INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

JaSPEx: Speculative Parallelization on the Java Platform

Ivo Filipe Silva Daniel Anjo

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

Júri

Presidente:	Doutor Nuno João Neves Mamede
Orientador:	Doutor João Manuel Pinheiro Cachopo
Co-Orientador:	Doutor António Paulo Teles de Menezes Correia Leitão
Vogal:	Doutor João Manuel Santos Lourenço

Outubro 2009

Resumo

Processadores multicore, capazes de executar concorrentemente múltiplas threads de processamento, são cada vez mais comuns em servidores, computadores de secretária, computadores portáteis, e máquinas ainda mais pequenas. Infelizmente, durante grande parte do tempo estas máquinas são subutilizadas, pois muito do software actual não está preparado para aproveitar as suas capacidades de multiprocessamento. Além disso, nestas novas máquinas a adição de mais *cores* de processamento não se traduz numa maior performance na execução sequencial de código, levando a que aplicações sequenciais existentes não se executem mais rapidamente em multicores.

Com este trabalho pretendo abordar este problema utilizando especulação ao nível da thread, combinada com uma memória transaccional em software, para automaticamente paralelizar programas sequenciais.

Será apresentado o Java Speculative Parallel Executor (JaSPEx), um sistema que combina a modificação de aplicações para que se executem transaccionalmente sobre o controlo de uma memória transaccional em software (o processo de *transactificação*), com a execução paralela especulativa de uma aplicação, numa Máquina Virtual Java não-modificada.

Tanto a transactificação como a execução especulativa são feitas automaticamente, sem serem necessárias modificações à aplicação original ou intervenção do seu programador, para todas as aplicações existentes que se executem na Máquina Virtual Java.

As dificuldades inerentes ao processo de transactificação de uma aplicação são também descritas, assim como a relação entre a execução especulativa de um programa e a forma como é utilizada a memória transaccional em software.

Apresento também resultados promissores para a minha aproximação, obtidos com uma implementação preliminar do sistema JaSPEx.

Abstract

Multicore processors, capable of running multiple hardware threads concurrently, are becoming common on servers, desktops, laptops, and even smaller systems. Unfortunately, most of the time these new machines are underutilized, as most current software is not written to take advantage of their multiprocessing capabilities. Also, with these new machines, more cores do not translate into more sequential performance, and existing sequential applications will not speed up by moving to a multicore.

This work proposes to tackle this problem with the use of thread-level speculation based on a software transactional memory, to parallelize automatically sequential programs.

The Java Speculative Parallel Executor (JaSPEx) system is described, which combines the modification of an application so that it can execute transactionally under the control of a software transactional memory (the *transactification* process), with the speculative parallel execution of the application, on top of an unmodified Java Virtual Machine.

Both the transactification and the speculative execution are done automatically, without needing any changes to the original application or input from its programmer, for all existing sequential applications that execute on the Java Virtual Machine.

I also address the difficulties inherent to the process of transactifying an existing program, and what is the relationship between the speculative execution of a program and the software transactional memory that it is using.

I present early benchmark results with a proof-of-concept implementation of the JaSPEx system, that show promising results for the approach I propose.

Palavras-chave

Execução Especulativa
Especulação ao Nível da Thread
Memória Transaccional em Software
Aplicações Legadas
Arquitecturas Multicore
Programação Paralela

Keywords

Speculative Execution
Thread-level Speculation
Software Transactional Memory
Legacy Applications
Multicore Architectures
Parallel Programming

Acknowledgements

The present work was a result of almost a year's worth of ideas, possibilities, research and development. It evolved from an initial idea to a life-size prototype, and it picked up quite interesting solutions for some problems it and I faced along the way.

I would like to start by thanking my colleagues Hugo Rito and Luís Pina, for constantly and patiently letting me bounce ideas off them, and for their out-of-the-box thinking when suggesting solutions for the problems I described, and the white blackboard that just sat there without complaints through most of my senseless scribblings.

