that the resulting name is unique, avoiding accidental overrides. An example for name mangling is shown in Listing 5.7: Method `m()` is renamed to `m$private$A()`, all references to it are updated, and the method is safely made public.

Currently a limitation of this mechanism is that it cannot be used to invoke a specific method in a hierarchy of classes with overridden methods — it cannot be used as a replacement for calls to `super`. Since these are used sparingly in normal Java applications, and JaSPEx-MLS does not require this feature, I opted against applying further code transformations to solve this issue.

This alternative reflective invocation mechanism has also been extracted into a library —

[1]The extra number in the package name is used to distinguish between multiple method overloads.

```java
public class A {
    private void m() {
        System.out.println("Right: A.m()");
    }

    public void runM() {
        m();
    }
}

public class B extends A {
    public void m() {
        System.out.println("Wrong: B.m()");
    }

    public static void main(String[] args) {
        new B().runM();
    }
}
```

Listing 5.6: Example for the problematic change of method protection modifiers. Executing class `B` yields "`Right: A.m()`", because `runM()` from `A` directly invokes its own local `m()`, as it is a private method, and the Java semantics specify that private methods are not overridable. If instead `A.m()` is changed into a public method, executing class `B` now yields "`Wrong: B.m()`", showing that semantics for the program have changed.

```java
public class A {
    public void m$private$A() {
        System.out.println("Right: A.m()");
    }

    public void runM() {
        m$private$A();
    }
}
```

Listing 5.7: Modified version of class `A` from Listing 5.6, after applying name mangling to `m()`. The method `m()` is renamed to `m$private$A()`, all references to it are updated, and the method can safely be made `public`.

`advice`[2] — that is used to perform bytecode transformation by recent versions of the JVSTM[3] and the Fénix Framework[4].

5.3 Summary

This chapter motivates the need for first-class continuations as a mechanism for implementing Method-Level Speculation. I then present an overview of existing approaches for continuations in Java, dividing them into two big groups: bytecode-based and VM-based.

Because bytecode-based approaches suffer very high overheads, and there are no VM-based approaches available out-of-the-box, I opted for enhancing a previously-existing OpenJDK

[2] *Advice Library*, http://inesc-id-esw.github.io/advice/ (2013)
[3] *JVSTM - Java Versioned STM*, http://inesc-id-esw.github.io/jvstm/
[4] *Fénix Framework*, http://fenix-framework.github.io/

HotSpot patch, optimizing it for the use-cases of Method-Level Speculation.

To avoid a hard dependence on a single VM, I implemented a high-level library for interfacing with continuations — the `contlib` library. This library includes backends for two VM-based continuations, namely [Stadler et al., 2009] and my custom VM. The `contlib` library along with the custom VM provides the APIs necessary for the implementation of Method-Level Speculation, as I shall describe over the next chapters.

In addition, to workaround a limitation with the continuation-supporting VM, I introduced an alternative reflective invocation mechanism that allows reflective method calls to be used by JaSPEx-MLS in combination with continuations.

Chapter 6

Bytecode Preparation

Because the JaSPEx-MLS framework is itself programmed in Java, it still needs to work on top of the JVM. Apart from the special support for first-class continuations described in Chapter 5, JaSPEx-MLS has access only to the same APIs that are exposed to any other Java application. To exercise more precise control over the execution of a given application, JaSPEx-MLS needs to perform a number of modifications to its bytecode, adding hooks that will allow the target application to be dynamically controlled.

These needed modifications to the input application's bytecode are performed by the bytecode preparation component of the JaSPEx-MLS framework. Bytecode preparation is done on-demand at class load-time: JaSPEx-MLS includes a custom class loader, and as each class from the target application is requested by the VM, the bytecode preparation is triggered, and afterwards the modified bytecode is loaded. Using this approach, an application may be executed directly, without any kind of offline precompilation step: Bytecode modifications are done in-memory before the code is loaded. A big advantage of this load-time transformation strategy when compared to an offline precompilation step is that it avoids needing to deal with stale versions of transformed classes and possible mismatches of library versions. Optionally, and to avoid preparation overheads, the modified classes can be cached by JaSPEx-MLS for subsequent application runs.

The bytecode modifications performed aim at two main objectives: (1) enabling transactional execution of application bytecode (Section 6.1); and (2) inserting speculation spawn points and futures into the application (Section 6.2). Whereas the first objective is solely concerned with allowing parts of a program to be executed concurrently with no interference between them, the second takes care of adding the necessary hooks so that JaSPEx-MLS can create and coordinate speculative task execution.

Note that, to simplify presentation, many of the presented examples use plain Java, rather than Java bytecode, but no transformations are ever done by JaSPEx-MLS at the source code level.

6.1 Transactional Bytecode Execution

To support transactional bytecode execution, JaSPEx-MLS prepares bytecode so that it can safely execute with transactional semantics, under the control of the framework's custom

Software Transactional Memory implementation. In particular, the objective of this interception is to allow the STM to provide two important features to speculative code execution:

- *Atomic execution.* A speculative transaction executes in an all or nothing fashion. Either all the changes performed by the transaction become visible to the entire system, or none of them.
- *Isolation.* While a speculative transaction is happening, other transactions are not affected by its actions nor vice-versa.

The JaSPEx-MLS framework needs to have the ability, while executing a given block of code, to both choose *when* its resulting state changes can be applied to the global program state, and *whether* they should be applied at all. Neither unapplied nor discarded changes should be able to interfere with other parts of the system.

Modifying Java bytecode to execute transactionally consists of analyzing and transforming the code, while taking into account:

- *Heap access operations.* In the JVM, these are accesses to fields (either static or instance), and to elements of arrays (Section 6.1.2).
- *Non-transactional operations.* As computing systems are generally not transactional, boundaries between transactional and non-transactional code are always present. On the JVM Platform, these mainly consist of JDK classes, which are not allowed to be modified, and native code, which is difficult to analyze and may call into the operative system (Section 6.1.3).
- *Concurrency primitives and dangerous operations.* Thread-safe single-threaded code has to be modified so that it does not cause deadlocks when executed under the control of the JaSPEx-MLS framework. Operations that may interfere with the framework or that may change their semantics when executed speculatively must be blocked (Section 6.1.4).

Over the next subsections, I explore how to deal with each of these operations, with the goal of obtaining safe transactional execution of Java bytecode. The process of transforming a given block of code into its transactionally-safe version is called its *transactification*, and the output of the transactification process is *transactified* code — as part of a *transactified* class.

The transactification process employed by JaSPEx-MLS is generic — it is independent of the individual speculation technique applied afterwards, and can easily be reused on other systems needing automatic conversion of code for transactional execution.

During transactification of a class, I assume that all other (non-JDK) program classes are also under the control of the JaSPEx-MLS framework and have undergone or will undergo the same set of preparations — that is, I assume that methods on non-JDK classes will behave transactionally. To fulfill this assumption, each individual class loaded by the framework's class loader will be modified to be fully transactional, or at least (as we shall see in coming sections) that any non-transactional methods will be protected by the class itself. Calling any method from any class transactified by the JaSPEx-MLS framework should result in transactionally-safe code execution.

```java
class ExampleC0 extends java.io.FilterReader {
    private int i;
    private static Object o;
    private final long l;

    public ExampleC0() {
        super(null);
        i = 0;
        o = new Object();
        l = ++i;
        int[] tempArray = new int[10];
        tempArray[5] = i;
    }

    public void doSomething() {
        System.out.println(toString());
    }

    public void doSomethingElse() {
        doSomething();
    }
}
```

Listing 6.1: Running example for the bytecode transactification process. The `ExampleC0` class extends the JDK-provided `java.io.FilterReader` class.

As a running example for the bytecode transactification process, I shall use the sample class presented in Listing 6.1 to show the various features of the transactification process.

6.1.1 `Transactional` Classes and `$transactional` Methods

As a first step of the modification process, a class being prepared for parallelization is modified to implement the `jaspex.speculation.runtime.Transactional` interface. This allows instances of classes changed by the JaSPEx-MLS framework to be easily identified at run-time by using a simple "`reference instanceof Transactional`" test.

The framework then duplicates every method `m()` in a transactified class into a `m$transactional()` method. An exception to this is that constructors can only be named `<init>()`, and cannot be renamed: In this case, an extra argument of type `jaspex.MARKER.Transactional` is added to each copied constructor, allowing the two versions of the method to be distinguished by their differing signatures.

The need for duplicating methods and preserving the originals is motivated by two main reasons:

- `native` methods cannot be renamed. Because the framework will need to add a call to a special method before executing each native method, as described in Section 6.1.3, the framework will additionally need to add wrapper methods that call into the original, native ones. As the original native methods cannot be renamed — dynamic linking of native code is performed using the method's name and signature — I chose to give the wrapper methods new names.

- In some cases, JDK code may call back into a user class directly, such as when extending or implementing a JDK class, or when using reflection — a simple example is when a Java collection class calls an `equals()` method provided by a user class. Because JDK code is not transactified the interleaving of `$transactional` and JDK methods in this fashion only adds unneeded overhead without providing safe transactional execution.

This transformation is also useful when debugging: Due to the extensive amount of methods added and changed by the JaSPEx-MLS framework, this renaming is helpful because any missing methods due to modification bugs will be detected by the JVM during load-time bytecode verification.

The `$transactional` version of each method is a copy of the original method with invocations to other methods replaced by calls to their `$transactional` versions, if possible; calls to constructors are modified to call the new versions of the constructors that include the previously-added extra dummy argument, passing null as its value. Additionally, for methods inherited from JDK classes that have not been overridden by the class itself, JaSPEx-MLS adds trampoline methods that call the original non-`$transactional` versions. After these modifications, during transactional execution the program's execution will only flow through `$transactional` methods.

An example of these changes when applied to the `ExampleC0` class from Listing 6.1 is shown in Listing 6.2: The class is modified to implement the `Transactional` marker interface, and each method is copied into a `$transactional` alternative. Each method invocation is modified to point at a `$transactional` version and JaSPEx-MLS also adds `$transactional` trampoline methods for methods inherited from JDK classes. Trampoline methods are added because, as defined before, other transactified classes will assume `$transactional` versions of every method exist, and this modification will additionally be used as part of the run-time prevention of non-transactional operations described in Section 6.1.3.

Because trampoline methods are also injected by the Java compiler to implement some of the Java language's features, the JVM's JIT compiler is already built to optimize and eliminate most of the added overheads from such methods. JaSPEx-MLS's trampolines are designed to be very similar to those used by the Java compiler, so that they are also able to take advantage of these optimizations.

After this first step, all further method bytecode modifications are applied to `$transactional` methods, no changes are ever made to the originals, and the framework proceeds to perform the modifications for transactional execution.

6.1.2 Intercepting Heap Accesses

In the JVM Platform, accessible memory locations are either fields — which may be either static or instance fields — or array elements.

To control any heap accesses done in the application code, the JaSPEx-MLS framework substitutes the bytecodes used for these operations with calls to its own STM API, a technique also used by the Deuce STM [Korland et al., 2010]. Object creation and subsequent garbage collection is still left to the VM; these operations do not need to be intercepted.

These accesses correspond to the following bytecode operations:

```java
class ExampleC0 extends java.io.FilterReader
    implements jaspex.speculation.runtime.Transactional {
    ...

    public ExampleC0() { ... } // unchanged
    public ExampleC0(Transactional dummy) { ... }

    public void doSomething() { ... } // unchanged
    public void doSomething$transactional() {
        System.out.println(toString$transactional());
    }

    public void doSomethingElse() { ... } // unchanged
    public void doSomethingElse$transactional() {
        doSomething$transactional();
    }

    // Framework-generated method, trampoline for method inherited from java.lang.Object
    public String toString$transactional() {
        return super.toString();
    }

    // Framework-generated method, trampoline for method inherited from java.lang.Object
    public int hashCode$transactional() {
        return super.hashCode();
    }

    // Framework-generated method, trampoline for method inherited from java.io.FilterReader
    public void close$transactional() {
        super.close();
    }

    // Framework-generated method, trampoline for method inherited from java.io.FilterReader
    public void reset$transactional() {
        super.reset();
    }

    // ... more methods inherited from java.io.FilterReader and its superclasses
}
```

Listing 6.2: `Transactional` version of the `ExampleC0` class. The class is modified to implement the `Transactional` marker interface, and each method is copied into a `$transactional` alternative. Each method invocation inside a `$transactional` method is modified to call other `$transactional` methods, and JaSPEx-MLS also adds `$transactional` trampoline methods for methods inherited from JDK classes.

```
// Instance loads
public static Object loadObject(Object instance, Object value, long offset);
public static int loadInt(Object instance, int value, long offset);
//... variants for the remaining types: loadBoolean(), loadByte(), loadChar(),
// loadDouble(), loadFloat(), loadLong(), loadShort()

// Static loads
public static Object loadObject(StaticFieldBase sfb, Object value, long offset);
public static int loadInt(StaticFieldBase sfb, int value, long offset);
//... variants for the remaining types (similar to those above)

// Instance stores
public static void storeObject(Object instance, Object value, long offset);
public static void storeInt(Object instance, int value, long offset);
//... variants for the remaining types: storeBoolean(), storeByte(), storeChar(),
// storeDouble(), storeFloat(), storeLong(), storeShort()

// Static stores
public static void storeObject(Object value, StaticFieldBase sfb, long offset);
public static void storeInt(int value, StaticFieldBase sfb, long offset);
//... variants for the remaining types (similar to those above)
```

Listing 6.3: STM API for transactional field access. Each native type (and `Object`) has its own set of methods for load/store and for both the static and non-static variants. Argument order is changed between instance and static variants of the store method as an optimization for the expected argument order on the bytecode stack after transactification (avoiding one extra `swap` operation).

- `putfield`/`putstatic`: write to normal/static field
- `getfield`/`getstatic`: read from normal/static field
- `aaload`/`iaload`/`baload`/.../`laload`/`saload`: load from array (typed instruction)
- `aastore`/`iastore`/`bastore`/.../`lastore`/`sastore`: store to array (typed instruction)

To take control of these operations, JaSPEx-MLS replaces them with calls to a custom STM API. This API — contained in class `jaspex.stm.Transaction` — features a load and store method for each JVM primitive type, with instance and static variants, as shown in Listing 6.3 and, in addition, a load and store method for each array type, as shown in Listing 6.4. By having type-specific variants for each method, JaSPEx-MLS is able to avoid unneeded boxing overhead and facilitates the crucial JIT-compiling and inlining of these methods.

I defer the discussion on how JaSPEx-MLS's STM is itself implemented to Section 7.4, and instead in this section I discuss how STM API calls are injected into the code being transactified. Note that the `offset` values received by the STM API are those employed by the JVM's `Unsafe` API as described in Section 4.3.

As part of the transactification process, the JaSPEx-MLS framework replaces the bytecodes identified above by calls to its STM API. Consider the transactification of field accesses from the example class in Listing 6.1 that is shown in Listing 6.5. In the first case, a write to instance field `i` is replaced by a call to `storeInt()`, receiving as arguments the target instance, the value to be written to the field, and the framework-internal offset of the field. In the second case, a write to static field `o` is replaced by a call to `storeObject()`, receiving as arguments

```java
// Array loads
public static Object arrayLoadObject(Object[] array, int pos);
public static int arrayLoadInt(int[] array, int pos);
//... variants for the remaining types: arrayLoadBoolean(), arrayLoadByte(),
// arrayLoadChar(), arrayLoadDouble(), arrayLoadFloat(), arrayLoadLong(),
// arrayLoadShort()

// Array stores
public static void arrayStoreObject(Object[] array, int pos, Object value);
public static void arrayStoreInt(int[] array, int pos, int value);
//... variants for the remaining types: arrayStoreBoolean(), arrayStoreByte(),
// arrayStoreChar(), arrayStoreDouble(), arrayStoreFloat(), arrayStoreLong(),
// arrayStoreShort()
```

Listing 6.4: STM API for transactional array access. Similarly to the transactional field access API, each native array type (and `Object`) has its own set of methods for load/store.

```java
// Original:
i = 0; // implicit: this.i (instance field of this)
o = new Object(); // implicit: ExampleC0.o (static field of ExampleC0)

// Replacement:
Transaction.storeInt(this /*instance*/, 0 /*value*/, ExampleC0.$offset_i);
Transaction.storeObject(new Object() /*value*/,
  ExampleC0.$staticFieldBase /*sfb instance*/, ExampleC0.$offset_o);
```

Listing 6.5: Transactification of field accesses from Listing 6.1. The write to instance field `i` is replaced with a call to `storeInt()`, whereas the write to `static` field `o` is replaced with a call to `storeObject()`.

the reference to be written, the `StaticFieldBase` corresponding to the `ExampleC0` class, and the framework-internal offset of the field.

Array accesses are also changed into calls to the STM API, as shown in Listing 6.6. Note that in this example the operation "`tempArray[5] = i`" consists on both reading field `i` and then writing its value to the target position of `tempArray`, resulting in two calls to the API being injected, one for each operation.

Generically, a read from a field `targetObject.targetField` is replaced with a call to `loadType(targetObject, targetField, $offset_targetField)` where `Type` is the field's type (either a native type, or `Object`), `targetObject` is the instance we want to read from, and `$offset_targetField` is a special constant injected by JaSPEx-MLS that allows the framework to reference the field internally. Similarly, a write of `newValue` to a field `targetField` is replaced with a call to `storeType(targetObject, newValue, $offset_targetField)`. Static accesses to fields substitute the `targetObject` for the static field base instance representing the class, and additionally the argument order on the `storeType()` operation is reversed. For arrays, the framework replaces a read of `array[position]` of `Type` with `arrayLoadType(array, position)`, and a write "`array[position] = newValue`" with `arrayStoreType(array, position, newValue)`. Table 6.1 summarizes all the substitutions done to interface with the framework's STM API.

The `load*` API to read a field is called with the current value of the field allowing the JaSPEx-MLS framework's STM to quickly perform field reads when executing in non-transactional mode, as it can immediately return that value.

Operation	Pattern	Replacement
read instance field	*targetObject.targetField*	load*Type*(*targetObject*, *targetObject.targetField*, *targetClass*.$offset_*targetField*)
write instance field	*targetObject.targetField* = *newValue*	store*Type*(*targetObject*, *newValue*, *targetClass*.$offset_*targetField*)
read static field	*targetClass.targetField*	load*Type*(*targetClass*.$staticFieldBase, *targetClass.targetField*, *targetClass*.$offset_*targetField*)
write static field	*targetClass.targetField* = *newValue*	store*Type*(*targetField*, *targetClass*.$staticFieldBase, *targetClass*.$offset_*targetField*)
read array	*targetArray*[*targetPosition*]	arrayLoad*Type*(*targetArray*, *targetPosition*)
write array	*targetArray*[*targetPosition*] = *newValue*	arrayStore*Type*(*targetArray*, *targetPosition*, *newValue*)

Table 6.1: Transformation of heap access bytecodes into STM API calls. For field accesses, I consider accesses to the field *targetField* of type *Type*, belonging to object (if applicable) *targetObject* of class *targetClass*. For arrays, I consider accesses to position *targetPosition* of array *targetArray*.

```
// Original:
tempArray[5] = i; // implicit: this.i (instance field of this)

// Replacement:
Transaction.arrayStoreInt(tempArray /*array instance*/, 5 /*array position*/,
  Transaction.loadInt(this, i, ExampleC0.$offset_i) /*value*/);
```

Listing 6.6: Transactification of write to array from Listing 6.1. The array write is replaced by a call to `arrayStoreInt()`, and the read of field `i` is replaced by a call to `loadInt()`.

Because local variables are stored locally on the thread stack, they are not part of the program state stored on the heap. As each thread operates on its own local state (and due to the chosen MLS parallelization model), local variables do not need to be changed to use JaSPEx-MLS's STM, allowing local computation to be performed without any added overheads. Whenever needed, the stack can instead be checkpointed: it can be saved and restored via the previously-described first-class continuation implementation from Chapter 5.

Final fields (such as `final long l` from line 4 in Listing 6.1) are not transactified. Because after initialization they are not changed, the framework may allow direct reads from them — bypassing the STM API — as well as the direct write that performs initialization (line 10 in Listing 6.1). To apply this optimization even when accessing a `final` field from another class, JaSPEx-MLS additionally analyzes outbound field references to distinguish between normal and `final` fields, so as to avoid adding unneeded instrumentation to those accesses.

In some cases, the JaSPEx-MLS framework user may know of non-`final` fields that are safe to access without STM protection — the user may derive this understanding from knowledge of the application's code and behavior, or they may have been discovered by using a (possibly third-party) profiling tool. To allow for this extra information to be used, the JaSPEx-MLS framework supports receiving as input a configuration file with a list of fields that should not be transactified, and that are expected to behave as if they were `final`.

The `$staticFieldBase` and `$offset_*` fields required by the STM API are also inserted into each modified class: A constant *offset* field is added for each field from the original class (except for `final` fields), whereas the `$staticFieldBase` is used to represent the class when accessing `static` fields. Field initialization for these fields is then simply added to the static

```java
class ExampleC0 extends java.io.FilterReader {
    private int i;
    private static Object o;
    private final long l;

    // Added by transactification process
    public final static long $offset_i;
    public final static long $offset_o;
    // no $offset_l is added because l is final and thus not transactified
    public final static StaticFieldBase $staticFieldBase;

    // Initializations are injected into a new static initializer
    // or prepended to existing class static initializer
    static {
        $offset_i = Transaction.getFieldOffset(ExampleC0.class, "i");
        $offset_o = Transaction.getFieldOffset(ExampleC0.class, "o");
        $staticFieldBase = Transaction.wrappedStaticFieldBase(ExampleC0.class);
    }

    // ... rest of the class
}
```

Listing 6.7: Injection of `$staticFieldBase` and `$offset_*` fields into the transactified version of class `ExampleC0` from Listing 6.1. A constant *offset* field is added for each field from the original class (except for `final` fields), whereas the `$staticFieldBase` is used to represent the class when accessing `static` fields.

class initializer, as shown in Listing 6.7.

With the above modifications, every mutable field and array access performed by the original application code is now placed under the control of the JaSPEx-MLS framework's STM.

6.1.3 Dealing with Non-Transactional Operations

Most real applications employ a number of non-transactional operations — operations that cause side-effects that cannot be undone, and that are hard or even impossible to transactify, as they are outside the control of the JVM. For instance, consider most application-external interactions: accessing and mutating common file systems, communicating with other applications and systems, and user interaction — these are all operations that cannot be easily undone.

In the JVM Platform two types of operations are non-transactional: (1) `native` methods, which are implemented by precompiled platform-native code, and that are hard to analyze and change as they are used to invoke the operating system and native libraries; and (2) code belonging to the JDK, as the JVM does not allow alternative versions of these classes to be loaded by the JaSPEx-MLS framework's class loader.

Per their definition, the JaSPEx-MLS framework cannot transactify non-transactional operations. In earlier work [Anjo, 2009], I showed that attempting to detect the execution of `native` methods via static analysis resulted in the overly pessimistic categorization of most method calls as possibly invoking native code. This happened because statically, among the possible code paths, there may exist an execution flow that eventually leads to a `native` method being invoked, even if this execution flow is a rare corner case that is never taken in practice.

```java
public class ExampleC1 {
    private void doSomething() { ... }

    private native void doSomethingNative();

    public void maybeNative() {
        if (choice()) {
            doSomething();
        } else {
            doSomethingNative();
        }
    }

    private static boolean choice() { ... }
}
```

Listing 6.8: Example showing the `maybeNative()` method. The execution of this method may lead to a `native` method being invoked: Depending on the outcome of the call to `choice()`, either the normal method `doSomething()` or the `native` method `doSomethingNative()` are invoked.

As an example for overly pessimistic detection, consider Listing 6.8: In this example, the execution of a `native` method call depends on the outcome — at run-time — of the `choice()` method. The execution of the `choice()` method controls when the `doSomethingNative()` method is invoked, but statically it is not possible to determine how often (if at all) this will happen and as such it would be overly pessimistic to consider `maybeNative()` as always triggering the execution of the `doSomethingNative()` `native` method.

A further example is shown in Listing 6.9: In this case, because the `Sortable` instances received by the `sortAll()` method may either be `MyList` or `NativeList` instances, static pessimistic detection would lead the `sortAll()` method to be considered as invoking native code and thus not safe to invoke during a speculative execution. At run-time, it is possible that none or only a small number of `NativeList` instances are ever used, revealing the static decision as being overly pessimistic.

Instead, the JaSPEx-MLS framework employs run-time prevention [Anjo, 2009] of non-transactional operations. During bytecode preparation, the framework scans modified classes for direct calls to non-transactional operations, and prepends them with a call to a special `nonTransactionalActionAttempted()` method. This process takes advantage of the previously described (in Section 6.1.1) transformation of classes to use only `$transactional` methods. To intercept execution of `native` methods in user classes, the `$transactional` counterpart of a `native` method is modified to call `nonTransactionalActionAttempted()`, as shown in Listing 6.10.

For JDK methods, a similar technique is applied: method calls to JDK methods are prepended with calls to `nonTransactionalActionAttempted()`, as shown in Listing 6.11. Because considering all JDK methods as non-transactional is overly pessimistic, JaSPEx-MLS includes a whitelist of safe JDK classes and methods, which was compiled via a mix of manual inspection and automatic analysis.

Whenever an application class extends a JDK class (and thus inherits its methods) the framework makes use of the previously added trampoline methods (Section 6.1.1 and Listing 6.2), and prepends the calls to the JDK superclass with a call to the `nonTransactionalActionAttempted()`

```java
public interface Sortable {
    public void sort();
}

public class MyList implements Sortable {
    public void sort() { ... } // normal method
}

public class NativeList implements Sortable {
    public native void sort(); // native method
}

public class CollectionUtils {
    public void sortAll(Sortable[] list) {
        for (Sortable s : list) {
            s.sort(); // is the target of this method call a native or a normal method?
        }
    }
}
```

Listing 6.9: Example where two different classes (`MyList` and `NativeList`) implement the same `Sortable` interface, providing both normal and `native` implementations for the `sort()` method. The `sortAll()` method may invoke `native` methods depending on the instances of `Sortable` it receives at run-time. A static approach to method categorization would thus assume that `sortAll()` is not safe to be executed while speculating, but this may be an overly pessimistic decision if none or only a small number of `NativeLists` are actually used at run-time.

```java
class ExampleC1 implements jaspex.speculation.runtime.Transactional {
    private void doSomething() { ... }
    private void doSomething$transactional() { ... }

    private native void doSomethingNative();

    private void doSomethingNative$transactional() {
        SpeculationControl.nonTransactionalActionAttempted();
        doSomethingNative();
    }

    public void maybeNative() { ... }

    public void maybeNative$transactional() {
        if (choice$transactional()) {
            doSomething$transactional();
        } else {
            doSomethingNative$transactional();
        }
    }

    private static boolean choice() { ... }
    private static boolean choice$transactional() { ... }
}
```

Listing 6.10: `$transactional` version of class `ExampleC1` from Listing 6.8, after the addition of run-time prevention of non-transactional operations. The original `doSomethingNative()` `native` method is unchanged, whereas its `$transactional` counterpart is prepended with a call to `nonTransactionalActionAttempted()`.

```
// Original:
public void doSomething() {
    System.out.println(toString());
}

// Replacement:
public void doSomething$transactional() {
    SpeculationControl.nonTransactionalActionAttempted();
    System.out.println(toString$transactional());
}
```

Listing 6.11: Run-time prevention of the execution of a JDK method (from the running example in Listing 6.1). The call to `nonTransactionalActionAttempted()` allows the framework to control if and when the subsequent `println()` should be executed.

method.

Using this modification technique, each class is able to protect its own `native` and JDK-inherited methods, so that other classes do not have to worry about their invocation. For direct calls to JDK classes, and because those classes cannot be changed by JaSPEx-MLS, the framework places the invocation of `nonTransactionalActionAttempted()` directly in the method performing the JDK method call. Thus, for direct calls to JDK methods the framework changes the caller method, whereas for `native` and inherited JDK methods it changes the callee.

Inside the `nonTransactionalActionAttempted()` method itself, the framework validates if the subsequent operation should be allowed to execute; in some cases, this may involve waiting for some other speculative task to finish its work. After validation, if the framework decides that the non-transactional operation should be allowed to execute, the method returns normally; otherwise the speculative task is aborted and the method never returns.

6.1.4 Handling Concurrency Primitives and Dangerous Operations

Modern programming languages strongly promote code reuse, and it is common for most Java applications to rely on several third-party libraries. Although the application that is being executed by JaSPEx-MLS is supposed to be sequential, the third-party libraries it uses may have been designed to be thread-safe, so that they can be used by both sequential and concurrent Java applications.

Usually, leaving such concurrency primitives in place has little to no impact on sequential applications, but it may lead to deadlocks when the same application is speculatively parallelized by the JaSPEx-MLS framework.

To avoid having to manually change applications and libraries, and as part of the bytecode preparation step, the JaSPEx-MLS framework detects these operations and warns the user about them. Optionally, the framework can automatically remove all implicit (`synchronized` on a method) and explicit (`synchronized` on an object, corresponding to the `monitorenter` and `monitorexit` bytecodes) monitor locking and unlocking operations. Because library code also undergoes the same transactification process as application code and is converted to use STM, libraries are still safe to invoke from concurrent speculative tasks even without their old synchronization code, as JaSPEx-MLS will be enforcing even stronger as-if–sequential semantics for their execution.

In some cases though, there are operations that are dangerous to the parallelization framework because they may cause application code to misbehave, causing divergences from the original sequential application semantics to occur. Consider, for instance, that a library explicitly creates a number of threads to perform some work; or, that an application wants to perform its own class loading; or even that an application relies on reflective introspection, which would reveal many of the bytecode changes performed by JaSPEx-MLS.

Executing these dangerous operations will most likely break the original sequential semantics, due to code ending up being executed outside the control of the framework, and the results from these operations cannot be trusted or even validated as being correct.

Dangerous operations are discovered by matching them with a manually-compiled blacklist. This blacklist is both exhaustive and pessimistic, and it includes:

- The `java.util.concurrent` package, which contains Java concurrency data structures and primitives;
- The `java.lang.reflect` package, which contains Java's reflection (introspection) APIs;
- The class loading features of the `java.lang.Class` class, and any `ClassLoaders`, which contain introspection and class loading;
- All thread-related classes, such as `Thread`, `ThreadLocal` and `InheritedThreadLocal`;
- A number of system classes which provide static user-mutable state (such as `java.lang.System`).

JaSPEx-MLS additionally allows the whitelisting of select parts of these classes and packages — for instance, the `java.lang.reflect.Array` class provides reflective access to arrays, which are unmodified by JaSPEx-MLS and thus are safe to access outside the framework's control. Note that any such accesses are still controlled using `nonTransactionalActionAttempted()`.

The manual creation of the dangerous classes blacklist is facilitated by the fact that Java APIs are organized into packages, allowing a package-by-package analysis, augmented by a careful analysis of all core `java.lang` classes. As stated before, this list is pessimistic, with packages being blacklisted by default. On a case-by-case basis, I additionally whitelisted safe subsets of these APIs which were commonly-used by applications.

When dangerous operations are encountered during bytecode preparation, the JaSPEx-MLS framework issues a warning. Despite there being references in the code to these dangerous operations, they may never actually be called, for reasons similar to those presented in the previous section for non-transactional operations: Their execution may depend on specific program flows being taken, and these flows may in practice never be followed. Thus, these operations are still allowed to be referenced in an application that is being speculatively parallelized by the JaSPEx-MLS framework, as long as they are never invoked.

To enforce that no dangerous operations are ever invoked, the JaSPEx-MLS framework additionally prepends each of these dangerous operations with a call to a special `blacklistedActionAttempted()` method that, when executed, terminates application execution with an error. The special method is prepended to the original dangerous operation instead of replacing it so as to simplify the transformation, but as the special `blacklistedActionAttempted()` will never return the result is similar.

```
private static final class DummyReplacementThread extends Thread {
    // Singleton thread object
    private static final Thread INSTANCE = new DummyReplacementThread();
    private DummyReplacementThread() { ... }
};

// Replacement method for java.lang.Thread.currentThread() that
// always returns the same singleton thread object
public static Thread java_lang_Thread_currentThread() {
    return DummyReplacementThread.INSTANCE;
}
```

Listing 6.12: Example of JaSPEx-MLS's built-in replacements for dangerous operations. The replacement `currentThread()` method always returns the same thread independently of the actual worker thread currently allocated to execute program code.

Note that it is possible — on a case-by-base basis — to provide safe alternatives to some of these dangerous operations, as described in Section 6.1.5.

6.1.5 JDK Method and Class Replacements

To solve the issues with dangerous operations that may lead to deviations from the original sequential application semantics when executed speculatively in parallel — as described in Section 6.1.4 — and also to unlock further parallelism from applications, the JaSPEx-MLS framework includes support for replacing the usage of JDK static methods and even JDK classes in application code: At class preparation time, JaSPEx-MLS can replace references to JDK methods and classes with references to framework-provided alternative versions.

This support for replacing JDK methods/classes is then used to provide a number of built-in replacements for problematic JDK operations, which could otherwise interfere with the correct speculative execution of applications.

For instance, consider the `java.lang.Thread.current()` method: The original version of this method as provided by the JDK returns a reference to the current thread's `Thread` object. On a sequential application, application code executes on the same thread from start to finish, and thus the value returned by `Thread.current()` never changes. When application code is executed by the JaSPEx-MLS framework, it is split onto multiple helper threads, and thus the object returned by the JDK's `Thread.current()` method varies as different parts of the application are executed by different helper threads. To solve this issue, the JaSPEx-MLS framework provides an alternative safe implementation of `Thread.current()`, shown in Listing 6.12. The safe version of `Thread.current()` always returns the same singleton thread object, so that any application that relies on this API can still be successfully (and correctly) executed.

The JaSPEx-MLS framework also provides alternative implementations for:

- `Collections.synchronized[Collection/List/Map/Set/SortedMap/SortedSet]()` methods, which originally returned wrapped collections with all of their methods synchronizing on a single per-collection monitor before executing. On a sequential application, these collections behaved similarly to their non-synchronized counterparts, and thus the framework-provided alternative version of these methods returns non-synchronized collections. During speculative parallelization, whenever these collections are accessed concurrently by multiple concurrent threads, accesses are instead protected by the framework's transactification mechanisms, as described above.

- `Class.forName()` and `ClassLoader.getSystemClassLoader()`, which perform class loading, are redirected to JaSPEx-MLS's class loader, so as to preserve the assumption from Section 6.1 that every non-JDK class is loaded by the framework's class loader and is thus prepared to behave transactionally.

- `System.arraycopy()`, which performs copies between two given array instances. The original method was implemented using native code and was under the control of the JVM and as such could not be modified to use JaSPEx-MLS's STM mechanisms to register transactional reads from the source array and transactional writes to the target array. The alternative version provided by the framework is fully transactional.

- `StringBuilder` class, that is used in string concatenation, again allowing the safe and transactional operation of its internal buffer.

- `ThreadLocal` class, which is used to keep inside a single instance a set of values, with each program thread having access to its own independent copy of the value. Similarly to other operations listed above, it is not uncommon to find `ThreadLocal` instances on thread-safe libraries. In a sequential application, only a single value — corresponding to the single sequential program thread — will exist, and the alternative version of this class simply transforms `ThreadLocal` instances into simple containers for a single value, with this same value being accessible by all program threads.

- `Random` class, which is used to generate random numbers. The alternative version of this class employs transactional seed updates, allowing deterministic behavior even when concurrently accessed by multiple speculative threads.

- `ConcurrentHashMap` class, which is a concurrency-friendly `HashMap` implementation. Similarly to other alternatives above, `ConcurrentHashMap` instances can be found on thread-safe libraries. The alternative version of this class replaces it with the regular non-concurrent Java `HashMap` implementation.

6.2 Spawn and Future Insertion

After going through all the changes described in the previous sections so that it behaves transactionally, application bytecode is ready to be adapted for Method-Level Speculation.

Taking as input the `$transactional` methods produced in the previous steps, the framework now renames them to become `$speculative` methods, to mark that they have been modified for speculative parallelization. This extra step is done separately from the transactification process not only to help with the debug of intermediate stages but also to facilitate the future reuse of code from the JaSPEx-MLS framework. In practice, the `$transactional` versions of methods are never loaded by the JaSPEx-MLS framework into the VM, only their `$speculative` counterparts.

Method-Level Speculation — as introduced in Section 2.4.4 — employs method calls as speculative task spawn points. To prepare an application for MLS, the JaSPEx-MLS framework

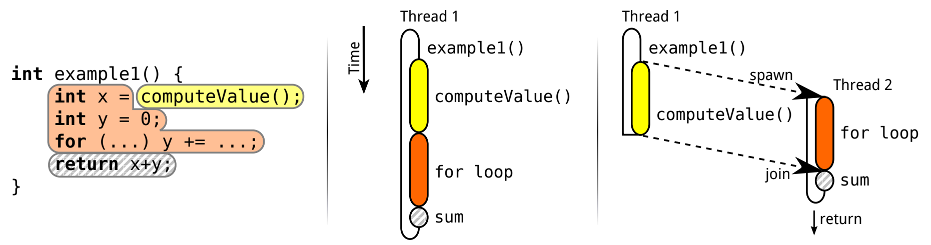


Figure 6.1: Re-reproduction of Figure 2.2: Execution of the `example1()` method (left) when run normally (center) and parallelized with MLS (right). This transformation is profitable only if the `for` loop does enough computation to be worth executing in parallel with the `computeValue()` method.

instruments method call sites so that they can be used to trigger the creation of new speculative tasks.

The first step of the MLS transformation process is the selection of call sites for conversion into spawn points, as described in Section 6.2.1. For this step, the JaSPEx-MLS framework will need to strike a balance between the added overheads and the flexibility to decide at run-time if and when a speculative task should be spawned.

After selection, to convert a given call site on a host method into its MLS-parallelizable version, the JaSPEx-MLS framework performs two changes: (1) converting the method call site into a speculative task spawn point (Section 6.2.2); and (2) adapting the method hosting the spawn point to deal with futures, which are used as placeholders for the pending return value from the replaced method call (Section 6.2.3).

6.2.1 Selecting Call Sites for Conversion

Although JaSPEx-MLS's speculative task spawn operation has been carefully designed to be as lightweight as possible, replacing a method call site with a speculative task spawn point adds non-negligible overhead, especially for frequently-used method call sites. As such, it is important to cull as many non-profitable spawn points as possible.

The selection of call sites for conversion is done by taking into account both global profiling information and local static analysis.

Global profiling information is gathered using JaSPEx-MLS's profiling mode, as described in Section 7.5. The output of the profiling process is an automatically-generated list of methods that the profiler believes are not profitable to speculate on; this list can then be used as part of the call site selection process.

In addition to the automatically-gathered information, knowledge from profiling with third party tools or a programmer's own knowledge about the application may also be supplied as input for the call site selection process.

Local static analysis, in contrast, attempts to decide if there is a non-trivial amount of computation between a task spawn point and its expected synchronization point. For instance, consider the example in Figure 6.1: A new speculative task is created at the beginning of `example1()`, and then before returning, its result `x` is needed. In this example, I have been assuming that the `for` loop between the spawn operation and the end of the method does

```
int exampleOverspec$speculative(int val) {
    int x = computeValue$speculative();
    if (val < 0) return x;
    else return x + val;
}
```

Listing 6.13: Example method that may lead to an unprofitable speculation. It is unprofitable to spawn a new speculative task because there are only a few instructions between the call to `computeValue$speculative()` and the program needing its return value.

```
int exampleOverspec2$speculative(int val) {
    int x = computeValue$speculative();
    int y = 0;
    nonTransactionalActionAttempted();
    ...
}
```

Listing 6.14: Another example method that may lead to an unprofitable speculation. Because there are only a few instructions between the call to `computeValue$speculative()` and a non-transactional operation that would cause a speculative execution to stall until it could become non-speculative again, it is unprofitable in this case to spawn a new speculative task.

enough computation to be worth executing in parallel with the `computeValue()` method — this may not be always the case.

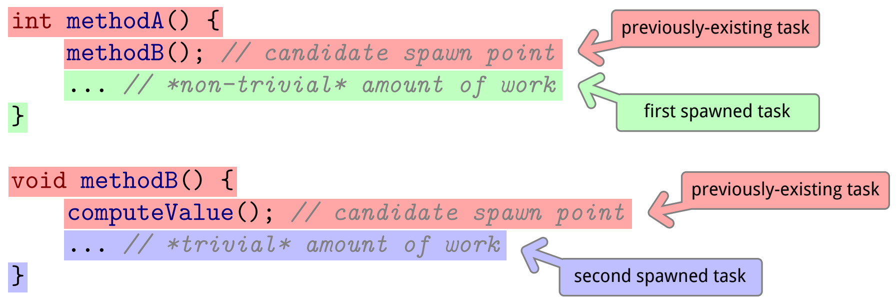
Now consider the example method shown in Listing 6.13: In this case there is clearly not enough computation between the calling of `computeValue()` and the program actually accessing its returned value. Optionally, if Return Value Prediction was used (as described in Section 7.4.5) this issue could be alleviated, but then the profitability of performing speculation would become dependent on the predictor having a high ratio of correct predictions.

Listing 6.14 shows a further problematic example: the existence of a non-transactional operation soon after a method call. In this case, it would be unprofitable to convert the call site into a speculative spawn point, as almost immediately the speculative execution would need to stall until it could safely execute the non-transactional operation.

To avoid the problem of over-eager placement of speculative spawn points — that I refer to as *over-speculation* — the JaSPEx-MLS framework performs a local intra-method analysis in an attempt to determine if there is a non-trivial amount of work performed in the continuation of a candidate method call.

I consider the work performed as non-trivial if between a candidate spawn point and either its synchronization point (an operation that operates on its return value) or stall (due to a non-transactional operation being found) the JaSPEx-MLS framework finds one or more of the following:

- *Method calls.* Because these method calls will execute in parallel with the method call from the spawn point, I consider them as performing a non-trivial amount of work. These method calls may themselves be other candidate spawn points, with the JaSPEx-MLS framework evaluating each one separately.



```
int methodA() {
    methodB(); // candidate spawn point
    ... // *non-trivial* amount of work
}

void methodB() {
    computeValue(); // candidate spawn point
    ... // *trivial* amount of work
}
```

Listing 6.15: Example of possible over-speculation whenever a spawn point without a corresponding synchronization point is hosted inside a `void` method. The previously-existing task is highlighted in red, the first spawned task in green, and the second spawned task in blue. In this example, the second spawned task (that executes the continuation of `computeValue()`) would only perform a trivial amount of work before finishing, and as such its execution would be unprofitable, even though a purely-local analysis would not uncover this issue. JaSPEx-MLS is able to detect and avoid this unprofitable conversion of `computeValue()` into a spawn point.

- *Backwards edges*. These are indicative of one or more loops, that I optimistically consider as performing a non-trivial amount of work.

- *End of the method*. If the method ends without either a synchronization point or a stall being found, this means that the method will return and continue speculatively executing its caller, and I optimistically consider the speculative execution as performing a non-trivial amount of work.

  This case is very common for spawn points created from `void` methods, and as such they are normally very good candidates for the conversion process. Nevertheless, we must be careful to avoid over-speculation when a spawn point is itself placed inside a `void` method, as shown in Listing 6.15: In this example, two candidate spawn points are shown: `methodB()` and `computeValue()`. When considering the conversion of `computeValue()` into a spawn point, and without taking into account that its host method `methodB()` is `void`, this would cause a task to be spawned to execute only a trivial amount of work (in blue). In this case, JaSPEx-MLS would not consider the conversion of the method call to `computeValue()` into a spawn point to be profitable.

The flow analysis algorithm for identifying and avoiding over-speculation works as follows. First, each candidate spawn point is numbered sequentially, and the framework then simulates the execution of the method under analysis by extending the semantic bytecode analyzer present in the ASM framework [Bruneton et al., 2002]: Each return value originating from a candidate spawn point is tagged, and the JaSPEx-MLS framework performs an instruction-by-instruction simulation, at each bytecode instruction computing the expected state of the stack and local variable slots — this includes computing backwards edges and sets of possible types for each stack/local variable position. An example of the tagging and origin tracking for each variable is shown in Listing 6.16.

After the tagging step is completed, the JaSPEx-MLS framework has information on where each value that originated from a candidate spawn point is used, and can proceed to the second

```
void trackingExample() {
    int x = calcX(); // Origin x:calcX()
    int y = calcY(); // Origin x:calcX() y:calcY()
    int z = calcZ(); // Origin x:calcX() y:calcY() z:calcZ()
    ...
    int k = x;  // Origin x:calcX() y:calcY() z:calcZ() k:calcX()
    ...
    return k/y; // Origin x:calcX() y:calcY() z:calcZ() k:calcX()
}
```

Listing 6.16: Output of the process of tagging each variable use with its originating candidate spawn point.

step of the analysis: deciding in a case-by-case basis if the work performed after a candidate spawn point would be trivial or non-trivial, according to the definition given above.

For this second step, the framework analyzes the control flow starting at each candidate spawn point until it finds either that a non-trivial amount of work is performed, or it finds a synchronization/stall point. This step is performed as a single-pass operation (repeated for each candidate spawn point), as the framework simulates the multiple branches that code execution may follow, but never follows backwards edges, as I consider them as indication of a task being non-trivial. In cases where different branches lead to different results (trivial vs non-trivial), the framework currently performs result combination optimistically, and selects a call site for conversion if a task spawned from it is expected to perform enough work on at least one of the branches.

The output of the selection process is a list of call sites to be transformed by the JaSPEx-MLS framework into speculative task spawn points.

6.2.2 Spawn Insertion

Returning to the example from Figure 6.1, let us consider that the call to the `computeValue()` method has been selected for conversion into a speculative task spawn point. To do so, JaSPEx-MLS replaces the call to `computeValue()` with a call to a special framework-internal method: `spawnSpeculation()`. To reify the original method call, the framework makes use of the bytecode-based reflective invocation mechanism described in Section 5.2 to create a `Callable` that is then passed as argument to `spawnSpeculation()`.

A first step of the spawn insertion process for the code from Figure 6.1 is shown in Listing 6.17. The original call to `computeValue()` is converted into a `Callable` object that contains the method arguments needed for the invocation (in this case, only the implicit `this`), which is then passed to the special `spawnSpeculation()` framework method.

Remember from Section 5.2 that the `Callable` object itself is an instance of an automatically-generated class that, when invoked via the `call()` method, proceeds to execute the original method, as shown in Listing 6.18. Every method that is converted by JaSPEx-MLS to use `spawnSpeculation()` has its own corresponding custom `Callable` class.

To simplify further examples, instead of using the very long `spawnSpeculation()` method call shown in Listing 6.17, I instead represent the spawn operation using the conceptual `spawn` keyword. Thus, the spawn operation from the preceding example becomes

```
// First step of conversion for MLS
int example1$speculative() {
    //int x = computeValue$speculative();
    int x =
      ContSpeculationControl.spawnSpeculation(
        new jaspex.speculation.runtime.codegen.
          Codegen$0$Example1.computeValue$speculative(this));
    int y = 0;
    for (...) y += ...;
    return x + y;
}
```

Listing 6.17: First step of the conversion of `example1()` from Figure 6.1 for MLS: The call to `computeValue()` is substituted by a call to `spawnSpeculation()` that receives as argument the reification of the method call created using the bytecode-based reflective invocation mechanism from Section 5.2.

```
// Automatically-generated Callable class for computeValue$speculative()
package jaspex.speculation.runtime.codegen.Codegen$0$Example1;

public final class computeValue$speculative implements Callable {
    private final Example1 arg0; // target instance

    public computeValue$speculative(Example1 arg0) {
        this.arg0 = arg0;
    }

    public Object call() {
        return Integer.valueOf(
          // Original method call
          arg0.computeValue$speculative());
    }
}
```

Listing 6.18: Automatically-generated `Callable` class used in Listing 6.17 to reify the call to `computeValue$speculative()`.

"`int x = spawn computeValue$speculative();`". The semantics remains the same: it is an entry point for the JaSPEx-MLS framework to trigger speculative execution, if it so decides.

6.2.3 Future Insertion

After the transformation of a method call site into a spawn point, a further issue arises: dealing with the return value from the spawn.

Because the JaSPEx-MLS framework executes a method call and its continuation concurrently, the return value from the method call is not made available when the speculative task is spawned — it will only be available later, when the concurrently-executing method finishes its work.

Nevertheless, most of the times the already-existing code is expecting the method call to return a value, which is usually either saved onto a local variable, somewhere on the heap, or kept on the stack for later use.

How can we deal with the absence of a return value?

```java
package java.util.concurrent;

public interface Future<V> {
    // Attempts to cancel execution of this task
    boolean cancel(boolean mayInterruptIfRunning);

    // Returns true if this task was canceled before it completed normally
    boolean isCancelled();

    // Returns true if this task completed
    boolean isDone();

    // Waits if necessary for the computation to complete, and then retrieves its result
    V get();
}
```

Listing 6.19: Java API for the `java.util.concurrent.Future` interface.

As alluded to before, a possible solution for this issue is to predict the return value using a technique called Return Value Prediction (Section 7.4.5). But, although useful in some cases, this technique is not a panacea, especially for object-oriented applications, because, for instance, it is hard for a predictor to simulate a method that always returns a different or even a new instance of a given domain object.

A more flexible solution employed by the JaSPEx-MLS framework is to have the `spawn` operation return not a prediction nor a dummy value but instead a placeholder for the computed value, in the form of a future.

The Java API for the `Future` interface is shown in Listing 6.19. The `cancel()`/`isCancelled()` methods allow canceling the execution of a task, `isDone()` allows checking its state, and most importantly, the `get()` operation allows retrieving its result — by waiting for the result to become available, if necessary.

An example of the conversion of the code from Figure 6.1 to save the return value from `computeValue()` into a future is shown in Listing 6.20. This conversion involves a change in type from an `int` into a `Future<Integer>` for the local variable `x`, and also changing whenever `x` is read to instead call the `get()` method from the future.

Although the conversion of a method invocation into a spawn point described in the previous section is straightforward, dealing with the returned future from the `spawn` operation may require a number of transformations to the method. Consider again, for instance, the `example1()` method from Listing 6.20: In this case, `int x` needed to be replaced with `Future<Integer> x` in line 11; and in line 14 the read of `x` was changed to `x.get()`.

The main objective of the transformation to use futures is to delay to as late as possible the retrieval of the result from the future, allowing concurrent execution to proceed for as long as possible without it, avoiding the need to heavily depend on value prediction.

Depending on its type and how the return value from the invoked method is used, the JaSPEx-MLS framework has to handle future insertion differently. Over the next sections I describe how the framework deals with these different usages.

```
// Original code
int example1() {
    int x = computeValue();
    int y = 0;
    for (...) y += ...;
    return x+y;
}

// Conversion for MLS
int example1$speculative() {
    Future<Integer> x = spawn computeValue$speculative(); // call to spawnSpeculation()
    int y = 0;
    for (...) y += ...;
    return x.get() + y;
}
```

Listing 6.20: Conversion of the invocation of `computeValue()` into a speculative task spawn point. The `spawn` operation returns a future, and the example illustrates the needed changes to the type of and access to variable `x`.

6.2.3.1 Return value is `void` or not needed

The simplest case happens whenever the target of the `spawn` operation is a `void` method, or a value is returned but never actually assigned. In this case, no further modifications need to be made to the method other than adding the `spawn` call and discarding the future returned by the `spawn` operation.