The support and dedication of Professor João Cachopo, who reviewed and discussed my work countless times, were essential to this work. Without his knowledge, insights and especially without his constant encouragement I would not have come so far.

I also want to tip my (non-red) hat to the people of the Software Engineering Group (ESW) at INESC-ID, for all the interesting presentations and for sitting around for some of my own.

Finally, I would like to extend a big THANK YOU and I LOVE YOU to my parents and sister, for putting up with me during all this time, especially during stressful heavy-work times.

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

September 2009
Ivo Anjo

Contents

List of Figures

List of Tables

List of Abbreviations

API	Application Programming Interface
CMP	Chip Multiprocessor
HTM	Hardware Transactional Memory
IDE	Integrated Development Environment
JVM	Java Virtual Machine
SMP	Symmetric Multiprocessing
STM	Software Transactional Memory
TLS	Thread-Level Speculation
TM	Transactional Memory
VM	Virtual Machine

Chapter 1

Introduction

The transition to multicore architectures is ongoing. Chip designers are no longer racing to design the fastest uniprocessor, instead turning to parallel architectures, with newer designs incorporating multiple cores, capable of running many threads simultaneously.

The full power of these multicore chips is unlocked only when all cores are busy executing code. Yet, most desktop applications fail to take advantage of these processors, having little, if any, parallelism.

The problem for legacy applications is that for years new, faster, uniprocessors brought "*free*" speedups for all applications: just upgrade to the latest processor and your applications go faster, with no other modifications needed; this does not apply with multicore processors.

Concurrent programming is, unfortunately, harder than sequential programming. Moreover, even if newly developed applications are written with multicore architectures in mind, most of the already developed code is still sequential and it is not feasible to rewrite it within a reasonable time frame.

So, the problem is that we cannot expect to get better performance on most legacy applications just by upgrading to a multicore processor, and rewriting or adapting legacy applications poses many problems. Thus, an enticing alternative is to parallelize applications automatically. In fact, there is already significant research towards this goal.

For instance, parallelizing compilers [1, 2] try to automatically extract concurrency from a sequential program description, while still maintaining program correctness. The problem is that they still fail to parallelize many applications, because of data and interprocedural dependencies that are very hard to analyze at compile-time in a fully static way.

This work explores a different approach — speculative parallelization. Rather than parallelizing only code that is provably able to run in parallel, speculative parallelization uses a more aggressive approach that parallelizes code that may have dependencies, and relies on the ability to roll back a speculative execution when it detects that the parallelization could not have been done.

Unlike other approaches to automatic parallelization that rely on hardware-supported speculative execution (e.g., [3, 4, 5]), the distinguishing feature of my proposal is the use of a software transactional memory (STM) [6, 7] to back up the speculative execution. To the best of my knowledge, I am the first to propose the use of an STM for speculative parallelization.

I argue that using an STM for speculative execution has several advantages over hardware-supported approaches. First, because STM-based executions are unbounded, I may extend the scope of possible speculative parallelizations, thereby increasing the potential for extracting parallelism from sequential applications. Second, I may apply these techniques to applications that run on hardware that does not support speculative execution (including all of the current mainstream hardware). Finally, I may benefit from much of the intense research being done in the area of transactional memory.

Yet, switching from hardware-supported speculation to an STM-based approach, introduces other challenges, such as being able to transactify a program to run it speculatively. This work describes JaSPEx — the Java Speculative Parallel Executor — a system that automatically parallelizes programs for the Java Virtual Machine (JVM) using an STM-based speculative approach. JaSPEx rewrites the bytecode as it is loaded by the JVM runtime, modifying it to run speculatively on top of an STM.

1.1 Thesis Statement

This dissertation's thesis is that it is possible to build a speculative parallelization system for existing sequential applications that run on the Java Virtual Machine platform, without needing any kind of special support from the Virtual Machine, and transparently to the programmer, by rewriting application bytecode and relying on the help of a software transactional memory.