6.2.3.2 Return value is consumed immediately

On some call sites, the result from the invoked method is immediately used without being explicitly saved to the heap or onto local variables. Consider, for instance that the previous example method `example1()` from Figure 6.1 is modified as shown in Listing 6.21 to create `example2()`. In this case, by default, JaSPEx-MLS would not choose `computeValue()` for conversion into a spawn point — it would not try to spawn a new speculative task, as its result would be immediately needed for continuing the execution of `example2()`, as described in Section 6.2.1.

A `get()` that immediately follows a `spawn` makes the `spawn` useless, only adding overhead. The exception enabling this case happens when, optionally, return value prediction is enabled, as in this case the `get()` operation can immediately return a prediction of the expected return value, allowing the executions of `example2()` (and its speculative return) and `computeValue()` to still proceed concurrently.

6.2.3.3 Heap Write: Return value is written onto a field or an array position

An example of a return value being immediately written onto the heap is shown in line 15 from Listing 6.22, where a naïve conversion for MLS causes the `spawn` operation to be immediately followed by a `get()`.

At first sight this case could be considered to be a subset of the previous one, as the value is immediately consumed as part of its transfer from the stack and onto the heap, precluding any speedup. The difference in this case is that the written value is not needed for the rest of the code to continue executing, and JaSPEx-MLS is able to take advantage of this: Any write

```java
// Modified version of example1()
int example2() {
    int y = 0;
    for (...) y += ...;
    return computeValue() + y;
}

// Conversion for MLS
int example2$speculative() {
    int y = 0;
    for (...) y += ...;
    Future<Integer> temp = spawn computeValue$speculative();
    return temp.get() + y;
}
```

Listing 6.21: Modified version of Listing 6.20 where `computeValue()` is called at the end of the method, instead of explicitly saved onto variable `x`, and its corresponding MLS conversion. Note that this MLS conversion only makes sense when supported by return value prediction, otherwise it is skipped by JaSPEx-MLS as it would only add overhead for no added gain.

```java
1   public int fieldx;
2
3   // Another modified version of example1()
4   int example3() {
5       fieldx = computeValue();
6       int y = 0;
7       for (...) y += ...;
8       return fieldx + y;
9   }
10
11  // Unsuccessful naïve conversion for MLS
12  int example3$speculative() {
13      Future<Integer> temp_fieldx = spawn computeValue$speculative();
14      // Retrieve value from Future temp_fieldx, and store it in fieldx
15      Transaction.storeInt(this, temp_fieldx.get(), $offset_fieldx);
16      int y = 0;
17      for (...) y += ...;
18      return Transaction.loadInt(this, fieldx, $offset_fieldx) + y;
19  }
```

Listing 6.22: Another modified version of `example1()` where the value from `computeValue()` is saved onto an object field, and its unsuccessful naïve MLS conversion, where `get()` is called immediately after spawning a new task in line 15.

operation to memory location $m1$ can be delayed until immediately before a read operation that accesses the same $m1$, as by definition, no other operation is able to observe whether $m1$ contains the write or not. Thus, the JaSPEx-MLS framework is free to reorder and delay a write until its matching read operation needs to be executed.

To allow this reordering to occur, JaSPEx-MLS's STM model was designed to allow the registration of intents to write in place of the concrete return values — allowing JaSPEx-MLS to parallelize methods that would normally not be able to be parallelized by other MLS frameworks. This means that when using this STM model, futures can be written onto heap memory locations (fields or array positions) as temporary placeholders for the original values. Listing 6.23 shows

```java
// Future instance stores
public static void storeFutureObject(Object instance, Future<Object> future, long offset);
public static void storeFutureInt(Object instance, Future<Integer> future, long offset);
//... variants for the remaining types

// Future static stores
public static void storeFutureObject(Future<Object> future, StaticFieldBase sfb, long offset);
public static void storeFutureInt(Future<Integer> future, StaticFieldBase sfb, long offset);
//... variants for the remaining types

// Future array stores
public static void arrayStoreFutureObject(Object[] array, int pos, Future<Object> future);
public static void arrayStoreFutureInt(int[] array, int pos, Future<Integer> future);
//... variants for the remaining types
```

Listing 6.23: Extended STM API, allowing the writing of futures to fields and arrays. This API complements the STM API previously presented in Listing 6.3 and Listing 6.4, and similarly has versions for each native type and for `Object`. Because the STM transparently handles futures (Section 7.4.4), there is no need to have the corresponding counterpart methods for reading futures stored on fields/arrays.

```java
1  // Conversion for MLS with STM support for Futures
2  int example3$speculative() {
3      Future<Integer> temp_fieldx = spawn computeValue$speculative();
4      // Store the Future in temp_fieldx directly into fieldx
5      Transaction.storeFutureInt(this, temp_fieldx, $offset_fieldx);
6      int y = 0;
7      for (...) y += ...;
8      return Transaction.loadInt(this, fieldx, $offset_fieldx) + y;
9  }
```

Listing 6.24: Successful conversion for MLS of `example3()`, using STM support for futures. Note that, in contrast to the naïve conversion from Listing 6.22, the future itself is now directly stored on `fieldx`, dispensing with the need to call the `get()` method to obtain its concrete value.

the extended versions of methods presented in Listing 6.3 and Listing 6.4, which are used to write futures to fields and arrays.

Using the extended STM API for futures, we can modify the naïve conversion of the `example3()` method shown in Listing 6.22 into a version where STM support for futures is used, shown in Listing 6.24. Although the code is similar to the previous naïve conversion, the naïve version precluded the concurrent execution of `example3()` and `computeValue()` — as it would immediately stall waiting for the `get()` operation to finish — whereas in the new version both methods successfully execute in parallel, as the return value is not needed for the field write to succeed and for the `example3()` method to continue its execution. Note also that the read of `fieldx` at the end of the method is unchanged in both examples (line 18 from Listing 6.22, line 8 from Listing 6.24), as the same API is used whether `fieldx` contains a value or a future — JaSPEx-MLS's STM transparently handles these cases and returns the correct value, by internally calling the `get()` operation and waiting for a future to finish computing if needed.

6.2.3.4 Stack Write: Return value is kept on the stack or is saved onto a local variable

This is the case featured in the original version of `example1()` shown in Listing 6.20.

```
1       int example4$speculative() {
2   ●       int x; // x is an int
3   ●       if (...) {
4     ○         Future<Integer> x =
5     ○           spawn computeValue$speculative(); // x is has now been redefined to be a future
6   ●       } else {
7   ●           x = 10; // x is an int
8   ●       }
9     ⊗     int y = 0;
10    ⊗     for (...) y += ...;
11    ⊗     x = x + y; // error: is x an int or a future?
12    ●     return x; // x is an int
13      }
```

Listing 6.25: Example for the problematic replacement of a returned value with a future. Execution may reach line 9 via lines 4 and 5, in which case x is a `Future<Integer>` or via line 7 in which case x is still an `int`, resulting in an error. ● is used to tag lines where x is an `int`, ○ where x is a `Future<Integer>`, ⊗ those where there is an error, and lines are used to show the two possible control flows through the method. Due to these ambiguity issues, the JVM's bytecode verifier would refuse to load this code.

As the JVM bytecode specification allows any data type to be stored in any local variable, and as this type can change during execution of a method (as introduced in Section 4.2.1 and Listing 4.3), JaSPEx-MLS takes advantage of this to substitute the future for the original value in the same local variable, changing its type.

The same happens with values that are kept on the stack — although in this case some bytecode adjustments may be needed to account for the difference in word size between object references which use a single word (such as the future object) and long/double native types which occupy two stack words, as there are bytecodes with variants for both single and double words (such as `dup`/`dup2`, `pop`/`pop2`, ...). Otherwise, an operation such as `pop2` that is meant to remove a `long` (two words) off the top of the stack would remove both its replacement future (only one word) and an additional unrelated word below it. Similarly to previous erroneous examples, the JVM's bytecode verifier is able to detect and reject code with such issues.

Because variables now contain futures, any access to them must be prepended with a call to the contained future's `get()` method so as to retrieve its concrete value. Note that this modification is non-trivial: Unlike in the previous cases where the modifications were localized — discarding values off the stack, immediately invoking `get()` or changing STM writes into STM future writes — this modification entails tracking every read of a local variable (or stack slot) containing a future and adding a call to `get()` to obtain the concrete value from the future immediately before it is used.

Unfortunately, this substitution may lead to issues when multiple branches lead to different types being possible for the same variable, as exemplified in Listing 6.25. In the presented example, depending on the run-time flow through the method, when execution arrives at line 11 the type for variable x may either be an `int` or a `Future`. Adding a call to `get()` would cause an error for the branch where x still contains an `int`, whereas not adding it would cause an error for the branch where x contains a future. Because of this, the JVM verifier would reject this code, and further modifications are needed to solve the issue.

```
1  int example4$speculative() {
2      int x; // x is an int
3      if (...) {
4          Future<Integer> x =
5            spawn computeValue$speculative(); // x is has now been redefined to be a future
6          goto x_is_a_future;
7      } else {
8          x = 10; // x is an int
9      }
10     int y = 0;
11     for (...) y += ...;
12     x = x + y; // x is an int
13   rest_of_the_method:
14     return x; // x is an int
15   x_is_a_future:
16     int y = 0;
17     for (...) y += ...;
18     x = x.get() + y; // x is a future
19     goto rest_of_the_method;
20 }
```

Listing 6.26: Fixed version of `example4()`, after applying semantic analysis and code duplication to solve the type inconsistency for variable `x` caused by the injection of the spawn point. The code in lines 10 to 12 is duplicated onto lines 16 to 18, so that in line 12 the sum is performed with variable `x` containing an `int` and on line 18 with variable `x` containing a future. ● is used to tag lines where `x` is an `int`, ○ where `x` is a `Future<Integer>`, and lines are used to show the two possible control flows through the method. Unlike the previous Listing 6.25, there is never an inconsistency in types, and no lines are tagged with ⊗.

To solve this issue, my framework again relies on a semantic bytecode analyzer (adapted from [Bruneton et al., 2002]). As a first step, JaSPEx-MLS must identify regions of the method under analysis where a variable's type is inconsistent — where the type may either be the original type or a future depending on the control flow followed through the method.

To identify problematic regions JaSPEx-MLS simulates the state of the stack and the local variables, keeping for each position a set of the expected types for that position. Then, it scans for problematic instructions — instructions that attempt to operate on a value where the set of possible types for that value is bigger than one. In the case of the example from Listing 6.25, this would correspond to line 11, where an `iadd` (addition of two `int`s) instruction will attempt to produce the sum of `x` and `y`.

Afterwards, JaSPEx-MLS attempts to identify the start of the problematic region, by scanning backwards from the problematic instruction, including going back through branches, and taking loops into account.

In the presented example from Listing 6.25 line 9 is the first of the problematic region. Thus, we have identified the problematic region: from lines 9 to 11. After line 11 there are no type issues as `x` becomes an `int` again after the add operation is retired, otherwise the framework would consider the end of the current basic block as being the tentative end of the problematic region (further iterations would later detect and duplicate more code blocks if needed).

To fix the problem, the JaSPEx-MLS framework then duplicates the problematic region, and changes the method, diverting one of the branches that created the issue by modifying it to jump to the newly-added duplicate of the problematic region. The duplication and jumps needed to fix the example method from Listing 6.25 are shown in Listing 6.26.

The fixing process iterates over each problematic instruction (if there are multiple) and may have to be repeated several times if several `spawn` operations are involved. As a safeguard, if too much code gets duplicated the JaSPEx-MLS framework undoes the placement of the `spawn` and the subsequent modifications, as the VM will not optimize overly large methods, and falling back to the interpreter will negate most performance gains from parallelization.

The fixing algorithm also supports forwards/backwards branches and exception blocks. After transformation, the resulting bytecode has no inconsistent types and is accepted by the VM's bytecode verifier.

The above changes allow futures to be saved both to local variables and to heap memory locations, allowing the JaSPEx-MLS framework to unlock further parallelism in applications.

6.3 Summary

In this chapter, I cover the bytecode modifications performed by the JaSPEx-MLS framework to prepare applications for MLS parallelization.

I start by describing the process of transactification — modifying application code to execute with transactional semantics — including the interception of heap operations, and dealing with both non-transactional and dangerous operations.

I then describe the task selection and insertion process. First, JaSPEx-MLS performs local static analysis to identify profitable methods for parallelization. Then, the selected call sites are modified into speculative spawn points. Finally, the method is modified to allow the future returned by the spawn point to be used in place of the original return value.

Chapter 7

Application Execution

After the previously-modified bytecode is loaded into the Java Virtual Machine, JaSPEx-MLS invokes the target application's entry point. Application bytecode then starts executing in non-speculative mode until the first spawn point is reached, triggering the creation of the first speculative task.

In this chapter, I discuss the JaSPEx-MLS framework's run-time internals. I examine speculative task management in Section 7.1, introduce two novel techniques for improving thread pool usage efficiency in Section 7.2 and Section 7.3, delve into my custom STM model in Section 7.4, and describe, in Section 7.5, the framework's profiling mode and analysis tools.

7.1 A Program as a Set of Tasks

For the run-time part of the JaSPEx-MLS framework, a program is comprised of a set of tasks. At any given time, most of these tasks are speculative, and one is not: the task that is executing code in program-order — in the original sequential application order.

In fact, the normal execution of a sequential application can be seen as the execution of a single giant task that spans the entire program, never spawns other tasks, and is always executing non-speculatively in program-order.

Under the chosen speculation model, tasks (speculative or not) being executed by JaSPEx-MLS may create further tasks — as part of the `spawn` operation. Speculative tasks may be successful and *commit*; or they may fail or no longer be needed (for a number of reasons) and *abort*.

The first task in the system is the bootstrap task. When JaSPEx-MLS starts up, similarly to the JVM itself, it either receives as argument the target application's main class or discovers that information as part of the metadata on a JAR file. After the framework is initialized, a special bootstrap task is created, marked as being in program-order, and submitted for execution to JaSPEx-MLS's thread pool.

After a thread from the pool picks up the bootstrap task for execution, the task's code in turn invokes the target application's entry point — the framework-prepared `main$speculative()` method — and application code starts its framework-instrumented execution.

Over the next sections, I shall describe the various parts of the life-cycle of a speculative task: how and when a task is created (Section 7.1.1), how it is executed (Section 7.1.2), and how it is

```
1 int example1$speculative() {
2     Future<Integer> x = // conceptually: spawn computeValue$speculative();
3       ContSpeculationControl.spawnSpeculation(
4         new jaspex.speculation.runtime.codegen.
5           Codegen$0$Example1.computeValue$speculative(this));
6     int y = 0;
7     for (...) y += ...;
8     return x.get() + y;
9 }
```

Listing 7.1: MLS-ready version of the `example1()` method from Figure 2.2.

completed (Section 7.1.3). The entire life-cycle of a task is then summarized in Section 7.1.4. As a running example throughout the next sections I shall use the previously-presented `example1()` method from Figure 2.2, shown converted for MLS in Listing 7.1.

7.1.1 Spawning a New Speculative Task

When program execution hits one of the previously-inserted spawn points, the `spawn` operation gets triggered, resulting in a call to `spawnSpeculation()` (as described in Section 6.2.2) that receives as an argument a `Callable` instance representing the reification of the call site that the spawn point replaced.

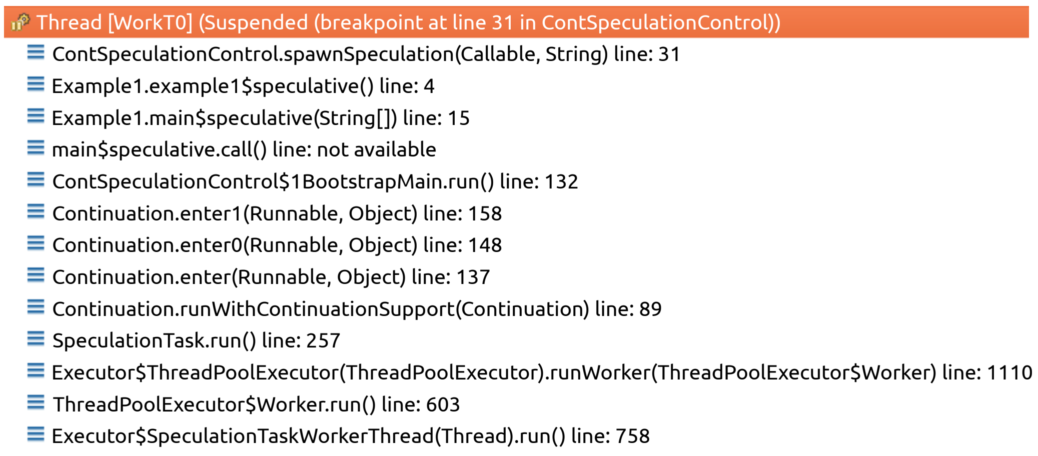
For instance, when executing the example from Listing 7.1, the spawn point is reached (in line 2) and `spawnSpeculation()` is called, receiving as argument the newly-created `Callable` representing the `computeValue$speculative()` method.

At this point, and inside `spawnSpeculation()`, the framework first checks the state of the thread pool, to avoid the overhead of submitting work to a full thread pool that will reject it. This state check is approximate, as other threads in the pool may be picking up or retiring tasks concurrently, and works by either checking a counter of busy threads that is atomically updated by each thread as it picks up and retires tasks when the thread pool is in the direct hand-off mode, or by checking to see if the buffer is full in buffered mode (thread pool modes are described in more detail in Section 7.2). Because this check adds overhead to each spawn operation, the chosen approximate designs trade off accuracy with communication overhead, as either check is a read-only operation that does not interfere with the other threads' works.

If the thread pool is detected to be full, the JaSPEx-MLS framework *early rejects* the creation of a new speculative task. If, instead, the framework decides that it should go ahead with spawning a new speculative task, it first creates a new speculative task object — an instance of `SpeculationTask`.

In the current arrangement, the current task (the task that triggered the `spawn` operation) will execute the `Callable`, whereas the new task being created will execute the code that follows the return of the `spawn` operation. In the case of the running example, this will mean that the current task (and its hosting thread) will go on to execute `computeValue$speculative()` whereas the new task will execute the rest of the body of `example1$speculative()`, from lines 6-8.

Now consider the state of the stack when `spawnSpeculation()` is called from inside `example1$speculative()`, as shown in Figure 7.1. As we can see, the stack contains all application methods from `main$speculative()` upwards, with framework and Java-internal



```
Thread [WorkT0] (Suspended (breakpoint at line 31 in ContSpeculationControl))
    ContSpeculationControl.spawnSpeculation(Callable, String) line: 31
    Example1.example1$speculative() line: 4
    Example1.main$speculative(String[]) line: 15
    main$speculative.call() line: not available
    ContSpeculationControl$1BootstrapMain.run() line: 132
    Continuation.enter1(Runnable, Object) line: 158
    Continuation.enter0(Runnable, Object) line: 148
    Continuation.enter(Runnable, Object) line: 137
    Continuation.runWithContinuationSupport(Continuation) line: 89
    SpeculationTask.run() line: 257
    Executor$ThreadPoolExecutor(ThreadPoolExecutor).runWorker(ThreadPoolExecutor$Worker) line: 1110
    ThreadPoolExecutor$Worker.run() line: 603
    Executor$SpeculationTaskWorkerThread(Thread).run() line: 758
```

Figure 7.1: Example stack trace for the execution of `spawnSpeculation()` from inside the `example1$speculative()` method from Listing 7.1.

methods below it. This stack state will now become the new task's starting state, and will need to be transferred onto the new task's host thread.

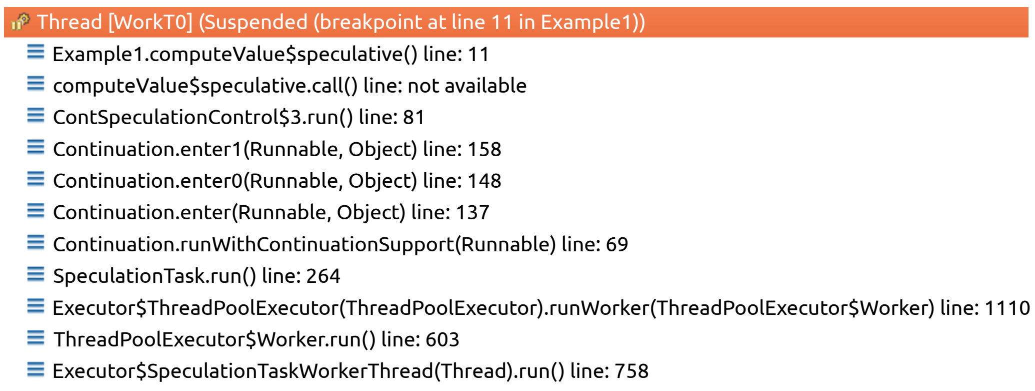
To transfer the current thread's state onto another thread, the JaSPEx-MLS framework leverages on the support for first-class continuations previously introduced in Chapter 5: The framework captures a continuation containing the current state of the thread, and then attaches it to the newly-created `SpeculationTask` object. Afterwards, the `SpeculationTask` is submitted to the thread pool.

Between the earlier check of the thread pool state and the submission of the newly-created `SpeculationTask` for execution, the state of the pool may change. Thus, the submitted task can still, after submission, be rejected by the pool, due to it being full. I call such cases *late rejects*, in contrast to the earlier ones.

Whenever the creation of a new task is rejected, the `Callable` received as argument by `spawnSpeculation()` is instead directly executed by the current thread, similarly to the original, sequential application. After the method inside the `Callable` method finishes executing, the `spawnSpeculation()` method returns its resulting value and execution of the code after the `spawn` continues. If the spawn operation had been successful, the code after the `spawn` would instead be executed by a new task; because it was not, the code is executed as part of the current task. Because, as we saw in Section 6.2, application code has been modified to receive a future as a return value from `spawnSpeculation()` instead of the original return value, the JaSPEx-MLS framework wraps the return value from the direct execution in a immediate future object that can then be returned by `spawnSpeculation()`. This immediate future contains only a field for storing the return value, with the `get()` operation just returning the stored value.

The cost of an early task reject is comparatively small: it entails only the creation of the `Callable`, the invocation of `spawnSpeculation()`, the thread pool state check, and the creation of the immediate future object. Late task rejects, on the other hand, additionally include the creation of the `SpeculationTask` object, the continuation capture operation, and the submission of the task to the thread pool. For this reason, late rejects should be avoided as much as possible, and I shall later discuss the impact of thread pool buffering on late rejects in Section 7.2.1.

The newly-created `SpeculationTask` object implicitly forms the link between the parent task



```
Thread [WorkT0] (Suspended (breakpoint at line 11 in Example1))
    Example1.computeValue$speculative() line: 11
    computeValue$speculative.call() line: not available
    ContSpeculationControl$3.run() line: 81
    Continuation.enter1(Runnable, Object) line: 158
    Continuation.enter0(Runnable, Object) line: 148
    Continuation.enter(Runnable, Object) line: 137
    Continuation.runWithContinuationSupport(Runnable) line: 69
    SpeculationTask.run() line: 264
    Executor$ThreadPoolExecutor(ThreadPoolExecutor).runWorker(ThreadPoolExecutor$Worker) line: 1110
    ThreadPoolExecutor$Worker.run() line: 603
    Executor$SpeculationTaskWorkerThread(Thread).run() line: 758
```

Figure 7.2: Example stack trace for the execution of `computeValue$speculative()`, after a successful submission of a new task to the thread pool, showing the newly-cleared thread stack. In contrast with Figure 7.1, the previously-pending methods (above `Continuation.enter1()`) are gone, replaced with only `computeValue$speculative()` (top of stack) and its `Callable` (below top).

(the one that entered the `spawn` operation) and the newly-created child task. It also implements the `Future` interface, and is used as the future that is returned by the spawn operation, serving as a placeholder for the real result from the `spawn`-replaced method (`computeValue$speculative()`, in the running example).

After the new speculative task is successfully submitted to the thread pool, the parent task can proceed to execute the `Callable` it received when the `spawn` was triggered. Because the existing thread state has been attached to the newly-created speculative task, the current thread has no longer any need for this state — it will not be used again. As this stack state is not needed, the framework first wipes the thread's stack clean by resuming an empty continuation, so that the VM clears the stack and replaces it with a previously-created empty continuation. Afterwards, the thread proceeds to execute the `Callable` — `computeValue$speculative()`, in the running example. As an example, Figure 7.2 shows the state of the parent thread's stack from Figure 7.1 after the clear operation: Notice that the previously-pending methods from Figure 7.1 (above `Continuation.enter1()`) are gone, replaced with only `computeValue$speculative()` at the top of the stack with its corresponding `Callable` immediately below. This clear operation has two main objectives: (1) it shortens the height of the stack, so that further `spawn`s will have less state to migrate between threads, and (2) it facilitates debugging, as the smaller stack is easier to analyze, and there are no duplicate segments of the client application's stack on different active threads.[1]

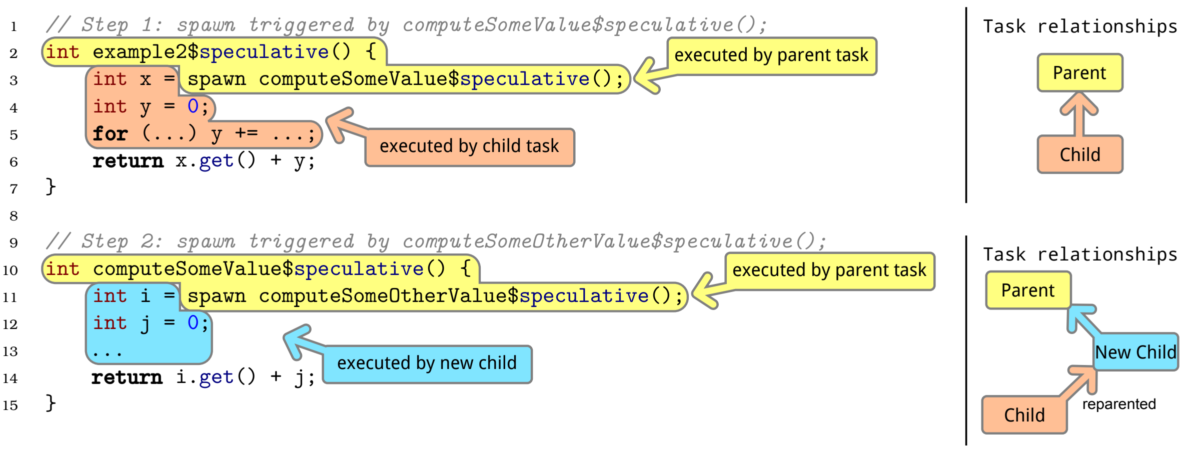
With this final step, the `spawn` is complete, and the parent task proceeds to execute the code inside `computeValue$speculative()`. The implementation of the `spawnSpeculation()` operation is shown in Listing 7.2.

Note that because the JaSPEx-MLS framework supports nested speculation, the parent task may itself be speculative, and have its own parent, so the execution of `computeValue$speculative()` may be the result of an earlier `spawn` operation.

[1]This means that, for instance, at any given time, only one thread will contain the application entry point and the bootstrap task in its stack.

```
 1 public static Future<?> spawnSpeculation(
 2   final Callable continueExecution, String source) throws Exception {
 3     // Check the state of the thread pool
 4     if (!Executor.hasFreeThreads()) {
 5         // early reject task
 6         return new NoSpeculationFuture(continueExecution.call());
 7     }
 8 
 9     SpeculationTask specTask = new SpeculationTask(continueExecution, source);
10 
11     // Save the current childTask, which will become the new task's child
12     // (and the current task's grandchild)
13     SpeculationTask grandchildTask = SpeculationTask.getChildTask();
14 
15     Continuation continuation = captureContinuation();
16     if (!continuation.isResumed()) { // We are still in the parent
17         specTask._taskRunnable = continuation;
18 
19         if (!Executor.tryExecute(specTask)) {
20             // late reject task
21             return new NoSpeculationFuture(continueExecution.call());
22         }
23 
24         // Register the new task as the current task's child
25         SpeculationTask.setChildTask(specTask);
26 
27         final SpeculationTask childTask = specTask;
28         cleanStackAndContinueWith(new Runnable() {
29             public void run() {
30                 ExecutionResult result = null;
31                 try {
32                     result = ExecutionResult.newObjectResult(continueExecution.call());
33                 } catch (SpeculationException e) { terminate(e, false); }
34                   catch (VirtualMachineError e)  { terminate(e, false); }
35                   catch (AssertionError e)       { terminate(e, false); }
36                   catch (Throwable t) {
37                     result = ExecutionResult.newThrowableResult(t);
38                 }
39                 try {
40                     childTask.setResult(result);
41                 } catch (Throwable t) { terminate(t, false); }
42             }
43         });
44         throw new AssertionError("Should never happen");
45     } else { // The child task has resumed the continuation
46         SpeculationTask.inheritChildTask(grandchildTask); // Save the inherited child
47         SpeculationTask.checkThrowable();
48         return specTask;
49     }
50 }
```

Listing 7.2: Implementation of the `spawnSpeculation()` operation. Note that due to the use of the VM continuations, the method never returns in the parent thread after the `cleanStackAndContinueWith()` method executes — in that case it is the child task that will resume the method's execution and execute lines 46-48, eventually returning.



```
// Step 1: spawn triggered by computeSomeValue$speculative();
int example2$speculative() {
    int x = spawn computeSomeValue$speculative();
    int y = 0;
    for (...) y += ...;
    return x.get() + y;
}

// Step 2: spawn triggered by computeSomeOtherValue$speculative();
int computeSomeValue$speculative() {
    int i = spawn computeSomeOtherValue$speculative();
    int j = 0;
    ...
    return i.get() + j;
}
```

Figure 7.3: Example illustrating the dynamic parenting scheme employed by JaSPEx-MLS. In the first step, inside `example2$speculative()`, a speculative task is spawned, leading to the original Parent ← Child relationship. Afterwards, inside `computeSomeValue$speculative()`, another speculative task is spawned. Because this task, in the original program order, is ordered after the parent and before the existing child, the task relationships are changed, with the new child becoming the new parent for the existing child, leading to the new Parent ← New Child ← Child relationship.

7.1.1.1 Task Parenting Relationships

An important aspect of the parent/child relationship between tasks is that this relationship is dynamic and may change during a task's execution.

A change in the parenting relationship happens whenever a task's parent spawns other speculative task: This new speculative task will be executing code that comes before (in the original sequential program) the current child task, and as such it becomes its new parent. This switch can be seen in lines 13, 25 and 46 from Listing 7.2.

An example of this dynamic parenting relationship is shown in Figure 7.3. In step 1, a new task is spawned to execute the continuation of `computeSomeValue$speculative()`, resulting in the Parent ← Child relationship. Afterwards, in step 2, a further spawn of `computeSomeOtherValue$speculative()` creates a new child that, in the original program order, would execute before the older child, and so the parenting relationship is updated to reflect that, becoming Parent ← New Child ← Child.

JaSPEx-MLS supports both *out-of-order* and *in-order* speculative task spawn:

- *Out-of-order* spawning allows the same parent to create several child tasks, and its name comes from the observation that newer tasks are created in an order that does not follow the original sequential program order. It is precisely the out-of-order model that leads to the dynamic parenting exemplified in Figure 7.3, where during the lifetime of a task its parent task can change.

- *In-order* spawn happens when a speculative child task itself spawns a further child task, and that task repeats the same process: new tasks are created, but as their creation order follows from the code execution order in the original sequential program — the existing

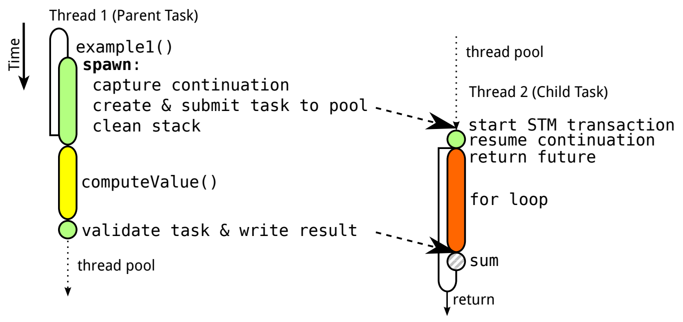


Figure 7.4: Execution of `example1$speculative()` from Figure 2.2: The parent task creates and submits a new speculative task and goes on to execute `computeValue()`, whereas the new speculative task executes the `for` loop.

child itself becomes the parent of the new child — task parenting relationship remains unaltered after a task is first created.

The combination of both spawn models creates a very flexible model, that is better able to extract parallelism from applications, but also poses some challenges to task execution ordering, as I shall discuss in Section 7.2.

7.1.2 Executing a New Speculative Task

When an idle thread from JaSPEx-MLS's thread pool picks up a new speculative task for execution, it first checks to see if this new task should be executed in program-order (all of its predecessors have already finished) or speculatively (at least some of its predecessors are still active). If executing speculatively, the thread invokes the STM API to start a new STM transaction to protect its heap memory accesses.

Afterwards, the thread resumes the continuation containing the state received from the parent task. After the continuation is resumed, execution jumps back inside the `spawnSpeculation()` method. In the case of the running example from Listing 7.1, the stack seen in Figure 7.1 is restored, but now on a *different* thread.

Inside `spawnSpeculation()`, the method uses the `Continuation.isResumed()` API (Listing 5.1) to discover that it is now running on a new speculative task. The method then returns the `SpeculationTask` object (which also implements the `Future` interface) — the spawn process for the child thread is complete, and application code execution is resumed.

In the case of the running example from Listing 7.1, execution is resumed at line 6 inside `example1$speculative()`, and the child proceeds to execute the loop. The entire `spawn` process for the example is illustrated in Figure 7.4: the parent task reaches the `spawn` instruction, creates a new speculative task and goes on to execute `computeValue$speculative()`. The newly-created child task resumes the continuation and then continues executing the `for` loop.

7.1.3 Completing and Committing a Speculative Task

The validation and posterior commit/abort of a speculative task may be triggered by one of several conditions:

- The speculative task finished its own work. For instance, when reaching the end of the `computeValue$speculative()` method from the running example in Listing 7.1, or when execution terminates with an exception.
- The speculative task attempted to execute a non-transactional operation, and the `nonTransactionalActionAttempted()` method was triggered.
- The STM detected that the current transaction was doomed to abort, and signaled the task to do so.
- The parent of the current speculative task has finished its own execution, and signaled its child that it could now commit its own work, as part of the *early commit* feature described in Section 7.4.3.

A speculative task may only validate its work and attempt to commit if it is the oldest-running task in the system — that is, if all of its predecessors, including its parent, have finished their own execution and committed successfully.

In my design, every task has a parent that spawned it, and that parent is responsible for writing its result onto the child's `SpeculationTask` object. Before writing its result, a parent task validates and commits its own transaction; thus, when a child receives the result from its parent, it knows that its parent has finished its own work.

Whenever a task is ready to begin validation, but no result from its parent is available, it must wait for this value to become available. Note that it may be possible for the parent itself to be in the same situation, and for a sequence of speculative tasks to all be waiting for their own parents. After writing its result onto the child's `SpeculationTask`, the parent task wakes up the child's thread (if applicable), so that it can resume its work.

The implementation of the task commit operation is shown in Listing 7.3. After receiving an object as a result from its parent, a task may attempt to validate itself and commit (Section 7.1.3.1); when, instead it receives an order to abort it must discard its state and return to the thread pool (Section 7.1.3.2); and whenever the result from the parent is an exception the task must discard its state and re-execute (Section 7.1.3.3).

7.1.3.1 Validation and Commit

When the current task's parent has successfully finished its validation and produced a proper return value, its child speculative task may attempt to validate its own STM transaction.

Whenever validation fails (or the transaction has been deemed to be doomed), the transaction must be aborted, and the task is re-executed by re-resuming the continuation received from the parent. A doomed transaction is one where the STM has already determined that validation will fail (as described in Section 7.4.2), and as such the framework knows in advance not to perform validation. For the re-execution, no new STM transaction is started, as the task will be running in program-order, rather than speculatively.

If validation is successful, the task commits its work. Afterwards, either the task continues working in non-speculative mode, or, if it had already finished its work, the task is marked as being finished, its result is written onto its own child's `SpeculationTask`, and the thread hosting the task is returned to the thread pool.

```java
public static void tryCommit(boolean doomed) {
    if (!Transaction.isInTransaction()) return;

    SpeculationTask current = current();

    // Wait for result from our parent
    current.waitForResult();

    // If we get a proper value as a result
    if (!doomed && current._result.isObject()) {
        // Attempt to validate and commit
        if (Transaction.commit()) return;
    }

    // Either the transaction was doomed, the validation failed,
    // or we got an exception as a result
    Transaction.abort();

    if (!current._childInherited) {
        // Abort child task (if any) only if it was not inherited from the parent
        abortChildTask();
    }

    if (current._result == ExecutionResult.ABORT_SPECULATION) {
        // We got an abort order, return the worker directly to the thread pool
        ContSpeculationControl.returnWorkerToPool();
    } else {
        // Re-resume continuation and re-execute task
        ContSpeculationControl.resumeContinuation(current._taskRunnable);
    }
    throw new AssertionError("Should never happen");
}
```

Listing 7.3: Implementation of the task commit operation.

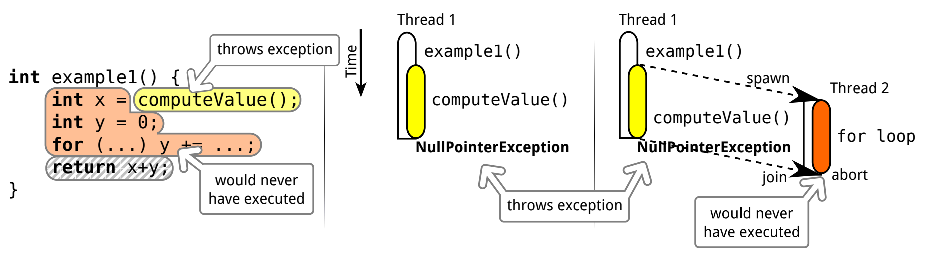


Figure 7.5: Variant of Figure 2.2, where a `NullPointerException` is triggered inside the `computeValue$speculative()` method. In the sequential version (center), the for loop would never have executed. In the MLS-parallelized-version (right) the for loop speculatively executes, but its results must be discarded.

7.1.3.2 Task Received Abort Order

A task may receive an abort order whenever it was spawned by an invalid parent, which itself also aborted.

If the task receives an order to abort, the task aborts its own transaction, signals its child (if any) to abort, and its hosting thread is returned to the thread pool. Any computation done was wasted, because it was based on invalid assumptions.

7.1.3.3 Result From Parent is an Exception

Whenever the result from a task's parent is an (uncaught) exception, a more complex case happens. Similarly to receiving an order to abort, whatever work the current speculative task has done is indeed invalid: In the original application, this code would never be executed, because the exception would be thrown before the code that the child was speculatively executing was reached. Figure 7.5 shows one such example: the `for` loop was executing speculatively, but `computeValue()` throws an exception; because `computeValue()` would execute before the loop in the original sequential application, this means that the loop would not have executed.

But, unlike in the previous case of an order to abort leading to a task being discarded, in the case of an exception the task must instead be re-executed. This re-execution is needed because the parent task no longer has access to the thread stack that was below the `spawnSpeculation()` method call, due to it being cleaned as part of the task spawn operation, as described in Section 7.1.1. Thus, all the remaining stack frames that were originally on the parent task are now part of the child task.

To correctly simulate the original behavior of the exception being thrown, the task first aborts its current STM transaction. If the task has a child speculation of its own, the task should signal the order to abort only if the child was spawned by the current task — that is, if it was the result of an in-order spawn, not the result of a task being re-parented.

The task must then re-resume its originating continuation. The resume operation re-initializes the execution stack back inside the `spawnSpeculation()` method. Inside `spawnSpeculation()`, a special check is then performed (in line 47 from Listing 7.2) for an exception received from the parent: When found, the exception is re-thrown, with the normal VM execution taking over and execution resuming at the applicable exception handler.

This abort ➜ re-resume ➜ rethrow exception scheme allows the JaSPEx-MLS framework to simulate the way the exception would appear in the original application — it would be seen as

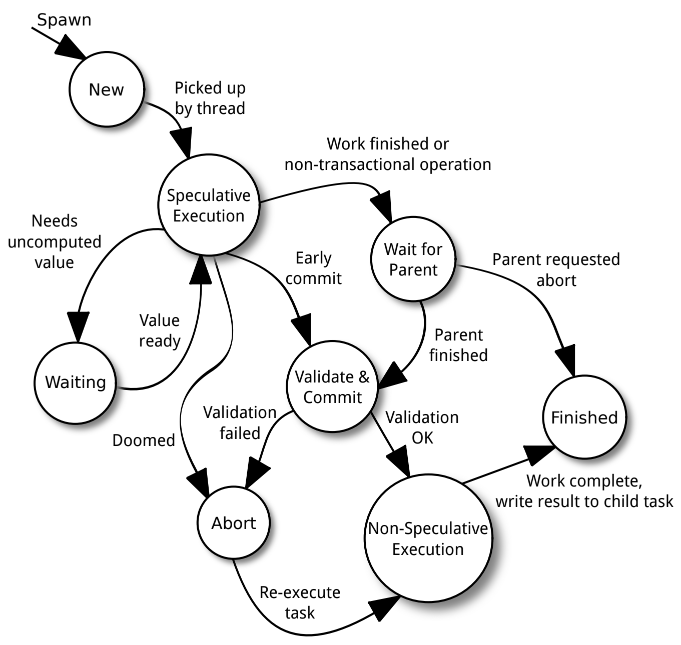


Figure 7.6: Life-cycle of a speculative task.

originating from inside the call site that the spawn point has replaced. Note that this process may occur several times, as the exception is propagated from parent to child task until either the exception is caught or only one task in the system remains, containing the application's `main()` method in its stack, and the exception is propagated past the `main()` method into an internal handler that displays an error and then shuts down the VM.

7.1.4 Speculative Task Life-Cycle

Figure 7.6 sums up the life-cycle of a speculative task. After being created, a task is queued for execution by JaSPEx-MLS's thread pool. The task then gets picked up, and starts its speculative execution. During execution, it may need to access a value from an older task: in this case, the task must wait until such a value is available (or generate a prediction, when using Return Value Prediction, as described in Section 7.4.5).

When one of the several conditions for triggering a commit/abort operation happens, the task first waits for its parent (if needed), and then tries to validate its work. If validation fails, the task aborts its transaction, and then re-executes in non-speculative mode. Otherwise, the task commits, finishes up any leftover work in non-speculative mode, and then writes its result to its child task, and its execution is complete.

7.1.5 Correctness and Progress

In a speculative parallelization system such as the JaSPEx-MLS framework, a speculative execution is defined as being correct if the observable state of the system at any given time is consistent with a possible state of the original, sequential application, given the same inputs. The JaSPEx-MLS framework takes advantage of this observable state definition by executing speculative tasks in a way that does not allow them to be observable until they are successfully validated.

The only observable difference between the original sequential application and its speculatively parallelized version should be execution performance. Note that execution performance does not necessarily mean speedup, as sometimes the framework may make poor decisions for a given application, and it also does not mean total execution time, as many applications run interactively, or for a set amount of time, instead of executing as fast as they can until they finish.

At the individual task level, a speculative task executes code that the framework believes would eventually be executed by the application. This execution is based on the currently-available tentative state of the program, and as such may not be correct. If the tentative state that a speculative task accesses remains unchanged until its validation, then the speculation was successful and its results can be published and incorporated into the system state — becoming part of the observable state; otherwise, it was not and its results must be discarded, and their existence must never be observable.

Consider, for instance, the example shown in Listing 7.4. As part of `example7()`'s execution, three methods are executed: `a()`, `b()`, and `c()`. Furthermore, consider that we run `a()`, `b()` and `c()` in parallel, with `b()` and `c()` executing speculatively, and `a()` executing non-speculatively. If all three methods start executing at around the same time, then `a()`, executing non-speculatively, will have access to the actual program state — the same state that would be seen by `a()` in the original, sequential application. In contrast, `b()` and `c()` will execute based on the same state as `a()`, whereas in the original application, `b()` would execute based on the state produced by `a()`, and `c()` would execute based on the state produced by both `b()` and `a()`. In the original application, `b()` would always observe the value `1` for the `count` variable, and `c()` would always observe `2`; when executed speculatively, it is possible for the speculative tasks to observe different values: if `b()` reaches its assert before `a()` then it will observe `count` to be `0`, and `c()` can observe count as `0` or `1` depending on the duration of the execution of both `a()` and `b()`.

Interference with the observable program state by a speculative execution is prevented by the transactification process (Section 6.1), which isolates heap changes through the use of the STM API and non-transactional operations via injected callbacks to the JaSPEx-MLS framework. At runtime, speculative execution is supported by JaSPEx-MLS's custom STM. The STM is responsible for both recording any state accesses done as part of a speculative execution for later validation, and with isolating any state changes done by that same speculative execution.

A speculative task may be incorrect due to two main reasons:

- *It never should have executed in the first place.* This case, previously shown in Figure 7.5, happens whenever the framework selects and starts executing a task that will not ever be needed given the particular state of this program instance.
- *The tentative state for the system was inaccurate.* An example for this is shown in Listing 7.4, where a given task may fail because of a concurrent access by an earlier task.

What happens when a speculative task becomes incorrect? Because the Java Platform is a managed code platform, incorrect tasks cannot go awry in the same manner as with native code applications: incorrect tasks cannot cause application crashes, nor — because of the transactification process — interfere with the observable program state. Incorrect tasks

```java
private class Example7 {
    private int count;

    public void example7() {
        count = 0;

        a(); // after a(), count is 1
        b(); // after b(), count is 2
        c(); // after b(), count is 3
    }

    private void a() {
        // ... code
        assert (++count == 1);
    }

    private void b() {
        // ... more code
        assert (++count == 2);
    }

    private void c() {
        // ...
        assert (++count == 3);
    }
}
```

Listing 7.4: Example to illustrate the different versions of the program state that are observed by speculative and non-speculative tasks. If `a()` were to be executed non-speculatively, with both `b()` and `c()` being executed speculatively, the assertions on `count`, which in the original application would never be triggered, may cause the speculative executions to abort, as both `b()` and `c()` are being executed based on tentative state, which may not include previous updates to the `count` variable.

executed by the JaSPEx-MLS framework can still be seen as safe because they cannot cause the state of the system to diverge from the semantics of the original sequential application.

As part of the early commit feature (described in Section 7.4.3), every task will eventually be forced to validate itself as long as it attempts to access the heap, allowing the framework to validate the task's execution and decide on how it should proceed.

It is also possible for a task to read a wrong value that causes the task to erroneously loop infinitely or for too long without calling any of the framework's hooks. In the case of these zombie tasks, it is hard to stop them because the Java Platform does not include APIs allowing threads to forcefully control other threads. In all of the program examples and benchmarks used in my work, I did not encounter this issue, and, as such, I did not address it. However, if needed, one way to tackle it would be to use the Java Debug Interface to control the VM and stop zombie threads.

The definition of progress on a speculative parallelization system is also different from those used in STMs [Bushkov et al., 2012], as individual tasks are constrained by the order imposed by the original application semantics.

Because JaSPEx-MLS allows a task to migrate from speculative to non-speculative execution, as each non-speculative task finishes and triggers the early commit of its child task, there is usually always an active non-speculative task in the system. Whenever there is a non-speculative task executing code, the system is guaranteed to be making progress, as it is executing code in the original sequential program order and that follows the original application semantics.

When a non-speculative task ends, and while its child task has yet to switch to executing in non-speculative mode, the system is not guaranteed to be progressing — only speculative tasks are active, and they may not be correct. Because of the early commit feature, this window is usually very small, and in practice I have found that there is no need for a heavier mechanism that would force the child to stop its work and validate itself immediately.

It is nevertheless possible for all of the tasks on the system to be running speculatively, and none of them be correct: this means that, while the oldest task is not aborted and restarted, the system will not be making any progress, because all the work being performed will be discarded. In this interval, no useful work will be done, but progress will be resumed as soon as a thread switches back to non-speculative mode, and unless it turns into a zombie, every thread will eventually detect that it is incorrect and abort, going back to the thread pool to fetch new tasks.

It is not impossible, though, that without the zombie detection mechanism the system may stop progressing, but again in all of my many experiments I never observed this to occur. With the zombie detection mechanism, the system would be guaranteed to always eventually progress, as threads would all in turn become non-speculative and finish their tasks (or at least be forcibly restarted by the zombie task detector).

7.2 Thread Pool Management

After a new speculative task is created, it is submitted for execution to the JaSPEx-MLS thread pool. The framework's thread pool is a custom thread pool that includes lightweight thread state polling mechanisms, support for live switching of queuing modes (described in Section 7.2.1), and support for task freezing (described in Section 7.3). It is based on Java's `ThreadPoolExecutor` API, and uses a limited number of threads, based on the number of available CPUs in the machine.

In my first design, the thread pool used direct hand-offs between threads. This meant that new tasks were only accepted if there were idle threads available on the system. This design was chosen to avoid deadlocks: Because, as described in Section 7.1.1.1, JaSPEx-MLS's design allows for both *out-of-order* and *in-order* speculative spawn operations, task spawn order becomes unpredictable. This unpredictable order, when combined with waiting, means that when employing buffering instead of using direct hand-offs, the system can end up in a state where all available threads in the thread pool are deadlocked.

To see why the use of buffering can lead to deadlocks, consider the example from Table 7.1, showing a thread pool that employs buffering and uses two threads: $t1$ and $t2$. Thread $t1$ starts out hosting Task A, whereas thread $t2$ is hosting Task B (Task B is Task A's child). Task A then attempts to spawn a new task — Task α. Without buffering, Task α would not be created, as both $t1$ and $t2$ are busy; using buffering, Task α is successfully spawned and placed in the thread pool's buffer. As explained in Section 7.1.1, this new out-of-order spawned task causes Task B to be reparented, so that the resulting parenting relationship becomes A $\leftarrow$ α $\leftarrow$ B. After

Thread t1	**Thread t2**	**Buffer**	**Parenting**	**Action**
Task A	Task B	-	A ← B	$t1$ spawns Task α
Task A	Task B	Task α	A ← α ← B	-
			...	
Task A	Task B	Task α	A ← α ← B	$t1$ spawns Task β
Task A	Task B	Task α, Task β	A ← β ← α ← B	-
			...	
Task A	Task B	Task α, Task β	A ← β ← α ← B	$t1$ retires Task A
Idle	Task B	Task α, Task β	β ← α ← B	$t1$ picks up Task α
Task α	Task B	Task β	β ← α ← B	-
			...	
Task α	Task B	Task β	β ← α ← B	$t2$ finishes work, needs to wait for α
Task α	Waiting for α	Task β	β ← α ← B	-
Task α	Waiting for α	Task β	β ← α ← B	$t1$ finishes work, needs to wait for β
Waiting for β	Waiting for α	Task β	β ← α ← B	-
		System is deadlocked		

Time ↓

Table 7.1: Example of task buffering leading to a system deadlock. Using a generic buffering scheme, and due to JaSPEx-MLS's unpredictable task spawn order, it is possible for the system to end up in a state where all the threads from the thread pool are busy waiting for a task that is still in the buffer to finish.

some time, Task A spawns another task — Task β — that is also placed in the thread pool's buffer. After this operation, the parenting relationship becomes A ← β ← α ← B. After finishing up Task A, thread $t1$ then picks up Task α from the buffer for execution, and herein lies the trouble: because Task α itself depends on Task β that is still in the buffer, when later thread $t1$ blocks waiting for Task β to finish, there will be no available threads to execute this task, as all threads are waiting for other tasks to finish.

Using a bigger thread pool is only a band-aid solution: it reduces the chance of deadlocks happening, but deadlocks such as the one in Table 7.1 can happen in any thread pool with a limited number of threads — applications which make use of recursion, in particular, can easily trigger this issue. Also, using a thread pool with significantly more threads than the number of hardware threads supported by the host machine leads to scheduling and context switching overheads, especially when most threads are working and not just waiting.

Tracking the parenting order in this case adds too much overhead, as all threads in the system would have to update and access a shared data structure on every buffer access.

By avoiding buffering, JaSPEx-MLS ensured that at least one of the threads in the system was making progress, as it was hosting the oldest task in the system, which would never need to wait, as by definition it could already retrieve any needed information from its ancestors.

7.2.1 Hybrid Thread Pool Buffering

Whereas not using buffering ensures the deadlock-free execution of speculative tasks, it also causes thread usage to be sub-optimal: because no speculative tasks are accepted unless there are idle threads available, this means that when a thread finishes its work, it will stay idle until another active thread reaches a spawn point. In cases where an application has large tasks, this means that for a long time only a few or in the worst case only one of the program threads will be executing, with the remaining threads all waiting for new work to be submitted.

As benchmarking revealed that task buffering when it did not cause any issues was more efficient than direct hand-overs to the thread pool, I developed a novel technique to solve this issue: *hybrid thread pool buffering*.

Hybrid thread pool buffering works by keeping both a buffered and a non-buffered work queue for managing tasks on the thread pool. By default, the buffered work queue is employed, allowing more tasks to be generated than there are threads in the pool to work on them.

The thread pool is then augmented with a dedicated thread that periodically polls the state of the buffered queue. Because tasks are queued for execution at most once, if the polling thread observes the same task at the head of the buffer for a configurable amount of time (I expect most tasks to execute in sub-second times), it then proceeds to check the current state (using `Thread.getState()`) of each of the pool's threads.

When the polling thread observes the same unchanging task at the head of the queue, and that all threads in the pool are in the `Thread.State.WAITING` state — a state that indicates that a thread is blocked and waiting for other thread to perform some action and wake it up again — then the system is possibly deadlocked.

When a deadlock is detected, buffering is disabled and task submission is switched back to direct hand-offs. To break the deadlock, the polling thread then creates a new temporary thread for each of the tasks still present on the buffered queue. While this means that temporarily there will be a large number of threads contending for the machine's resources, depending on the configured buffer size — by default JaSPEx-MLS uses a size of 64, resulting in 64 temporary threads — temporary threads only live to execute a single task, and after the threads finish executing their tasks and start retiring the normal number of threads in the system is restored. In my experiments, this "all-out" transition mode was faster than the alternative of only creating a single or small number of new threads, then randomly picking tasks from the pool for them to execute, and then to continue monitoring the system to discover if it would enter deadlock again, subsequently creating more threads and repeating the process until the task that was at the origin of the deadlock was finally executed.

Note that there is no impact to program correction if the polling thread suffers from a data race when checking for deadlocks, as it will only mean that the system will pessimistically switch to direct hand-offs when it would have not needed to. In practice, I never observed this to happen.

The pseudo-code for the thread pool polling mechanism is shown in Listing 7.5. My hybrid approach is able to combine the best of both task queuing modes: It maintains correctness for all applications while providing increased performance to those where buffering causes no issues, and posing no extra task tracking or execution overheads.

Whereas it would be possible to extend the pool so that it would switch back and forth between buffering and direct hand-offs, in practice I found that an application that triggers a deadlock will very likely trigger it more than once, and so the current design keeps using the direct hand-off approach after the first deadlock is detected.

Note that if, when a task is submitted, the pool is full, the spawn is aborted, and the application behaves the same as when running the original sequential code. This may happen in both pool queuing modes, either when the pool is fully busy (for the direct hand-off case), or when the buffer is full (when buffering is being employed).

```
while (true) {
    head = queue.peek();
    sleep(DELAY);
    if ((head == queue.peek()) && (all threads in WAITING state)) {
        // system is probably deadlocked
        fallback to direct hand-offs queue
        for (task t in old buffered queue) {
            create temporary thread to execute task t
        }
    }
}
```

Listing 7.5: Pseudo-code for the thread pool monitoring mechanism. A dedicated thread periodically tests the thread pool to check if the system is deadlocked and, if it is, the pool is switched back to direct hand-offs and temporary threads are created and assigned to work on the backlog.

Because the rejection of fully-created tasks — late rejects, as described in Section 7.1.1 — adds overhead, JaSPEx-MLS reduces this effect by performing a quick check of the state of the pool before attempting to spawn a new task. This quick check must also be aware of the hybrid buffering scheme: In buffering mode, it checks the current state of the buffer; in direct hand-off mode, it checks the number of busy threads.

A further advantage of using the buffered mode is that in practice I found that the buffer is seldom full, leading to a lower task rejection rate, especially costly late rejections.

7.3 Task Freeze

When, in Section 7.1.4, I examined the life-cycle of a speculative task, I described that there are a number of conditions in which a task must wait for some other task to finish execution:

- A task may need to access the result (future) from another speculative task. If the other task has not yet finished its computation, the current task must wait until the needed value becomes available.
- A task may need to wait before being able to validate its STM transaction (after finishing its work, or before invoking a non-transactional operation), and again the task must wait until its parent is finished, so that its transaction can be committed.

For both cases, threads hosting waiting tasks are still counted as busy, as they are unavailable for executing further tasks, thus leaving the machine's parallel resources underused.

A possible approach to solve this challenge would be to reuse threads, but straightforward thread reuse may itself lead to issues. To see why, consider a system state where the thread pool buffer contains a number of tasks: A, B, C. Additionally consider that the parenting relationship is A ⇠ B ⇠ C — A is the grandparent, B is the parent, and C is the child task.

While other threads are busy, a thread $t1$ picks up task A for execution. Then, after a while, another thread — $t2$ — picks up task B for execution. Task B then needs to execute a non-transactional operation, and must wait for task A to finish before it can validate its own transaction. Instead of waiting, consider now that $t2$ picks up task C for execution.

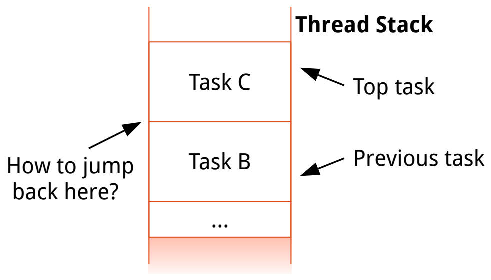


Figure 7.7: State of the thread stack for $t2$.

Consider now that task C finishes its work and must be validated before it can commit. Before it can do so, it must first wait for its parent — task B — to do the same. The problem is that its parent is pending on the same thread that task C is. This state of $t2$'s stack is shown in Figure 7.7. Because JaSPEx-MLS cannot access the previous task, the system would deadlock, unable to ever finish the new task or to switch back to executing the previous one.

To safely support thread reuse, I developed the concept of *task freezing*, which solves the above issue by relying on the extended JVM with support for continuations. Whenever a thread executing a task would block waiting for its parent task to finish, instead the framework *freezes* the task, by capturing a continuation containing the current state of the task and saving its currently-active STM transaction.

By snapshotting the thread's state, the continuation capture operation allows the task to later be reconstructed in the same or other thread, with the STM transaction being used to keep the heap state needed for the task.

After a task is frozen, the thread that was hosting it is returned to the thread pool, where it can pick up other tasks for execution.

Going back to the example, using task freeze would allow task B to be snapshotted, thread $t2$'s stack to be cleaned (task B would not stay pending on the stack), and $t2$ to start executing task C on a clean stack.

A frozen task is associated with its parent task, which will be responsible for finishing the frozen task's work after its own — frozen tasks are not submitted to the thread pool again.

The *thaw* operation happens when, after finishing its work, the parent task discovers a frozen child task waiting to be completed. As the parent task has finished its own work, the thread can directly switch to working on the child task without needing to return to the thread pool — the child's continuation is resumed, its STM transaction is validated, and execution proceeds from where the freeze left off.

In the example, after a while task C would need to wait for task B to complete, which itself would be frozen. Then, in the meanwhile, thread $t1$ would finish executing task A, and notice that task B was frozen. $t1$ would then thaw task B, and finish its execution. After task B is finished, task C is finally allowed to finish its work.

Note that it is possible for a queue of frozen tasks to form, and a parent may have to thaw several children, always directly switching between them without returning to the thread pool.

7.3.1 Lightweight Task Freeze

Because capturing continuations adds some overhead, I have further identified and optimized a common case where the JaSPEx-MLS framework can avoid the need for continuations. Whenever a child task is able to complete its work, but needs to wait for its parent to finish before it can validate and commit its own speculative state changes to the global program state — i.e. when a task is in the "Wait for Parent" state of the task life-cycle (Section 7.1.4) because it finished its work — the framework can employ a lightweight freeze technique.

Because the task has finished its work, there is no longer any state on the stack that needs to be preserved, only the task's transaction and outputs, in the form of a return value and any heap modifications done need to be preserved. In this case, the freeze operation only needs to save the STM transaction and the return value, and the thread stack is disregarded.

Thawing a lightweight frozen task consists only on validating and committing its STM transaction, avoiding an unneeded continuation capture/resume cycle.

7.4 Custom STM Model

A common complaint of Software Transactional Memories is that their added overhead is non-negligible when compared to other synchronization alternatives.

The design of JaSPEx-MLS's custom STM model is aimed at taking advantage of specificities of the speculative parallelization model to minimize overhead as much as possible. As a consequence, the framework's model is unsafe as a generic STM model — it is only safe when used to support an execution model that closely follows speculative parallelization, and as, in the coming sections, I describe this model, I shall highlight why and how it takes advantage of specific characteristics of the speculative parallelization model to be simpler.

7.4.1 Thread Execution Modes and Relaxed Isolation

The JaSPEx-MLS framework's STM recognizes two execution modes for threads executing tasks: there is a single *program-order* thread, and multiple *speculative* threads.

The program-order thread is the one running the oldest (in terms of original sequential execution timeline) code from the application. It is executing either the first task in the system, or a subsequent task, where all tasks that came before it have since retired, and the task's own transaction validation has been successful.

Thus, operations and modifications performed by the program-order thread are not speculative — they are executed with the same state and data that the normal sequential code would have, and they never have to be undone.

In contrast, speculative threads are running tasks that are still speculative — they have not been validated and their work might still need to be undone. As their work is speculative, and their results are tentative, these results should be isolated from other concurrently-running speculative threads.

Because the program-order thread is not running speculatively, it is allowed to directly access and mutate the program state, without extra instrumentation. This means that any write calls made by this thread are directly performed to the heap, and any read calls return the latest value

```java
public static Object loadObject(Object instance, Object value, long offset) {
    Transaction tx = Transaction.current();
    if (tx == null) return value;
    // ... normal speculative load path
}

public static void storeObject(Object instance, Object value, long offset) {
    if (instance == null) throw new NullPointerException();
    Transaction tx = Transaction.current();
    if (tx == null) {
        UNSAFE.putObject(instance, offset, value);
        return;
    }
    // ... normal speculative store path
}

// ... variants for other types
```

Listing 7.6: Partial implementation of the STM API shown in Listing 6.3. Heap operations invoked without an active transaction (by the program-order thread) are immediately applied, without going through the rest of the STM. A similar fast-path is implemented for array read and write operations.

from the memory position. Listing 7.6 illustrates part of the STM API implementation, showing that both reads and writes are directly performed to the heap when no active transaction is found. Note that the extra `value` parameter that was described as being passed to the STM as an optimization in Section 6.1.2 is useful exactly for this arrangement.

This direct-access strategy substantially lowers the overheads of code running in program-order, which is especially important whenever a stretch of program is being executed sequentially without any parallelization. Note that because the program-order thread never aborts, no undo log needs to be kept.

As a side effect of this modification, isolation is relaxed between the non-speculative thread, and other concurrently running speculative threads: speculative threads are able to observe heap mutations performed by the non-speculative thread as they happen.

Due to the relaxing of isolation, speculative threads may observe inconsistent states and concurrent accesses will be data races. This is where speculative parallelization differs from STM: Whereas in STM it is very important to enforce isolation between concurrent transactions — otherwise causing problems such as inconsistent reads [Dice et al., 2006], and breaking opacity [Guerraoui and Kapałka, 2008] — under speculative parallelization the model already lends itself to inconsistent states, as it is based on the concept of executing code from the future given the tentative version of the program state that is available now.

In particular, reading wrong values may indeed cause a speculative execution to get stuck on an infinite loop, because a speculative thread may have observed an invalid state that never would have been observed in the original, sequential code.

Because these issues are already a part of any speculative parallelization system, dealing with them should not be a part of the STM model — it should be up to the framework to identify and kill runaway zombie tasks.

This breach in isolation between the program-order thread and speculative threads also works as value forwarding: Speculative threads optimistically have earlier access to values from the program-order thread than they would have if that thread executed in isolation.

Note that whereas isolation has been relaxed, it cannot be eliminated: The program-order thread still needs to be isolated from speculative threads, as speculative threads are from each other; otherwise, this would mean that code from earlier in the program would be able to observe results from code later in the program.

7.4.2 STM Design Choices

My STM model employs optimistic concurrency control for threads that are running speculatively. Each thread keeps its own read-set. By default the read-set is represented using a linked-list. Each memory read operation is recorded in a tuple (*instance*, *offset*, *read value*), which is appended to this list. Optionally, the read-set may be switched to using a hash map, whenever there are many repeated accesses to the same memory locations. Repeated accesses may happen because, unlike with most other STMs, the transactified code has been written without any intent to use STM, so it may keep reading the same memory locations because the programmer decided not to or it was not practical to save working values into local variables. In the hash map version, the pair (*instance*, *offset*) is used as a key, and in contrast with the linked list version, only one entry for each memory location is stored in the map.

The read-set is accurate: Each individual memory location is recorded for later validation — no false conflicts may ever occur.

To avoid as much memory overhead as possible, and to allow STM access to arrays (array representation is VM-internal and thus unmodifiable) and JDK-provided objects, all metadata is thread-local and transient for the duration of the transaction, and no changes are needed to the in-memory layout of the original objects and classes for the STM to support them.

Validation is performed in a value-based manner: A read of a value v from a memory position m is valid if at the time of validation m still contains v. Additionally, if there are any duplicate entries representing reads of m in the linked-list read-set, they all must have observed the same value v. Whenever a map is being used to represent the read-set, the STM additionally implements early abort: If some memory location m has changed its value since an earlier read, the STM signals to the task management code that the current transaction is doomed and should be early aborted. Using a value-based approach both reduces the need to track object versions/epochs — as required by version/epoch-based STMs — and can also reduce aborts because only the final value for a memory location is tested, not any intermediate values[2] — as long as they are not observed by the concurrently-executing transaction.

The write-set is a redo-log; writes from speculative threads are kept in a thread-local map and are written-back at commit time.

7.4.3 Commit Operation and Commit Ordering

Unlike with normal STMs, tasks extracted from a sequential application have an implicit order: the execution order from the original application.

[2]Note that this may happen for instance with counters, which may be incremented and then decremented again, going back to their original state.

When these tasks are mapped to transactions, they must still follow this order: A speculative transaction must be linearizable [Herlihy and Wing, 1990] at the same relative moment the original code would be; that is, its results should only become observable from the exact moment where the original code would take effect, in the original sequential application.

This allows for several simplifications. The commit operation, for instance, does not need to be atomic — there is no problem if other threads observe partial commits, as they record all reads and will later detect possible issues during their own validation.

Only one thread — the one hosting the oldest task — that is transitioning from speculative into program-order mode, is allowed to commit at any one time, also simplifying synchronization.

To commit a transaction, the STM first validates its read-set. Unlike with other STMs, validation still needs to happen for read-only transactions, as the STM needs to ensure that they accessed the correct heap state while executing — unlike previous differences, this requirement adds overhead not normally present on other STMs. Because no older transactions may exist in the system at commit-time, validation also does not need any kind of extra synchronization to be correct. If validation succeeds, the STM writes back all values from the write-set, and the transaction is finished.

When a thread working in program-order finishes its execution, it is returned to the thread pool, and its child task can now validate its own work and become the new program-order thread. Similarly to the *implicit commits* performed by [Yoo and Lee, 2008], the retired task flips a flag on the child task indicating that it should validate its work and become the program-order thread; this flag is checked during each STM read and write operation. This *early commit* feature allows threads to switch to program-order mode faster, reducing unneeded speculative execution overheads.

7.4.4 Transparent Handling of Futures in the Write-Set

As previously described in Section 6.2, the STM directly supports hosting futures on the write-set as placeholders for the returned values from methods.

The first time an application attempts to read from a future previously recorded in its corresponding transaction's write-set, the STM calls the future's `get()`, waiting if needed for the value to become available. Afterwards, the STM caches the result, and further lookups will return this value directly.

At commit-time, each future is accessed to obtain its value, and this value is written-back to the heap. During commit, because of the strict commit ordering that is enforced, the STM never needs to wait for a future to finish computing — if it found a future, it means that the task that ran its corresponding `spawn` operation was ordered before the current task; because in the chosen speculative model a transaction can only be committed if it is hosting the oldest active task in the system, then that means the older task that allocated the future is guaranteed to have finished its work and not be active anymore.

7.4.5 Return Value Prediction

One of the biggest challenges in the MLS model is dealing with operations that manipulate the return values of methods that have yet to finish executing. The JaSPEx-MLS framework

```java
// Interface used to extend Callables with support for value prediction.
public interface PredictableCallable {
    public Object predict();
    public void updatePrediction(Object realValue);
}
```

Listing 7.7: The `PredictableCallable` interface, used by JaSPEx-MLS to augment `Callables` with support for value prediction.

represents the return values of these methods with futures, and the modifications performed to the input application's code allow futures to be written both to local variables and also to the heap, as described in Chapter 6.

But the above options are useful only if the value from the method call is not read immediately. Otherwise, no useful work is done in parallel: a child speculative task spawned to run code following a method call that needs the value from the method immediately stalls waiting for the result from its parent task. By default, whenever the bytecode preparation module detects that this would happen, it declines to inject the spawn operation that would create a new task.

An alternative solution to this predicament lies with the use of Return Value Prediction (RVP) [Hu et al., 2003, Pickett, 2012]. The idea of RVP is that whenever a task would stall waiting for a returned value to be produced by another task, we guess a probable value for the computation, and continue executing the task using this assumption — that is, we speculate on the returned value from a task.

I have implemented RVP as a novel extension to JaSPEx-MLS's STM model: Whenever a prediction is produced, it is registered with the STM as a read of a specially-reserved memory location. This memory location is unique for each task (future) from which a prediction is obtained: It is possible for a speculative task to obtain multiple predictions corresponding to values from multiple other tasks.

When a task finishes and produces the final return value, it writes it to the special memory location during its commit operation. When later the task that read the prediction attempts to commit, the memory location hosting the prediction is checked as part of the normal read-set validation. Because verification is value-based, if the prediction was correct, its value will be seen as valid by the STM, otherwise validation will fail and the speculative task is aborted and re-executed.

Predictors are associated with each `Callable` class, corresponding to a given task spawn point. When RVP is enabled, each generated `Callable` additionally implements the `PredictableCallable` interface shown in Listing 7.7. Predictor instances are kept inside static variables on each `Callable` class, and are initialized via calls to a `PredictorFactory` class.

The `PredictorFactory` implements pluggable prediction strategies, and can be used to instance different predictor types for each `Callable`. Currently, only a *last value* predictor is implemented.

Predictors are updated during transaction commit operations: whenever a result for a given `Callable` is being written, its corresponding predictor is also updated.

```
1 void example6$speculative() {
2     SomeClass sc = new SomeClass(null);
3     int x = spawn computeValue$speculative();
4     sc.field = 42;
5     sc.otherField++;
6     ...
7 }
```

Listing 7.8: Example method and its task division. The previously-existing task (in red) was active when the method was entered, and goes on to execute the `computeValue$speculative()` method; the new task (in green) is spawned to execute the code following the execution of `computeValue$speculative()`.

7.4.6 Adapting Captured Memory to MLS

Most of the STM overhead from threads executing in speculative mode comes from them having to deal with the read and write sets, instead of directly accessing the objects as the program-order thread does.

Because this interception is a source of overheads on many STMs, reducing them has been an area of active research for STM developers. [Dragojević et al., 2009b] identified that some memory accesses are made to what they define as *captured memory*: memory that is allocated inside a transaction and that does not and cannot escape its allocating transaction. For most STMs, this means that due to isolation, objects and arrays created inside a transaction will not be accessible to any other transaction until their creating transaction finishes, if at all (they may be throw-away objects). Because they are accessible only to their creating transaction, objects in captured memory may be mutated directly by that transaction, without needing to employ any kind of STM interception, and without causing concurrency issues.

To apply the concept of captured memory to the JaSPEx-MLS framework's STM and MLS models, I needed to relax its definition. Whereas under a normal STM transactions run isolated, and new objects and arrays created by them are not visible to other transactions, under the MLS model this does not hold. As an example, consider Listing 7.8: the existing task creates a new instance of `SomeClass`, and then a new speculation is spawned that will access this instance. The problem here is that under the original captured memory definition, `sc` is part of the previously-existing (red) task's captured memory; but, in lines 4 and 5, the new (green) task also accesses `sc`, breaking the assumption that objects in captured memory are not accessible to concurrent transactions.

Even though at first it appears that it is not possible to take advantage of captured memory in MLS — because as we saw from the example, objects in captured memory may be accessed by other tasks depending on run-time spawn behavior — my implementation is able to take advantage of the fixed commit order imposed by the MLS model to solve this issue. The key insight is that whenever a speculative task st_1 creates an object, and another task st_2 is able to access it, then by the MLS model's definition st_2 must be more speculative (comes later in the original sequential program order) than st_1, which created the object — otherwise, if st_2 was less speculative than st_1 and was still accessing st_1's state, this would mean that the original sequential program semantics was being broken. This can be seen in Listing 7.8: The previously-existing (red) task is clearly less speculative than the new (green) task. If the existing task was allowed to observe state changes from the new task, it would erroneously be performing reads from future state.

As a consequence, and although concurrent tasks may have access to objects from captured memory, the MLS model guarantees that their writes will come later than those from the creating task, which always precedes them. As such, this is very similar to the semantics that the framework's STM already provided between the program-order and speculative threads: the program-order thread always writes directly to memory, whereas speculative threads use STM to buffer their accesses. I extended JaSPEx-MLS's STM model by treating accesses to captured memory as if they are done by a task running in program-order mode, while the remaining more speculative threads use the normal STM mechanisms to validate and protect their accesses.

Whereas Dragojevic et al.'s proposal was aimed at a C++ runtime and needed extra bookkeeping, recently [Carvalho and Cachopo, 2013] introduced Lightweight Identification of Captured Memory (LICM), an improved technique for identifying captured memory that outperforms previous approaches while avoiding the need for extra data structures and better supporting managed runtimes. Their proposal works by adding a unique fingerprint object to each transaction, and by tagging every new object and array with the fingerprint of their encompassing transaction. Then, whenever a transaction accesses an object that is tagged with its own fingerprint, the object is accessed directly, bypassing the transaction's read and write-sets.

To optimize accesses to captured memory, I adopted a LICM-inspired approach in my custom STM, by using JaSPEx-MLS's bytecode preparation step to optionally add creating transaction's tag fields to objects, and by directly reading from, and writing to objects with a tag that matches the current transaction. The aforementioned issue of other transactions having access to objects in captured memory is handled by forcing them to access these objects via the usual STM mechanisms.

The STM modifications to support LICM are shown in Listing 7.9. Every `loadType()` and `storeType()` STM method (`loadInt()` and `storeInt()` in the example) is augmented with a call to the `isLocal()` method (lines 3 and 18), and directly reads from or writes to the target object when it determines that the object is part of the transaction's captured memory. The `isLocal()` method compares the creating transaction tag from the object (obtained by calling `$getOwner()`) with the current transaction's `OWNER_TAG`. The `getOwnerTag()` method is used during object initialization to obtain the transaction's `OWNER_TAG`.

An example of the changes to classes to support LICM is shown in Listing 7.10. A new field `$owner` is added, and the `$getOwner()` method from interface `Transactional` is implemented to allow access to it. `$owner` is initialized by calling the previously-introduced `Transaction.getOwnerTag()` method.

Of special interest to the MLS model, the addition of captured memory also solves an issue that would normally lead to speculation aborts. Consider again, for instance, the access in Listing 7.8 to `otherField` (line 5): If this field was initialized inside `SomeClass`'s constructor, this initialization would still be on the previously-existing task's (uncommitted) write-set, and so any access done by the new task would not be able to observe the correct value for the field. In general, without captured memory, accesses to newly created objects do not read the correct values while they are still being kept on the creating transaction's write-set, which is

```
 1  public static Object loadInt(Object targetObject, int value, long offset) {
 2      Transaction tx = Transaction.current();
 3      if (tx == null || isLocal(tx, instance))) {
 4          // access field value directly
 5          return value;
 6      } else {
 7          // access field value via STM mechanisms
 8          ...
 9      }
10  }
11  //... variants for the remaining types (and their static variants) are likewise
12  // changed: loadBoolean(), loadByte(), loadChar(), loadDouble(), loadFloat(),
13  // loadLong(), loadShort()
14
15  public static void storeInt(Object instance, int value, long offset) {
16      if (instance == null) throw new NullPointerException();
17      Transaction tx = Transaction.current();
18      if (tx == null || isLocal(tx, instance)) {
19          // write to field directly
20          UNSAFE.putInt(instance, offset, value);
21      } else {
22          // store field in the write-set
23          ...
24      }
25  }
26  //... variants for the remaining types (similar to those above)
27
28  static boolean isLocal(Transaction tx, Object o) {
29      return (o instanceof Transactional) &&
30        ((Transactional) o).$getOwner() == tx.OWNER_TAG;
31  }
32
33  // Used during object initialization to obtain the current transaction's (if any is
34  // active) OWNER_TAG
35  public static Object getOwner() {
36      Transaction tx = current();
37      return tx != null ? tx.OWNER_TAG : null;
38  }
```

Listing 7.9: Modifications to the JaSPEx-MLS framework's STM to support LICM. The `loadInt()` STM method is augmented with a call to the `isLocal()` method (line 3), and directly reads from the target object when it determines that the object is part of the current transaction's captured memory. The `storeInt()` STM method is also augmented with the `isLocal()` test (line 18), and directly writes to the target object using the `Unsafe` API when the object is part of the current transaction's captured memory. The `isLocal()` method compares the creating transaction tag from the object (obtained by calling `$getOwner()`) with the current transaction's `OWNER_TAG`. The `getOwnerTag()` method is used during object initialization to access the `OWNER_TAG` object.

```
// Original class
class LicmExample {
    // ...
}

// LICM Modifications
class LicmExample implements jaspex.speculation.runtime.Transactional {
    // Obtain the current transaction's (if any) OWNER_TAG
    public final Object $owner = Transaction.getOwnerTag();

    public Object $getOwner() {
        return $owner;
    }

    // ...
}
```

Listing 7.10: Modifying objects to support LICM. A new field `$owner` is added, and the `$getOwner()` method from interface `Transactional` is implemented to access it. `$owner` is initialized by calling the `Transaction.getOwnerTag()` method from the STM API.

thread-local and not accessible to concurrent transactions. By allowing direct writes to objects, this issue is solved, and another source of mis-speculations is removed.

7.5 Profiling an Application for MLS Parallelization

To avoid selecting unprofitable methods for conversion into spawn points, the JaSPEx-MLS framework includes a special execution mode that gathers profiling information.

This execution mode modifies JaSPEx-MLS's normal parallelized execution: Of all the threads in the thread pool, only one of them is allowed to be executing code at any one time. This restriction allows the framework to approximately measure the time spent executing each speculative task. At certain execution points, the executing thread yields so that one of its concurrent tasks may be executed.

By tracking how much time a task spawned from a given spawn point spent executing speculatively, we may decide if that spawn point is on average speeding up or slowing down the target application. The framework also tracks the success of STM transactions, although the simulated execution schedule means that these may not reflect all the possible outcomes from the normal execution mode.

An excerpt of the information generated by the profiler is shown in Listing 7.11. For each spawn point (in this case, a spawn point inside `avrora.sim.GenInterpreter.visit()` which called `avrora.sim.GenInterpreter.popByte()`), the output includes a count of tasks originating from the spawn point which executed speculatively inside a transaction (`inTx`), as well as average, minimum and maximum times (in μs) for these tasks. The percentage of tasks for which their duration was below a configurable cutoff is also shown. The same counts are shown for time spent executing outside a transaction (`outsideTx`). Lastly, the output includes a task commit and abort count, as well as the number of tasks that did not execute inside a transaction at all — this happens whenever a task is picked up for execution only after its parent task has already finished its work. In the shown example, this call site is clearly not

```
S0$avrora.sim.GenInterpreter.visit / 173$avrora.sim.GenInterpreter.popByte
       inTx executed: 1310 avg: 150 min: 90 max:   290 belowCutoff: 100.00% (0 above)
  outsideTx executed: 1310 avg:  73 min: 16 max: 55823 belowCutoff:  99.92% (1 above)
  committed: 17 / aborted: 1293 / noTransaction: 0
```

Listing 7.11: Sample profiler output for an execution of *Avrora* (Chapter 8) using JaSPEx-MLS's profiling mode. For each executed task, the profiler keeps execution times inside and outside a speculative transaction, and also if the task successfully committed or had to be aborted.

profitable for speculation: tasks are too short, averaging only 90 μs of speculative execution time, and most of them end with an abort, with only 17 out of 1293 speculative tasks having committed successfully.

After executing the target application using the profiling mode, a report is generated containing methods that should not be used as spawn points. This report can be used as an input to the method selection process previously described in Section 6.2.1, automatically improving subsequent invocations of the JaSPEx-MLS framework with this application.

7.5.1 Analyzing Transactions

The JaSPEx-MLS framework additionally features an execution mode which performs extra transaction statistics gathering: the transaction analysis mode. This mode gathers detailed information about each transaction's read and write-sets, and globally accounts for all the fields and arrays that caused transaction validation to fail. In this mode, the framework will also warn about any non-transactional operations that caused speculative executions to stall.

Listing 7.12 shows an example of the output of the transaction analysis mode for an aborted transaction. The transaction's read and write sets are divided by class, and for each class the number of read/write operations to its fields are shown. For instance, in the shown example and for instances of the `FileMarkingTokenManager` class, there was one recorded read of the `curLexState` field, one of the `jjround` field, and eight of the `input_stream` field. As the shown transaction failed to validate, the output additionally includes which fields were a source of conflicts. Note that JaSPEx-MLS's STM would normally flag a transaction as invalid after finding the first invalid read, but in this mode the STM always validates the entire read-set and accounts for all invalid fields.

After an application exits, the framework will also print a summarized list of all the fields that caused transaction aborts and the number of times their validation failed. This list can be used to identify which parts of an application are being a source of speculative task aborts.

While not directly usable by the JaSPEx-MLS framework, the information gathered by the transaction analysis mode can be used to guide a developer in identifying and refactoring parts of an application's code which are inhibiting concurrency. Thus, this information is useful both as a guide for modifying an application to better take advantage of speculative parallelization, and also as a basis for identifying shared state for manual application parallelization.

7.6 Summary

In this chapter, I describe the run-time part of the JaSPEx-MLS framework.

```
ABORTED Transaction Statistics for jaspex.stm.Transaction@2364e850 (host Thread[WorkT13])
Source: SpeculationTask@35cfee57
Parent: 438$avrora.syntax.atmel.AtmelParser.Statement,
        called from S1$avrora.syntax.atmel.AtmelParser.Module
ReadSet size: 57
    C[]
        [] 5
    J[]
        [] 1
    java.lang.String[]
        [] 1
    avrora.syntax.atmel.AtmelParser
        jj_ntk 3 (2 failed validation)
        token_source 1
        token 2 (2 failed validation)
    avrora.syntax.atmel.FileMarkingTokenManager
        curLexState 1
        jjround 1 (1 failed validation)
        input_stream 8
...
WriteSet size: 25
    I[]
        [] 10
    avrora.syntax.atmel.AtmelParser
        jj_nt 1
    avrora.syntax.atmel.FileMarkingTokenManager
        jjimageLen 1
        jjnewStateCnt 1
        jjround 1
        jjmatchedPos 1
        jjmatchedKind 1
        curChar 1
        image 1
...
```

Listing 7.12: Sample output of JaSPEx-MLS's transaction statistics gathering mode for an execution of *Avrora* (Chapter 8), showing a transaction which failed to validate and had to be aborted. The transaction's read and write sets are divided by class, and for each class the number of read/write operations to its fields are shown. Additionally, and as this transaction failed to validate, the output also includes which fields failed to validate.

I start by presenting the life-cycle of a speculative task: how program execution begins, how new speculative tasks are created, how tasks are submitted for execution by the framework's thread pool, and how they are validated and committed.

Then, I introduce two novel techniques for improving thread pool usage efficiency: hybrid thread pool buffering and task freezing.

I then move on to describing the characteristics of JaSPEx-MLS's custom STM model: its two thread execution modes, how read-sets and write-sets are kept and validated, and the optimizations performed to the commit operation. Afterwards, I discuss the native integration of futures, return value prediction and captured memory.

I finish by presenting the profiling mode of the JaSPEx-MLS framework, that can be used to gather information to better guide the automatic parallelization process, and to help a programmer to manually optimize his application for MLS parallelization.

Chapter 8

Experimental Results

In this section, I present experimental results of application execution using the JaSPEx-MLS framework. The results presented were obtained on multiple machines, allowing my testing to cover multiple architecture design trade-offs on the multicore spectrum and their impact on my system: single vs multiple socket machines, less faster cores vs more slower cores, and boost vs no-boost.

The following machines were used during my testing:

- *M1*: Intel Core i7 *Haswell* 4770 with 16GB of RAM, running Ubuntu Linux 13.10 64-bit, stock speed 3.4GHz, boost speed up to 3.9GHz, 4 cores / 8 hardware threads (hyper-threading).
- *M2*: Intel Core i5 *Lynnfield* 750 with 8GB of RAM, running Ubuntu Linux 13.10 64-bit, stock speed 2.66GHz, boost speed up to 3.2GHz, 4 cores / 4 hardware threads.
- *M3*: Intel Xeon *Nehalem* E5520, dual-socket, with 24GB of RAM, running Ubuntu Linux 10.04.4 64-bit, stock speed 2.26GHz, boost speed up to 2.53GHz, 8 cores / 16 hardware threads (hyper-threading).
- *M4*: AMD Opteron *Magny-Cours* 6168, quad-socket, with 128GB of RAM, running CentOS 6.4 64-bit, stock speed 1.9GHz, no boost available, 48 cores / 48 hardware threads.
- *M5*: AMD Phenom II X6 *Thuban* 1055T with 12GB of RAM, running Ubuntu Linux 13.10 64-bit, stock speed 2.8GHz, boost speed up to 3.3GHz, 6 cores / 6 hardware threads.

Machines *M1*-*M3* support Intel Turbo Boost [Charles et al., 2009], and Machine *M5* supports AMD Turbo Core. Both boost implementations were left enabled during testing; because of this, processor frequency is usually faster for the sequential versions of applications than whenever JaSPEx-MLS is being used, as the framework's usage of multiple threads normally precludes boost modes from kicking in. Nevertheless, I believe that testing with this configuration makes the results more realistic and a better representation of expected end-user machine configurations.

The chosen benchmarks for testing were:

- *Avrora* [Titzer et al., 2005], a cycle-accurate simulator and analysis framework for AVR microcontrollers. I use the included `fib_massive.asm` test workload in single simulation mode.
- *Sunflow*,[1] a rendering engine that employs ray tracing. I use the included `aliens_shiny.sc` test scene, and disable Sunflow's internal threading support, leaving only a single-threaded renderer.
- *Series* (Fourier coefficient analysis), *MonteCarlo* (financial simulation) and *Euler* (Euler's flow equations solver) benchmarks from the Java Grande Forum (JGF) benchmark suite [Bull et al., 2000], using the `SizeB` workload. Other benchmarks from this suite were not considered due to being either too small (with sub-second execution times) or unsuitable for Method-Level Speculation, as they perform all computation in loops with no method calls, resulting in zero candidate spawn points.
- *Barnes-Hut*, a benchmark from the Lonestar benchmark suite [Kulkarni et al., 2009a] that simulates gravitational forces using the Barnes-Hut n-body algorithm.
- *Mandel* (Mandelbrot set renderer) and *Life* (Conway's Game of Life) benchmarks adapted from the Aparapi project.[2]

All benchmarks were run on the previously-described custom OpenJDK HotSpot JVM with support for continuations (Chapter 5). The sequential performance of this JVM closely follows Oracle's JVM builds, as they share the same codebase; this means that the measured execution run times for the unmodified applications are very similar to those observed by end-users executing these applications on the stock Oracle JVM. Unless otherwise stated, each benchmark was executed five times, and the results presented are the average of all five runs.

8.1 Bytecode Changes

In this section, I present a quantitative analysis of the bytecode changes done to each of the chosen benchmarks. Table 8.1 presents a static analysis, for each of the chosen benchmarks, of the number of classes and the changes performed by the JaSPEx-MLS framework. The total number of classes is approximate for some of the benchmarks, as they are part of larger suites that include other benchmarks. Third-party libraries are included in the class count, as they are also analyzed and modified by JaSPEx-MLS, but JDK classes are excluded, as they are not modified.

Avrora and *Sunflow* present the largest class count, containing hundreds of classes. Nevertheless, we see that at run-time, a much smaller number of application classes is requested by the VM (and subsequently modified and loaded): 45% for *Avrora* and 16% for *Sunflow*. Of these, the number of classes actually used tends to be even smaller, as a number of classes are only initialized as dependencies of other classes or singletons that are never used again.

Series, *MonteCarlo*, *Euler*, *Barnes-Hut*, *Mandel* and *Life* were designed to be more self-contained in execution, and have no dependencies on external libraries.

[1] *Sunflow - Global Illumination Rendering System*, http://sunflow.sourceforge.net/ (2007)
[2] *aparapi - API for data parallel Java*, https://code.google.com/p/aparapi/ (2013)

Benchmark	Total Classes	Avg. Size	Total Size	Modified Classes	Size Before	Size After	Gen. Classes	Gen. Size
avrora	927	2.689	2.492.341	415 (45%)	964.118	2.514.509	278	376.286
sunflow	637	2.826	1.800.235	103 (16%)	452.308	1.543.588	190	266.462
jgf series	10	1.189	11.890	6 (60%)	10.514	35.136	2	2.785
jgf montecarlo	21	2.151	45.161	18 (86%)	44.206	146.201	3	3.844
jgf euler	11	2.903	31.932	8 (73%)	30.997	127.139	6	7.773
barneshut	28	3.184	89.147	10 (36%)	30.453	77.102	2	3.344
aparapi mandel	5	1.818	9.090	5 (100%)	9.090	125.140	3	3.180
aparapi life	3	2.461	7.383	3 (100%)	7.383	117.766	2	2.758

Table 8.1: Measuring the total, modified and framework-generated number of classes for each of the chosen benchmarks. Sizes are presented in Bytes. Results include third-party libraries (if applicable) but exclude any JDK-included classes. The *total classes* column indicates the number of classes in the application's code base; the *average size* indicates the average size of an application class; the *total size* indicates the combined size of all of the application's unmodified classes; the *modified classes* column counts the number of application classes actually requested by the VM and modified by the JaSPEx-MLS framework during testing; the *size before* and *size after* columns indicate the combined size of the classes requested by the VM before and after the modifications performed by JaSPEx-MLS; the *generated classes* column counts the number of `Callables` and other helper classes generated by JaSPEx-MLS for each application, and, finally the *generated size* column counts their combined size.

For all the benchmarks, the many bytecode modifications performed in Chapter 6 are apparent in the size of the produced classes: on average, framework-modified classes are three times the size of the original classes. *Mandel* and *Life* are very big outliers with 14x-16x the original sizes; this happens because they rely on many JDK classes (as they are Swing GUI applications) and many overrides need to be added to safely allow non-transactional operations (Section 6.1.3).

The *Gen. Classes* column accounts for framework-generated `Callables`, and also extra classes used by the STM to transactionally access fields from JDK classes. For *Avrora* and *Sunflow*, the many classes analyzed by the JaSPEx-MLS framework lead to many `Callables` being generated. Note that, again, these are static numbers: in practice, most `Callables` are seldom used, and could be removed with profiling passes — but because they are so seldom used, their impact is also very small.

8.2 Bytecode Preparation Overheads

To characterize the overheads introduced by the changes detailed in Chapter 6, I compared the original sequential execution times for the benchmarks with those from executing the same benchmarks after bytecode modification, but still in single-threaded mode.

Figure 8.1 shows the results from testing using Machine *M1*; the results for other machines are very similar and as such were omitted. The *Transactional Execution* results only reflect the bytecode modifications needed for transactional execution (Chapter 6), whereas the *All Modifications* results include both transactional execution and the spawn point/future insertion (all of Chapter 6). For these tests, JaSPEx-MLS uses a thread pool with only a single thread, so no speculation is ever performed, and the fallback path for a spawn point is always taken (i.e. the creation of a new speculative task is always early rejected).

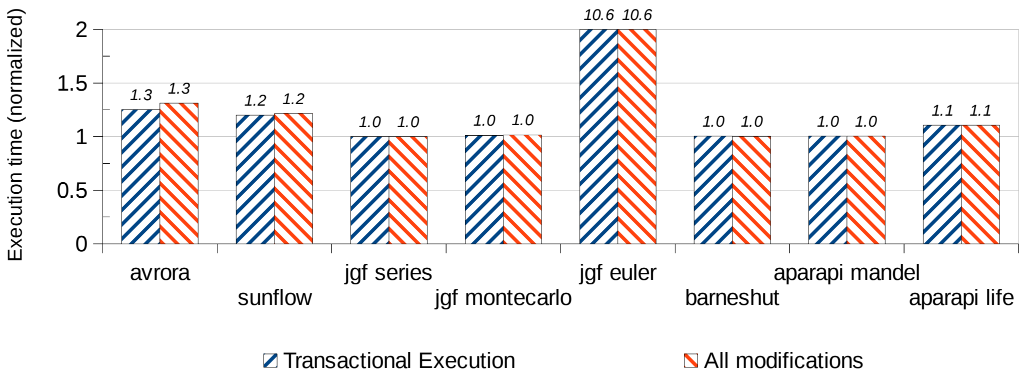


Figure 8.1: Impact of the bytecode modifications from Chapter 6 on sequential execution performance. Values are shown normalized to the execution time of the corresponding original sequential application. The *Transactional Execution* measurements include only the bytecode modifications from Section 6.1, whereas the *All Modifications* measurements include all bytecode changes described in Chapter 6. Apart from the *Euler* benchmark, measured execution times range from 1.0x to 1.3x the original application execution times.

The biggest impact to the execution time — 10.6x of the original application's execution time — happens in the *Euler* benchmark. This benchmark has very big methods with a very large number of array and field accesses, and the transactification of these methods, even without any other changes to inject spawn points, crosses the JIT compiler's maximum code size threshold, so the modified versions of methods are never properly compiled and optimized.

For the remaining benchmarks, the impact of the performed bytecode modifications ranges from 1.0x to 1.3x the original execution times. As we shall see in Section 8.3, this happens because most of the overheads are being removed by the OpenJDK VM's JIT compiler.

The presented results show that although the JaSPEx-MLS framework performs many changes to the applications' bytecode, non-speculative execution by the program-order thread is not severely impacted. This enables applications with a mix of parallel and sequential workloads to still benefit from the JaSPEx-MLS framework, as otherwise the slowdown caused by the extra overheads on sequential sections of the application would offset most of the gains from speculative parallelization.

8.3 Impact of OpenJDK HotSpot's Optimizing JIT Compiler

To analyze the impact of not using an optimizing JIT compiler, and to simulate the execution of the JaSPEx-MLS framework on top of a simpler virtual machine, Figure 8.2 shows the results of repeating the benchmarks from Figure 8.1 (that measured the added overhead from JaSPEx-MLS's bytecode transformations, without performing speculation) while disabling HotSpot's JIT compiler using the `-Xint` option, forcing the interpreter to be used for the entire program execution. Results are normalized to the corresponding runtime of the original unmodified applications when executed by HotSpot in interpreter-only mode.

The results from Figure 8.2 are completely changed from Figure 8.1: Execution times now reach up to 33x the original sequential code execution times, which in full speculative parallelization mode negates any gains from parallel execution. Only *Series* and *Mandel* are

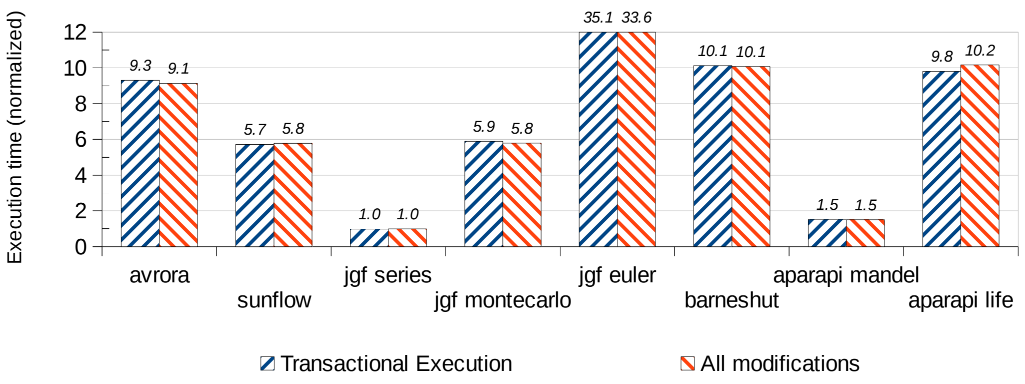


Figure 8.2: Impact of the bytecode modifications from Chapter 6 on sequential execution performance, when executed only using OpenJDK HotSpot's interpreter mode. Values are shown normalized to the run time of the corresponding original sequential application when executed by HotSpot in interpreter-only mode. The *Transactional Execution* measurements include only the bytecode modifications from Section 6.1, whereas the *All Modifications* measurements include all bytecode changes described in Chapter 6. In stark contrast with Figure 8.1, most applications are heavily affected: apart from *Series* and *Mandel*, other applications take at least 5x longer to execute, with some taking about 10x and *Euler* taking 33x. These results confirm my assertion that a production VM with an advanced JIT compiler is a crucial part of my approach, enabling speculative execution (and parallelization) with minimal overheads.

able to escape, as they are more compute-focused and thus suffer less from transactification overheads.

These results confirm my assertion that a production VM with an advanced JIT compiler is a crucial part of my approach, enabling speculative execution (and parallelization) with minimal overheads.

8.4 Characterizing the Impact of a Production VM

As part of my thesis statement, I proposed the development of a system that achieved practical speedup — that is, a system that achieved speedup when compared to commonly-used production virtual machines such as Oracle's OpenJDK HotSpot JVM.

My approach of building atop HotSpot to achieve practical speedup contrasts with past attempts at software-based automatic Java parallelization, which, as detailed in Section 2.4.3, build atop simpler research VMs, which are more amenable to the multiple changes needed for speculative execution.

In particular, I highlight two Java-based automatic speculative parallelization systems which build atop simpler VMs: SableSpMT [Pickett, 2012] and HydraVM [Saad et al., 2012].[3] SableSpMT is built atop the SableVM, which only provides an interpreter, and its author reports that it was unable to achieve any speedup over the original sequential SableVM performance. HydraVM is built atop Jikes JVM's lowest performance compiler (baseline), and was able to achieve speedups of up to 5x on the JOlden benchmarks on a machine with an 8-core multiprocessor.

[3]Both are described in more detail in Section 3.3.2.

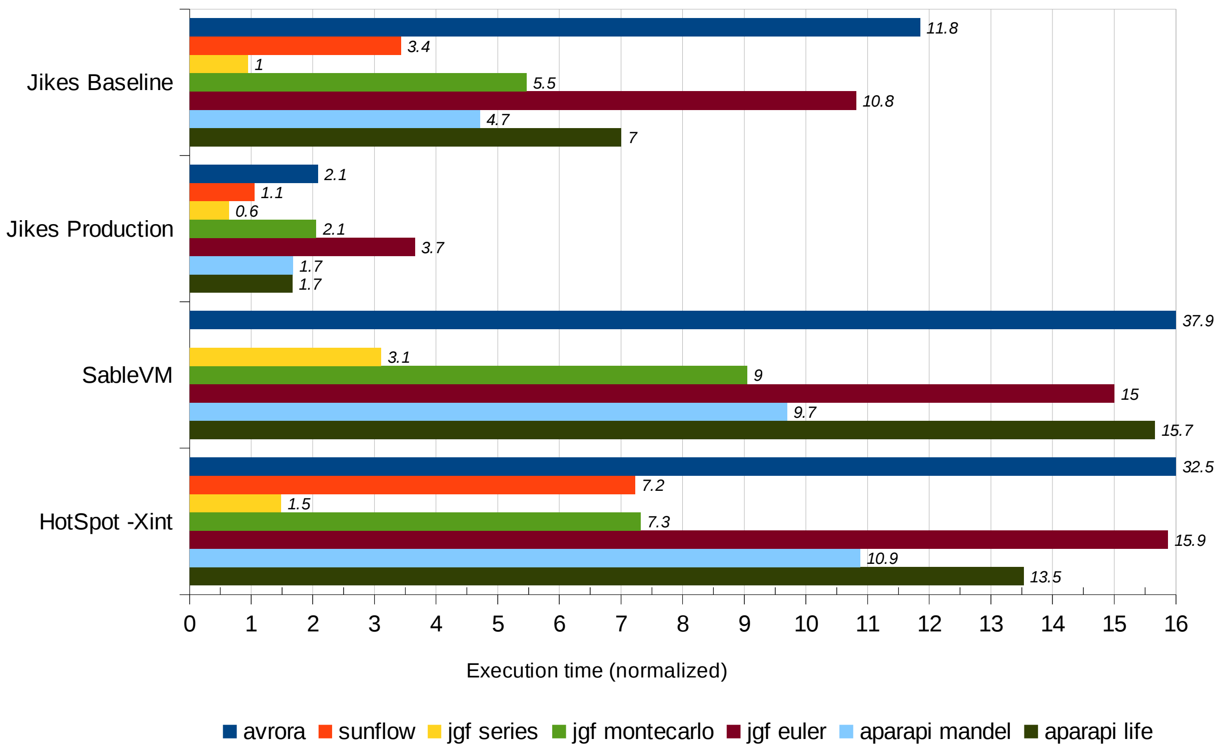


Figure 8.3: Testing the sequential versions of the chosen benchmarks with other Java Virtual Machines. Results are shown normalized to HotSpot's normal execution mode (not shown), and are grouped by VM configuration. Shown are the Jikes JVM in baseline compiler mode, Jikes JVM in its highest-performance production-profiled mode, SableVM using its default direct-threaded interpreter mode, and HotSpot in `-Xint` interpreter-only mode.

To better understand the sequential performance of both Jikes and SableVM, I tested each of my chosen benchmarks on those VMs. The results from this testing are shown in Figure 8.3. Five VM configurations were tested: the Jikes JVM in baseline compiler mode, the Jikes JVM in its highest-performance production-profiled mode, SableVM using its default direct-threaded interpreter mode, and OpenJDK HotSpot in both normal and `-Xint` interpreter-only modes.

The results are shown normalized to HotSpot's normal execution mode, and similarly to the previous sections they were benchmarked using Machine *M1*. As the results are normalized to HotSpot's execution time, values larger than 1.0 denote a VM is slower than HotSpot, and values smaller than 1.0 denote a VM is faster than HotSpot when executing a given benchmark.

As expected, other VMs cannot generally compete with HotSpot. Averaging the execution times for each of the VMs yields 6.32x for Jikes baseline, 1.83x for Jikes production, 15.06x for SableVM and 12.68x for HotSpot in `-Xint` mode; on average, all other VMs are slower than HotSpot's normal mode. In fact, except for the *Series* benchmark, Jikes in baseline mode and SableVM are always slower than HotSpot, with execution times going from 3.4x all the way up to 37.9x when compared to the production VM.

As both SableVM and HotSpot in `-Xint` mode employ an interpreter, their results are quite close, with SableVM having better performance on 2 out of the 6 benchmarks that were tested.

Whereas Jikes JVM's highest-performance mode is the faster of the group — and the only that is able to beat HotSpot in a single instance — for 5 out of the 7 benchmarks it is slower

Benchmark	Jikes baseline	Speedup (3.5x J.b. speedup)	Speedup (5x J.b. speedup)
avrora	11.85	0.30	0.42
sunflow	3.43	**1.02**	**1.46**
jgf series	0.95	**3.68**	**5.26**
jgf montecarlo	5.47	0.64	0.91
jgf euler	10.82	0.32	0.46
aparapi mandel	4.72	0.74	**1.06**
aparapi life	7.01	0.50	0.71

Table 8.2: Simulated speedup of HydraVM, when compared to HotSpot, when considering a speedup of both 3.5x and 5x the Jikes baseline compiler's performance. Only the highlighted values represent speedup, all others are expected to be slower than HotSpot; of those faster, two of them are only so by a small margin (2% and 6%).

than HotSpot by 70-270%.

The *Barnes-Hut* benchmark was excluded from the testing as neither Jikes nor SableVM were able to execute it, and, in addition, SableVM only supported the 12-year-old Java 1.4, which required recompilation of most of the test code, including minor porting changes where possible; *Sunflow* made extensive use of Java 5 features and as such was also not able to be executed by SableVM.

Going back to the other similar Java-based parallelization systems, the SableSpMT VM was never able to achieve a speedup. By considering a different baseline for comparison — an execution where every speculation fails to commit at the end — the author reported a mean speedup of 1.34x. This means that it would be very hard for SableSpMT to ever achieve practical speedup. On 5 out of 6 of the tested benchmarks, even a respectable 9x speedup would still not be enough to overtake HotSpot.

The authors of HydraVM reported a speedup from 2-5x when compared to Jikes JVM's baseline compiler. To simulate how HydraVM's performance can be compared to HotSpot, we can take their average speedup of 3.5x, and divide each of the execution times obtained with Jikes baseline compiler by that value.[4] Doing so reveals that HydraVM would obtain practical speedup when comparing to HotSpot for only 2 of the tested benchmarks: 1.02x for *Sunflow* and 3.68x for *Series*. If, instead, we considered the best-case scenario of a 5x speedup, this would translate into a speedup of 1.46x for *Sunflow*, 5.26x for *Series*, and 1.06x for *Mandel*. The full results are shown in Table 8.2.

Thus, we see that the impact of a production VM on obtaining practical speedup is very relevant: for many simpler systems, a great deal of parallelism has to be extracted from a sequential application before any speedup over a VM such as HotSpot can be achieved.

It its not impossible, however, for the speculation techniques proposed by the authors of SableSpMT and HydraVM to be applied to a production VM, but this would require a very large engineering effort that thus far has yet to be tackled. The integration of speculation techniques with a number of modern VM optimizations and features — such as complex garbage collection algorithms — also remains an open research question. In practice, no working proof-of-concept for this kind of complex integration has yet been published.

[4]Notice that the execution time metric is the inverse of the speedup. This means that a speedup of 2x would translate to 1/2 the execution time.

8.5 Benchmarking Task Buffering and Task Freeze

In Chapter 7, I proposed two novel techniques for optimizing the use of machine execution resources: hybrid thread pool buffering and task freezing. Hybrid thread pool buffering safely enables the use of task buffering when combined with JaSPEx-MLS's flexible task spawn model. Task freezing allows threads hosting speculative tasks that are awaiting uncomputed values to suspend and snapshot those tasks; the snapshotted tasks are queued for later execution after the needed values are available, and their hosting threads are freed to work on other tasks.

In this section, I present results from speculatively parallelized executions of the chosen benchmarks in three different configurations: only task buffering enabled, with freezing disabled; only freezing enabled, with task buffering disabled; and both task buffering and task freezing enabled. These results are normalized to a fourth configuration, where both task buffering and task freezing are disabled. The objective of these benchmarks is to measure the impact of these features in the performance of the JaSPEx-MLS framework both in isolation and when used together.

The results from testing using Machine *M1* are shown in Figure 8.4. Results above 1.0 denote relative speedup versus a JaSPEx-MLS configuration not using buffering nor freezing, and those below 1.0 denote relative slowdown.

The *Avrora* benchmark is not able to take advantage of task buffering, as it always leads to deadlocks and to the thread pool disabling buffering at run-time. Nevertheless, task freezing alone is able to improve performance for this benchmark.

The *Sunflow* benchmark also suffers from deadlocks when task buffering is used, but — and in contrast with the *Avrora* benchmark — when buffering is combined with freezing no deadlocks are observed. Unfortunately, both task buffering and task freezing have no impact on this benchmark.

The *Series* benchmark shows a very good improvement from using task buffering. We also see that when used alone, task freezing has little impact on this benchmark. In contrast, when task freezing is combined with task buffering, this feature is able to improve on task buffering's performance boost, yielding a speedup of 1.62x in the 4t configuration and 1.68x in the 4t + 4ht configuration.

The *MonteCarlo* benchmark shows little impact from either buffering or freezing in the 4t configuration. When hyper-threading is used, and similarly to what we will see with *Barnes-Hut*, the usage of buffering actually reduces performance for this benchmark. This happens because with hyper-threading, two hardware threads share a single processor core, which has been shown to, in some cases, cause performance loss [Tuck and Tullsen, 2003].

The *Euler* benchmark is not impacted by task buffering nor task freezing.

The *Barnes-Hut* benchmark shows very good improvement from using buffering in the 4t configuration, whereas freezing has a detrimental effect when used alone and only a slight effect when used in combination with buffering. Similarly to the *MonteCarlo* benchmark, the usage of task buffering in combination with hyper-threading has a detrimental effect to performance.

The *Mandel* benchmark shows a small improvement from both features, yielding a combined 1.04x speedup in the 4t configuration and a 1.06x speedup on the 4t + 4ht configuration.

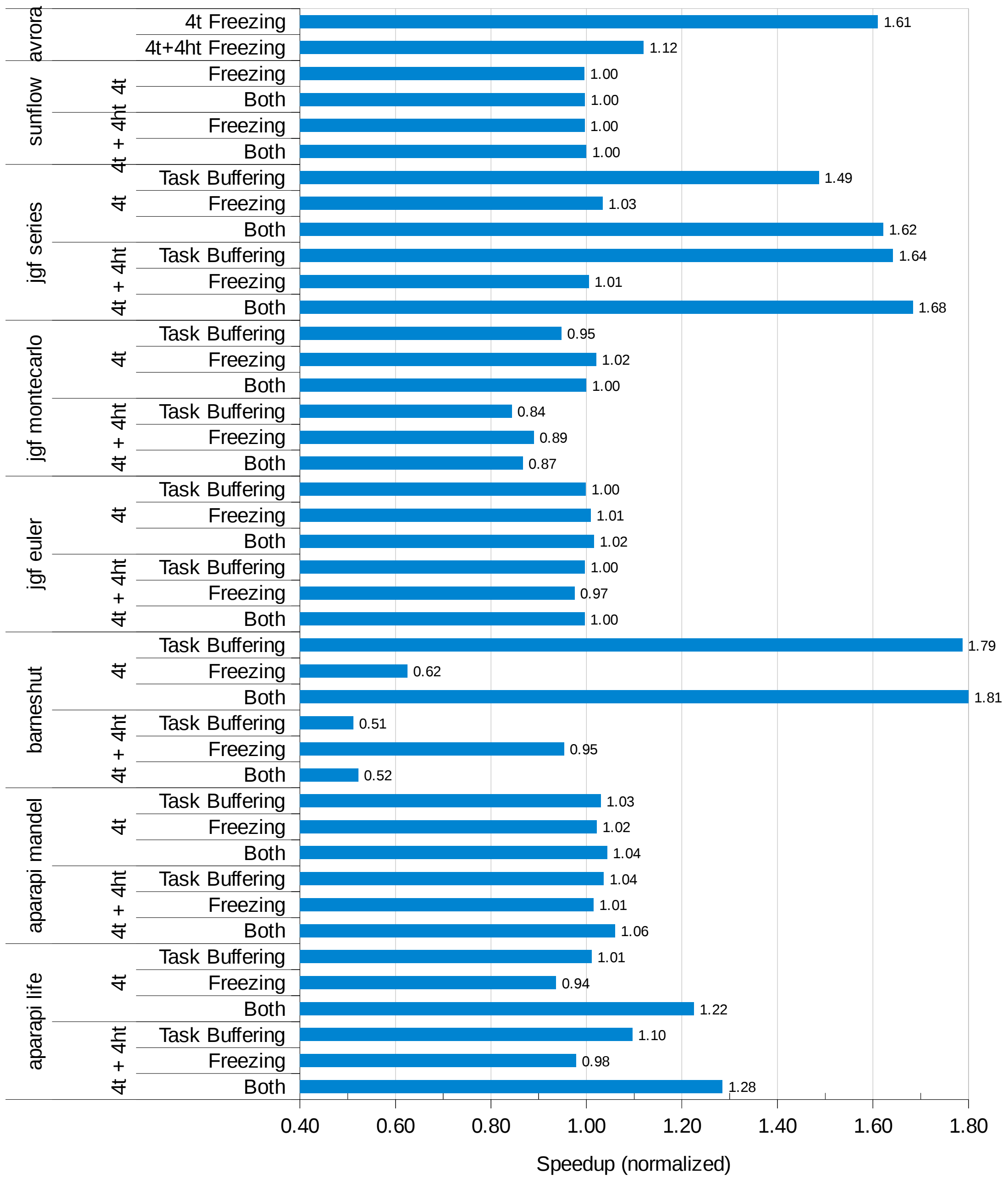


Figure 8.4: Benchmarking hybrid thread pool buffering and task freeze. Three different configurations are tested: only task buffering, with freezing disabled; only freezing, with task buffering disabled; and both task buffering and task freezing enabled; these results are normalized to a fourth configuration, where both task buffering and task freezing are disabled. For each benchmark, two thread configurations are used: 4 threads and 4 threads + 4 hyper-threads.

The *Life* benchmark once again shows that while task buffering alone improves performance for most benchmarks, freezing is only truly useful when combined with buffering. Especially in the hyper-threading configuration, we see that, when used alone, buffering led to a 1.10x speedup and freezing actually caused a slight slowdown, whereas their combined usage delivers a 1.28x speedup.

8.6 Speculative Parallelization Benchmarks

In this section, I present results from measuring the speedup of executing each benchmark while being parallelized by the JaSPEx-MLS framework. The results presented use the best framework settings for each benchmark at each data point. Note that due to the heterogeneity of core counts and hyper-threading options, some core configurations are only presented for a subset of the machines. For each benchmark, I first test with no speculation (as graphed in Figure 8.1 and Figure 8.2 under *All Modifications*), and then add results for the different thread (denoted by *t*) and hyper-threading (denoted by *ht*) configurations. The presented results are normalized to the execution time of the original unmodified sequential application on each of the test machines.

The *Avrora* benchmark (Figure 8.5) performs some speculation, but task size ends up being very unbalanced, and the observed slowdown comes from speculative execution overheads on very large tasks that end up dominating the running time.

The *Sunflow* benchmark (Figure 8.6) ends up being mostly single-threaded. While *Sunflow* itself has built-in support for multithreading, JaSPEx-MLS is currently not able to extract much parallelism from the (single-threaded) unmodified version without some refactoring to make it more MLS-friendly, and no speedup is attained in this benchmark. Unlike *Avrora*, most of the single-threaded work is performed in non-speculative mode, and as such the observed slowdown is in line with the expected transactification and framework overheads.

The *Series* benchmark (Figure 8.7) yields good scalability up to 6 and 8 thread configurations, reaching a speedup of around 4x for most machines, with a maximum of 4.6x on *M1*. After 8 threads, no more parallelism is able to be extracted from the application, and configurations with more than 8 threads mostly yield the same or worse performance results.

The *MonteCarlo* benchmark (Figure 8.8) is heavily optimized for sequential performance, avoiding object allocation by reusing many objects, which leads to a very high abort rate due to failed STM validations — around 25%, even after profiling. Too much time is lost on re-executions, penalizing the application's run time, and the extracted parallelism is not able to compensate this impact. Because of this, the JaSPEx-MLS framework is not able to attain speedup in this benchmark.

The *Euler* benchmark (Figure 8.9), as expected from the results of the bytecode modification overhead analysis, is not able to recover from the massive slowdown imposed by the JIT compiler declining to optimize the application's methods, as evidenced by the poor results even in the *nospec* configuration, which on other benchmarks is usually ranged 0.8-1x.

The *Barnes-Hut* benchmark (Figure 8.10) yields very interesting results: In this benchmark, all three Intel-based machines (*M1-M3*) are able to attain a speedup of around 1.6x, while both AMD-based machines (*M4* and *M5*) are never able to even match the performance of the original

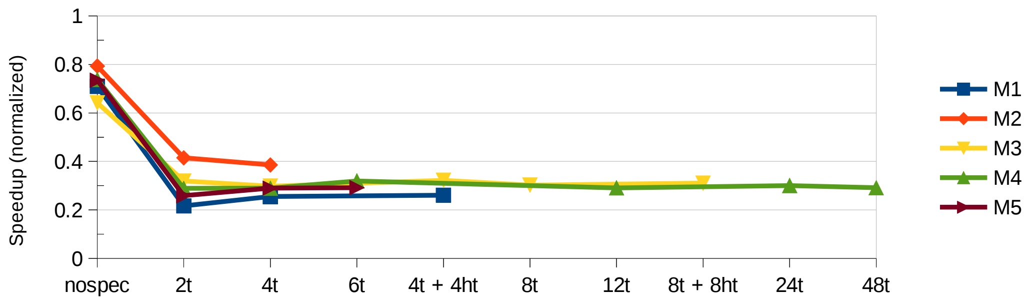


Figure 8.5: Speedup for the *Avrora* benchmark. In this benchmark, none of the test machines were able to attain a speedup.

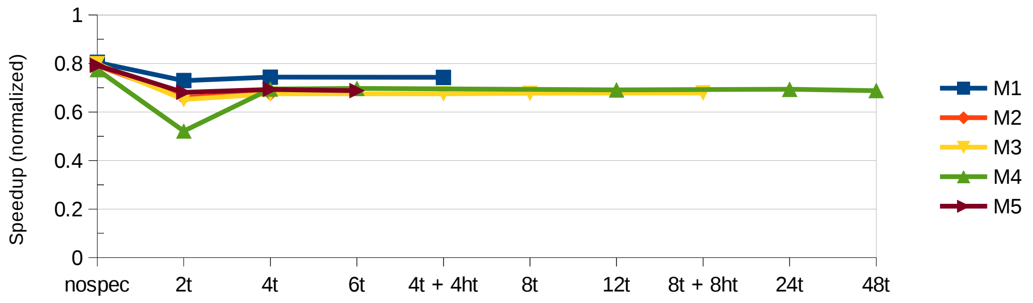


Figure 8.6: Speedup for the *Sunflow* benchmark. In this benchmark, most of the computation ends up being single-threaded, and none of the test machines were able to attain a speedup.

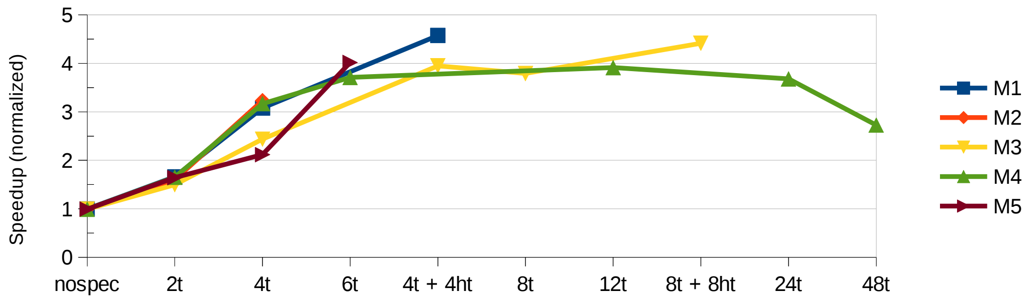


Figure 8.7: Speedup for the *Series* benchmark. The JaSPEx-MLS framework is able to achieve a speedup of up to 4.6x in this benchmark.

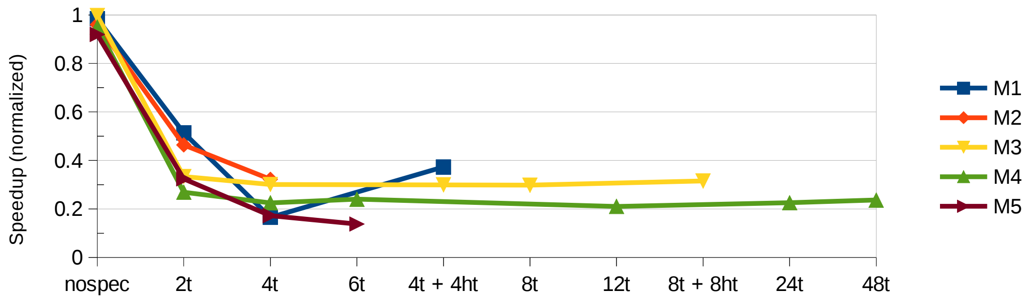


Figure 8.8: Speedup for the *MonteCarlo* benchmark. This benchmark attains no speedup due to a very high abort rate originating from heavily-optimized sequential code.

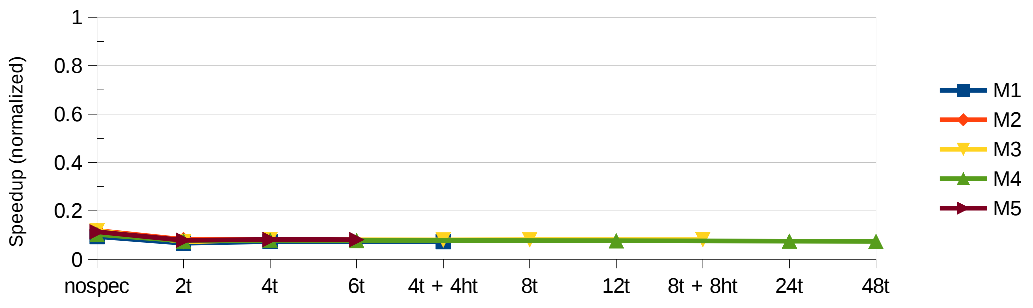


Figure 8.9: Speedup for the *Euler* benchmark. No speedup is attained in this benchmark, as the bytecode modification outputs methods that cross the JIT's maximum code size threshold (notice the impact on the *nospec* configuration).

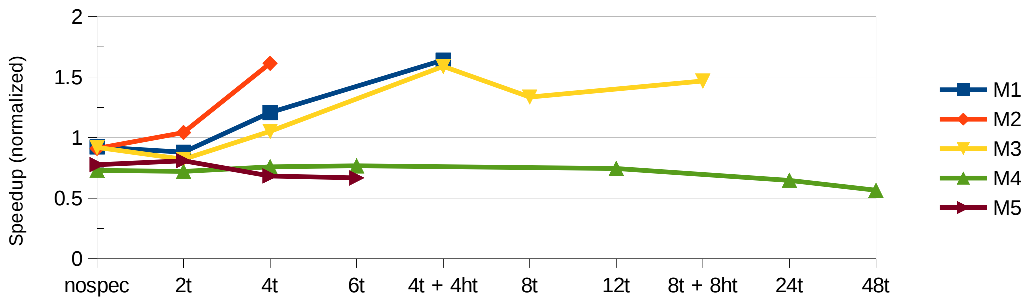


Figure 8.10: Speedup for the *Barnes-Hut* benchmark. Intel-based machines reach a speedup of about 1.6x on this benchmark, while AMD-based machines attain no speedup.

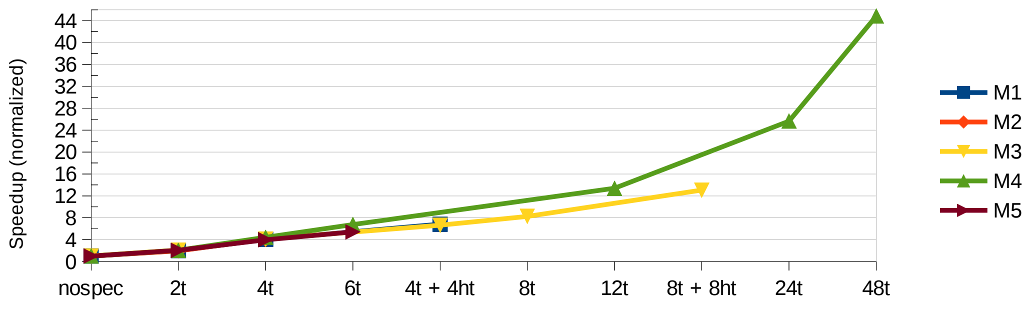


Figure 8.11: Speedup for the *Mandel* benchmark. This benchmark is able to scale almost linearly with the number of CPU cores.

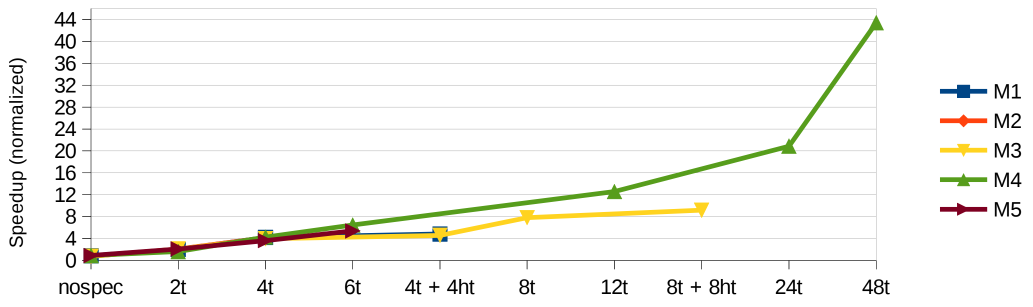


Figure 8.12: Speedup for the *Life* benchmark. This benchmark is able to scale almost linearly with the number of CPU cores.

application. Intel's architectures are clearly superior to AMD's in this test, highlighting both the difference in microarchitecture implementation and the potential for future improvements on AMD's part.

Finally, both *Mandel* (Figure 8.11) and *Life* (Figure 8.12) implement easily-parallelizable workloads (the original workloads were aimed at GPU architectures), that scale almost linearly with the number of CPU cores on the test machines, both reaching a speedup of around 44x on the 48-core test machine.

8.7 Summary

In this chapter I present various experiments to characterize both the overheads imposed on application execution, and the performance extracted from sequential applications by the JaSPEx-MLS framework.

I start with a quantitative study of the bytecode changes performed to classes, followed by measurements of the execution overheads imposed by those changes, and ending with a study of the impact of an optimizing VM on the imposed overheads. Afterwards, I test the execution performance of two alternative VMs used as basis by other Java-based speculative systems, and show that it is very hard to obtain practical speedup with these simpler VMs. I also present measurements of the impact of both hybrid thread pool buffering and task freezing in improving the performance of MLS parallelization.

Finally, I present individual speculative parallelization results for a number of applications on several machines with varying architectures and core counts. These results show that JaSPEx-MLS is able to extract parallelism from several of the chosen benchmarks, even when considering a normal execution by the OpenJDK HotSpot JVM as a baseline.

Chapter 9

Conclusions

In this final chapter of my dissertation, I summarize the main contributions of my work, followed by a discussion of future research ideas that originate from it.

9.1 Main Contributions

The motivation for this dissertation is that while new applications are being written to target multi and manycore machines, further technological advances are being held back by the large body of existing sequential and lightly-threaded applications. The emergence of automatic parallelization systems is a path to alleviating this issue, but, because no parallelization system is one-size-fits-all, different solutions are needed for different application domains.

In this dissertation, I concentrated on one domain that I felt posed unsolved challenges: that of object-oriented applications running atop managed platforms. I proposed the creation of an automatic parallelization framework that could achieve practical performance, and that would be able to compete with production managed runtimes.

The proposed integration of the parallelization framework not as part of but instead mostly on top of a production VM is the distinguishing feature of this work. Earlier attempts at parallelization on managed platforms, especially software-based ones, were hampered by their hosting platform: they either dived into a complex VM and got lost there (leaving mostly vague clues and anecdotal evidence behind), or chose simpler VMs which are very amenable to modifications but in no way reflect the inner workings of a complex VM.

As such, the effort in this work involved both the creation of novel techniques for speculative parallelization of applications and their highly-tuned implementation in the form of the JaSPEx-MLS parallelization framework.

From this effort, I distinguish the following main contributions:

- The creation of a new Software Transactional Memory model specifically aimed at speculative execution. This model includes support for futures, allowing them to replace concrete values on the stack, on the heap, and on arrays. Futures are read transparently by program code, with the STM performing any needed verifications and synchronization. This flexibility enables further parallelism to be extracted from sequential applications.

  This model also distinguishes between the thread running non-speculatively in program-order, and the remaining speculative threads on the system. This enables safe integration between transactional and non-transactional parts of the system, while at the same time allowing for relaxed isolation between them — enabling value forwarding and lower-overhead execution.

This model natively integrates support for Return Value Prediction. Predictions and their validation are integrated as part of the STM algorithm, and predictors are updated by the STM during transaction commit operations.

Finally, this STM model enables the novel combination of captured memory with Method-Level Speculation, allowing instrumentation-free access to objects created by the current transaction, while still allowing safe concurrent access and value forwarding to other speculative transactions.

- The high-performance implementation of the proposed STM model. This implementation allows very low-overhead instrumentation, with transactified code running between 75% to 100% of the original application code's speed. No ongoing memory overhead is added to objects, as all metadata is kept in thread-local structures, making memory footprint minimal.

- The creation of a novel algorithm for automatically transforming method invocations into spawn points that return futures, and to avoid overeager insertion of spawn points. This algorithm takes into account the various possible control flows that may be followed through a method, and is able to generate a verifiable-consistent class file that is accepted by the VM. Together with the introduced STM model, this allows for speculation of method execution without relying on value prediction nor needing explicit refactoring of program code.

- The extension and generalization of previous work on VM-based continuations on top of the OpenJDK HotSpot JVM, used in this dissertation as a basis for Method-Level Speculation, and that has already enabled further research on speculation in the context of distributed Software Transactional Memories [Fernandes, 2011, Peluso et al., 2012].

  Furthermore, the alternative reflective invocation mechanism developed as part of this VM extension has also been extracted into a library that is used to perform bytecode transformation by recent versions of the JVSTM[1] and the Fénix Framework,[2] which power the FenixEdu project.[3] FenixEdu is used daily by thousands of students across multiple Universities and Schools.

- The identification and solution of issues pertaining to the automatic transformation of bytecode to execute safely with transactional semantics. These include not only the interception of heap accesses, but also the identification of non-transactional, native, and dangerous operations, that if allowed to execute without proper protection and synchronization may cause deviations from the program's original sequential semantics.

- The introduction of a novel technique to support safe buffering of speculative tasks in the context of Method-Level Speculation: hybrid thread pool buffering. This technique allows the safe buffering of MLS tasks which must respect string ordering constraints imposed

[1] *JVSTM - Java Versioned STM*, http://inesc-id-esw.github.io/jvstm/
[2] *Fénix Framework*, http://fenix-framework.github.io/
[3] *FenixEdu*, http://fenixedu.org/

by the original sequential semantics, while avoiding deadlocks that uncontrolled buffering may cause.

- The introduction of a novel technique to allow thread reuse in the context of Method-Level Speculation: task freezing. This technique allows threads that are hosting speculative tasks that are awaiting uncomputed values to suspend and snapshot those tasks. Snapshotted tasks are then queued for later execution after the needed values are available. After a snapshotted task is queued, the thread originally hosting the task is free to work on other tasks, improving system efficiency.

All the above contributions, although usable stand-alone, were designed to complement each other and work together in the creation of a software-based speculative parallelization framework using Method-Level Speculation: the JaSPEx-MLS framework. This highly-optimized software-based framework targeted at commodity hardware and built atop the OpenJDK HotSpot JVM is able to take Java or JVM sequential applications, analyze and prepare them for speculative parallelization, optionally profile them to guide parallelization decisions, and afterwards coordinate their concurrent execution so as to achieve speedups over the original sequential versions when executed by Oracle's own stock JVM.

As part of my work, but not used by the current version of the JaSPEx-MLS framework (and thus not included in this dissertation), I also submitted a number of contributions to the JVSTM, of which I highlight VArray [Anjo and Cachopo, 2011a]: a lightweight versioned transactional array aimed at read-dominated workloads. VArray provides lock-free read operations, per-element tracking of conflicts, and access to older versions, while minimizing the in-memory footprint of the array and the overhead of read operations. Internally, VArray uses both a native array containing the latest versions for each array element and a log structure that keeps and enables the lookup of older element versions. At commit time, older element versions are transferred from the native array into the log, and any new writes take their place in the native array.

9.2 Future Research

The JaSPEx-MLS framework as presented in this dissertation works and has been tested with a number of Java and JVM applications. Application parallelization, and especially automatic parallelization continues to be a field of very active research, as technologies shift and methodologies evolve, and as I stated before, there is no one-size-fits-all solution to the posed challenges.

The following are (in no particular order) a list of ideas for future improvements and research directions that extend from this dissertation. This list is not exhaustive: it is a list of topics that I believe would be the most interesting and fruitful targets for future study.

- *Integration of support for first-class continuations as part of the Java VM specification.* Support for VM-based continuations is a key building block for the JaSPEx-MLS framework, without which practical speedups would not have been achievable. Unfortunately, the VM-based continuation implementation described in Chapter 5 and its companion `contlib` library still depend on non-standard experimental changes to the VM.

Whereas the Java programming language does not include support for continuations — as normally the Java programming language and the JVM Platform specification advance hand-in-hand — other JVM-hosted languages such as Scala or Ruby do include support for them. Even though Scala is able to implement limited continuations by having its compiler generate the needed entry points, JRuby is not able to support continuations as defined as part of the Ruby language.

Other web-development frameworks on top of Java such as RIFE and Jetty also employ different takes on the idea of continuations, with bytecode transformation (and their accompanying large overheads) being the most common implementation for them.

By extending the JVM Platform with support for continuations, better performance could be offered to approaches currently employing bytecode, and JaSPEx-MLS's dependency on non-standard VM features could be avoided.

- *Integration of support for transactional execution as part of the Java VM specification.* While I believe that JaSPEx-MLS's design of working on top of the VM as much as possible is sound, the process of transactification could be made simpler if the VM offered a low-level API on top of which transactional execution could be layered. A transactional execution API would benefit both speculative parallelization approaches such as JaSPEx-MLS's and also Software Transactional Memory implementations, avoiding the hurdles and sidestepping of VM safety that the current approaches depend on.

- *Integration with recently-released commercial Hardware Transactional Memory.* One of the biggest sources of speculative execution overhead is the use of Software Transactional Memory. Recent CPU designs from Intel and IBM (Section 4.1.2) include support for Hardware Transactional Memory. Whereas TM systems in general and JaSPEx-MLS in particular should not strictly depend on these very early hardware implementations, especially considering their currently best-effort scope and the need for still using STM as a fallback, their optional integration presents very interesting future research.

  While an early experiment I performed[4] shows that it is possible to use HTM without support from the Java VM, it also shows the limits of this approach, and strongly suggests that VM integration would be needed for proper HTM support. Like the proposed continuations support above, this HTM support would be especially valuable if was integrated as part of the JVM Platform standard.

- *Integration of loop-based parallelization techniques.* Whereas the main focus for the JaSPEx-MLS framework was on supporting object-oriented applications, the framework is not tightly bound to Method-Level Speculation and could be used to host other parallelization techniques, especially those that target loops, opening up JaSPEx-MLS to a breadth of new applications.

- *Development of replacement data structures optimized for speculative-parallelization.* Many applications rely on standard data structure implementations present on the basic Java JDK library. These data structures are not speculation-friendly (nor, in most cases, concurrency-friendly), and as such it would be very interesting to explore the usage of the JaSPEx-MLS framework's support for method and class replacement described

[4] *Java library for Restricted Transactional Memory*, https://github.com/ivoanjo/javartm (2013)

in Section 6.1.5 to additionally replace JDK-provided data structures with speculation-friendly alternative versions.

- *Improved guidance for developers.* Whereas the JaSPEx-MLS framework provides some tools (described in Section 7.5) that help to optimize an application for automatic parallelization, these are still very low-level and not very programmer-friendly. In the future, it would be very interesting to develop a tool that could parse the statistics gathered at run-time by JaSPEx-MLS, and better guide an application developer to perform minor changes that would enable the framework to extract further parallelism.

9.3 Published Work

The following publications were produced as a result of my research work:

- [Anjo and Cachopo, 2010] and companion poster, presented at PPPJ 2010, describes the objectives of the RuLAM project, and provides a small introduction to the JaSPEx framework, including transactification, speculation, and applications being targeted.

- [Anjo and Cachopo, 2011b], presented at the WANDS 2011 workshop, examines the challenges of performing speculative execution of Java applications without modifying the Java virtual machine: What is possible, what is missing, and what would be nice to have. It also surveys how issues such as transactification, continuations, non-transactional operations and scheduling can be dealt with.

- [Anjo and Cachopo, 2011a], presented at ICA3PP'11, focuses on transactional arrays: How they are currently implemented, and how they can be improved, with and without collaboration from the STM. It introduces a novel lightweight transactional array algorithm implemented on top of the JVSTM — VArray — that supports versioning, while still providing memory overheads comparable to native Java arrays, and read operation performance superior to previous alternatives.

- [Anjo and Cachopo, 2013a], presented at LCPC 2012, provides an introduction to the JaSPEx-MLS framework. It describes part of the bytecode modifications performed that allow Java bytecode to execute transactionally, and it details the transformation of method call sites into speculative task spawn points. It then examines the run-time management of spawned tasks, and the techniques employed by the framework's custom Software Transactional Memory model.

- [Anjo and Cachopo, 2013b], presented at ICA3PP'13, presents a number of techniques aimed at improving performance and efficiency of parallel speculation systems: it introduces *hybrid thread pool buffering*, *task freezing*, and STM-assisted Return Value Prediction and the results of their implementation on the JaSPEx-MLS framework.

- "*Design of a Method-Level Speculation Framework for Boosting Irregular JVM Applications*", accepted and awaiting publication on Elsevier's *Journal of Parallel and Distributed Computing*, describes the JaSPEx-MLS framework in detail: how application bytecode is modified to behave transactionally, how to handle non-transactional and dangerous operations, how to transform method call sites into spawn points to be used for Method-Level Speculation, how speculative tasks are created and managed at run-time, and presents JaSPEx-MLS's custom Software Transactional Memory model with support for futures, captured memory, and Return Value Prediction.

Bibliography

A. Adl-Tabatabai, T. Shpeisman, and J. Gottschlich. Draft specification of transactional language constructs for C++, version 1.1, 2012.

B. Alpern, C. Attanasio, J. Barton, M. Burke, P. Cheng, J. Choi, A. Cocchi, S. Fink, D. Grove, M. Hind, S. Hummel, D. Lieber, V. Litvinov, M. Mergen, T. Ngo, J. Russell, V. Sarkar, M. Serrano, J. Shepherd, S. Smith, V. Sreedhar, H. Srinivasan, and J. Whaley. The Jalapeño virtual machine. *IBM Systems Journal*, 39(1):211–238, 2000.

I. Anjo. JaSPEx: Speculative parallelization on the Java platform. Master's thesis, Instituto Superior Técnico, 2009.

I. Anjo and J. Cachopo. JaSPEx: Speculative parallel execution of Java applications. In *Proceedings of the Simpósio de Informática (INFORUM 2009)*. Faculdade de Ciências da Universidade de Lisboa, 2009.

I. Anjo and J. Cachopo. RuLAM Project: Speculative parallelization for Java using software transactional memory [extended abstract]. In *8th International Conference on Principles and Practice of Programming in Java (PPPJ 2010)*. Vienna University of Technology, 2010.

I. Anjo and J. Cachopo. Lightweight transactional arrays for read-dominated workloads. In *Proceedings of the 11th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'11)*, volume II, pages 1–13. Springer-Verlag, 2011a.

I. Anjo and J. Cachopo. Speculating on top of an unmodified Java VM. In *2011 Workshop on Wild and Sane Ideas in Speculation and Transactions*, 2011b.

I. Anjo and J. Cachopo. A software-based method-level speculation framework for the Java platform. In *Proceedings of the 25th International Conference on Languages and Compilers for Parallel Computing (LCPC 2012)*, pages 205–219. Springer-Verlag, 2013a.

I. Anjo and J. Cachopo. Improving continuation-powered method-level speculation for JVM applications. In *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'13)*, pages 153–165. Springer-Verlag, 2013b.

D. Baptista. Task scheduling in speculative parallelization. Master's thesis, Instituto Superior Técnico, 2011.

J. Barreto, A. Dragojević, P. Ferreira, R. Filipe, and R. Guerraoui. Unifying thread-level speculation and transactional memory. In *Proceedings of the 13th International Middleware Conference (Middleware '12)*, pages 187–207. Springer-Verlag, 2012.

L. Baugh and C. Zilles. An analysis of I/O and syscalls in critical sections and their implications for transactional memory. In *Proceedings of the IEEE 2008 International Symposium on Performance Analysis of Systems and Software (ISPASS '08)*, pages 54–62. IEEE Computer Society, 2008.

W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, and T. Lawrence. Parallel programming with Polaris. *Computer*, 29(12):78–82, 1996.

R. Blumofe and C. Leiserson. Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5):720–748, 1999.

C. Blundell, E. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17, 2006a.

C. Blundell, E. Lewis, and M. Martin. Unrestricted transactional memory: Supporting I/O and system calls within transactions. Technical report, Department of Computer & Information Science, University of Pennsylvania, 2006b.

E. Bruneton, R. Lenglet, and T. Coupaye. ASM: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 2002.

J. Bull, L. Smith, M. Westhead, D. Henty, and R. Davey. A benchmark suite for high performance Java. *Concurrency Practice and Experience*, 12(6):375–388, 2000.

V. Bushkov, R. Guerraoui, and M. Kapałka. On the liveness of transactional memory. In *Proceedings of the 2012 ACM Symposium on Principles of Distributed Computing (PODC '12)*, pages 9–18. ACM, 2012.

J. Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Instituto Superior Técnico, 2007.

J. Cachopo. Do we really need parallel programming or should we strive for parallelizable programming instead? In *SPLASH 2010 Workshop on Concurrency for the Application Programmer*, 2010.

J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Science of Computer Programming*, 63(2):172–185, 2006.

H. Cain, M. Michael, B. Frey, C. May, D. Williams, and H. Le. Robust architectural support for transactional memory in the Power architecture. *SIGARCH Computer Architecture News*, 41 (3):225–236, 2013.

B. Carlstrom, J. Chung, H. Chafi, A. McDonald, C. Minh, L. Hammond, C. Kozyrakis, and K. Olukotun. Transactional execution of Java programs. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

B. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. Minh, C. Kozyrakis, and K. Olukotun. The Atomos transactional programming language. In *Proceedings of the 2006 Conference on Programming Language Design and Implementation (PLDI 2006)*, volume 41, pages 1–13. ACM, 2006.

F. Carvalho and J. Cachopo. Lightweight identification of captured memory for software transactional memory. In *Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP'13)*, pages 15–29. Springer-Verlag, 2013.

J. Charles, P. Jassi, N. Ananth, A. Sadat, and A. Fedorova. Evaluation of the Intel Core i7 turbo boost feature. In *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC '09)*, pages 188–197. IEEE Computer Society, 2009.

M. Chen and K. Olukotun. Exploiting method-level parallelism in single-threaded Java programs. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques (PACT-1998)*, pages 176–184. IEEE Computer Society, 1998.

M. Chen and K. Olukotun. The Jrpm system for dynamically parallelizing Java programs. *ACM SIGARCH Computer Architecture News*, 31(2):434–446, 2003.

C. Click. A JVM Does That???, 2010. URL http://www.azulsystems.com/blog/wp-content/uploads/2011/03/2011_WhatDoesJVMDo.pdf.

G. Costanza. Speculative multithreading in a Java virtual machine. Master's thesis, Institute of Computer Systems, Swiss Federal Institute of Technology (ETH), 2007.

P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, pages 336–346. ACM, 2006.

J. Danaher. The JCilk-1 runtime system. Master's thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2005.

J. Danaher, I. Lee, and C. Leiserson. The JCilk language for multithreaded computing. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*, 2005.

D. Dice, O. Shalev, and N. Shavit. Transactional locking II. *Distributed Computing*, pages 194–208, 2006.

D. Dice, Y. Lev, M. Moir, D. Nussbaum, and M. Olszewski. Early experience with a commercial hardware transactional memory implementation. Technical Report SMLI TR-2009-180, Sun Microsystems, Inc., 2009.

N. Diegues and J. Cachopo. Review of nesting in transactional memory. Technical Report RT/1/2012, INESC-ID, 2012.

A. Dragojević, R. Guerraoui, and M. Kapalka. Stretching transactional memory. In *ACM SIGPLAN Notices*, volume 44, pages 155–165. ACM, 2009a.

A. Dragojević, Y. Ni, and A. Adl-Tabatabai. Optimizing transactions for captured memory. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*, pages 214–222. ACM Press, 2009b.

I. Dragos, A. Cunei, and J. Vitek. Continuations in the Java virtual machine. In *Second ECOOP Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'2007)*. Technische Universität Berlin, 2007.

J. Fernandes. Replicação preditiva para memória transaccional por software distribuída. Master's thesis, Instituto Superior Técnico, 2011.

S. Fernandes and J. Cachopo. Lock-free and scalable multi-version software transactional memory. In *Proceedings of the 16th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '11)*, pages 179–188. ACM Press, 2011.

R. Guerraoui and M. Kapałka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '08)*, pages 175–184. ACM Press, 2008.

T. Harris. Exceptions and side-effects in atomic blocks. *Science of Computer Programming*, 58 (3):325–343, 2005.

T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *Proceedings of the 10th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '05)*, pages 48–60. ACM Press, 2005.

T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2nd Edition*. Morgan & Claypool Publishers, 2nd edition, 2010. ISBN 978-1608452354.

C. Haynes, D. Friedman, and M. Wand. Obtaining coroutines with continuations. *Computer Languages*, 11(3-4):143–153, 1986.

M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*, pages 289–300. ACM, 1993.

M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

M. Herlihy, V. Luchangco, M. Moir, and W. Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the 22nd Annual Symposium on Principles of Distributed Computing (PODC '03)*, pages 92–101. ACM Press, 2003.

M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *ACM SIGPLAN Notices*, 41(10):253–262, 2006.

S. Hu, R. Bhargava, and L. John. The role of return value prediction in exploiting speculative method-level parallelism. *Journal of Instruction-Level Parallelism*, 5(1), 2003.

Intel Corporation. Intel® transactional memory compiler and runtime application binary interface, 2009.

Intel Corporation. Automatic parallelization with Intel® compilers, 2010. URL http://software.intel.com/file/39655.

C. Jacobi, T. Slegel, and D. Greiner. Transactional memory architecture and implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 45)*, pages 25–36. IEEE Computer Society, 2012.

G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG-3)*, 2010.

M. Kulkarni, M. Burtscher, C. Caşcaval, and K. Pingali. Lonestar: A suite of parallel irregular programs. In *Proceedings of the IEEE 2009 International Symposium on Performance Analysis of Systems and Software (ISPASS '09)*, pages 65–76, 2009a.

M. Kulkarni, M. Burtscher, R. Inkulu, K. Pingali, and C. Caşcaval. How much parallelism is there in irregular applications? *ACM SIGPLAN Notices*, 44(4):3–14, 2009b.

M.. Lam and R. Wilson. Limits of control flow on parallelism. *ACM SIGARCH Computer Architecture News*, 20(2):46–57, 1992.

D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande (JAVA'00)*, pages 36–43. ACM, 2000.

T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java Virtual Machine Specification, Java SE 7 Edition*. Addison-Wesley Professional, 2013. ISBN 978-0133260441.

W. Liu, J. Tuck, L. Ceze, W. Ahn, K. Strauss, J. Renau, and J. Torrellas. POSH: a TLS compiler that exploits program structure. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*, pages 158–167. ACM Press, 2006.

M. Mann. Continuations Library, 2008. URL http://www.matthiasmann.de/content/view/24/26/.

V. Marathe, W. Scherer III, and M. Scott. Design tradeoffs in modern software transactional memory systems. In *Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems (LCR '04)*, pages 1–7. ACM Press, 2004.

A. McDonald, J. Chung, H. Chafi, C. Minh, B Carlstrom, L. Hammond, C. Kozyrakis, and K. Olukotun. Characterization of TCC on chip-multiprocessors. In *Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques (PACT-2005)*, pages 63–74. IEEE Computer Society, 2005.

M. Mehrara, J. Hao, P. Hsu, and S. Mahlke. Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. *ACM SIGPLAN Notices*, 44(6): 166–176, 2009.

M. Mehrara, P. Hsu, M. Samadi, and S. Mahlke. Dynamic parallelization of JavaScript applications using an ultra-lightweight speculation mechanism. In *Proceedings of the 2011 IEEE 17th International Symposium on High Performance Computer Architecture (HPCA '11)*, pages 87–98. IEEE Computer Society, 2011.

J. Moss and A. Hosking. Nested transactional memory: Model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

Y. Ni, A. Welc, A. Adl-Tabatabai, M. Bach, S. Berkowits, J. Cownie, R. Geva, S. Kozhukow, R. Narayanaswamy, J. Olivier, S. Preis, B. Saha, A. Tal, and X. Tian. Design and implementation of transactional constructs for C/C++. *ACM SIGPLAN Notices*, 43(10):195–212, 2008.

C. Oancea, A. Mycroft, and T. Harris. A lightweight in-place implementation for software thread-level speculation. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*, pages 223–232. ACM Press, 2009.

M. Ohmacht and IBM Blue Gene/Q Team. Hardware support for transactional memory and thread-level speculation in the IBM Blue Gene/Q system. In *2011 Workshop on Wild and Sane Ideas in Speculation and Transactions*, 2011.

J. Oplinger, D. Heine, and M. Lam. In search of speculative thread-level parallelism. In *Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques (PACT-1999)*, pages 303–313. IEEE Computer Society, 1999.

J. Ortega-Ruiz, T. Curdt, and J. Ametller-Esquerra. Continuation-based mobile agent migration, 2010. URL http://jao.io/docs/spasm.pdf.

S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues. Specula: Speculative replication of software transactional memory. In *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS '12)*, pages 91–100. IEEE Computer Society, 2012.

C. Pickett. *Software Method Level Speculation for Java*. PhD thesis, School of Computer Science, McGill University, 2012.

C. Pickett and C. Verbrugge. Return value prediction in a Java virtual machine. In *Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop (VPW2)*, pages 40–47, 2004.

L. Pina and J. Cachopo. Profiling and tuning the performance of an STM-based concurrent program. In *Proceedings of the compilation of the co-located workshops on DSM'11, TMC'11, AGERE!'11, AOOPES'11, NEAT'11, & VMIL'11 (SPLASH 2011)*, pages 105–110. ACM Press, 2011.

R. Rajwar and J. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 34)*, pages 294–305. IEEE Computer Society, 2001.

R. Rajwar, M. Herlihy, and K. Lai. Virtualizing transactional memory. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA '05)*, pages 494–505. IEEE Computer Society, 2005.

A. Raman, H. Kim, T. Mason, T. Jablin, and D. August. Speculative parallelization using software multi-threaded transactions. *ACM SIGPLAN Notices*, 45(3):65–76, 2010.

J. Reinders. Transactional synchronization in Haswell. Intel® Developer Zone, 2012.

T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. *Distributed Computing*, pages 284–298, 2006.

RIFE Team. RIFE : Web continuations, 2006. URL http://rifers.org/wiki/display/RIFE/Web%20continuations.html.

A. Rountev, K. Valkenburgh, D. Yan, and P. Sadayappan. Understanding parallelism-inhibiting dependences in sequential Java programs. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance (ICSM 2010)*, pages 1–9. IEEE Computer Society, 2010.

M. Saad, M. Mohamedin, and B. Ravindran. HydraVM: Extracting parallelism from legacy sequential code using STM. In *Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism (HotPar'12)*, pages 1–7. USENIX Association, 2012.

B. Saha, A. Adl-Tabatabai, R. Hudson, C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '06)*, pages 187–197. ACM Press, 2006.

G. Schätti. HotSpec - A speculative JVM. Master's thesis, Institute of Computer Systems, Swiss Federal Institute of Technology (ETH), 2008.

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

M. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. Scott, C. Ding, and P. Wu. Fastpath speculative parallelization. In *Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing (LCPC 2009)*, pages 338–352. Springer-Verlag, 2010.

L. Stadler, C. Wimmer, T. Würthinger, H. Mössenböck, and J. Rose. Lazy continuations for Java virtual machines. In *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ 2009)*, pages 143–152. ACM, 2009.

J. Steffan, C. Colohan, A. Zhai, and T. Mowry. A scalable approach to thread-level speculation. *ACM SIGARCH Computer Architecture News*, 28(2):1–12, 2000.

C. Tismer. Continuations and stackless python. In *Proceedings of the 8th International Python Conference*, 2000.

B. Titzer, D. Lee, and J. Palsberg. Avrora: Scalable sensor network simulation with precise timing. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN '05))*. IEEE Press, 2005.

N. Tuck and D. Tullsen. Initial observations of the simultaneous multithreading Pentium 4 processor. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques (PACT-2003)*, pages 26–34. IEEE Computer Society, 2003.

A. Wang, M. Gaudet, P. Wu, J. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of Blue Gene/Q hardware support for transactional memories. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT-2012)*, pages 127–136. ACM, 2012.

Z. Wang, H. Qian, J. Li, and H. Chen. Using restricted transactional memory to build a scalable in-memory database. In *Proceedings of the Ninth European Conference on Computer Systems (EuroSys '14)*, pages 1–15. ACM, 2014.

F. Warg and P. Stenström. Limits on speculative module-level parallelism in imperative and object-oriented programs on CMP platforms. In *Proceedings of the 2001 International Conference on Parallel Architectures and Compilation Techniques (PACT-2001)*, pages 221–230. IEEE Computer Society, 2001.

A. Welc, S. Jagannathan, and A. Hosking. Safe futures for Java. *ACM SIGPLAN Notices*, 40(10): 439–453, 2005.

J. Whaley and C. Kozyrakis. Heuristics for profile-driven method-level speculative parallelization. In *Proceedings of the 2005 International Conference on Parallel Processing (ICPP '05)*, pages 147–156. IEEE Computer Society, 2005.

S. Wilkinson and I. Watson. Speculative multithreading: An object-driven approach. In *2008 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA 2008)*, pages 81–88, 2008.

R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy. SUIF: An infrastructure for research on parallelizing and optimizing compilers. *ACM SIGPLAN Notices*, 29(12):31–37, 1994.

H. Yamauchi. Continuations in servers. In *JVM Language Summit 2010*, 2010.

R. Yoo and H. Lee. Helper transactions: Enabling thread-level speculation via a transactional memory system. In *2008 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA 2008)*, pages 63–71, 2008.

L. Zhang and C. Krintz. As-if-serial exception handling semantics for Java futures. *Science of Computer Programming*, 74(5-6):314–332, 2009.

C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *1st ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, 2006.