The usage of a software transactional memory allows my approach to be applied to common mainstream hardware. By also targeting an unmodified Virtual Machine platform, this approach becomes as platform-independent as normal Java applications are: it can be applied on any JVM on any platform that complies with Sun's JVM specification.

By being automatic and transparent to the programmer, my approach can be applied to all applications, without requiring any changes to the application being executed.

To validate my thesis, I built the Java Speculative Parallel Executor system, a speculative execution system that embodies the proposed approaches. In this dissertation I describe in depth the design and implementation of the JaSPEx system, as well as the ideas I explored and limitations that had to be overcome.

1.2 Notation

The modifications done by JaSPEx to applications are done via Java Virtual Machine bytecode rewriting. But, because looking into these transformations at the bytecode level makes them harder to understand, in this dissertation I shall only use bytecode in a small number of examples; for most examples, these transformations will be presented as semantically equivalent changes at the Java programming language level.

A Java method's signature is composed of the name of the method, the types of its formal parameters (if any), and a return type (which can be `void`). An example method signature for a method returning the integer sum of two integer values is `int sum(int val1, int val2)`. In this document, whenever parts of a method signature are not relevant, they may be skipped, and I may refer to our example method as

`sum`, `sum()`, `sum(int, int)`, `int sum()` or `int sum(int, int)`, if it is clear from the context which method is being described.

Whenever possible, *generics* [8] are used in the examples. Note, however, that Java's implementation of generics is mostly invisible and irrelevant to the Java Virtual Machine environment, as they are implemented using *type erasure*.

1.3 Outline

The remainder of this dissertation is organized as follows:

- *Related Work.* Chapter 2 presents a review of relevant work related to this dissertation.
- *Problem Statement and Solution Overview.* Chapter 3 further describes the problem that I propose to solve, and the requirements for the solution. An overview of the solution is then presented, including the general design idea for speculative execution of method calls.
- *Transactification of JVM Applications.* Chapter 4 introduces the transformation of existing code so that it may run with transactional semantics, a process that I call transactification. Strategies for implementation of transactification of both fields and arrays are presented. Finally, the issue of handling execution of non-transactional operations is also analyzed.
- *Speculative Parallelization.* Chapter 5 starts by introducing and discussing the design of the speculative execution model that is used by the JaSPEx framework, and some of the compromises that had to be made to achieve positive results. It then describes how speculation is done: what changes are made to programs so that they can run speculatively; when the framework generates speculative execution tasks; how tasks are managed internally by the framework; and when can transactions commit their modifications. Also examined are the issues of reusing worker threads, and how JaSPEx interacts with the JVSTM. The chapter ends with a discussion of the overall solution, and how it is shaped by the limitations imposed by the Java Virtual Machine.
- *Experimental Results.* Chapter 6 presents measurements of the overheads introduced by the transactification process, and experimental results of applying speculative parallelization to a number of applications.
- *Conclusion.* Chapter 7 summarizes the work described in this dissertation, the results achieved, and what are its main contributions. It also describes future work possibilities to be explored, to expand the ideas and approaches introduced by this dissertation and the JaSPEx system.

Chapter 2

Related Work

The following sections introduce important research work related to this dissertation.

Parallelizing Compilers (Section 2.1) were the first approach to automatic parallelization. These compilers try to automatically extract concurrency from a sequential program description, but their approach based on static analysis fails to work with many applications, because of data and interprocedural dependencies.

A possible solution to the limitations presented by parallelizing compilers is the usage of speculative parallelization, where the parallelization system does not have to prove that a parallelization is always valid, because application code is run under a memory transaction that can be aborted and undone when a memory access conflict is detected.

Support for memory transactions can be achieved through the use of Hardware Transactional Memory (Section 2.2), Software Transactional Memory (Section 2.3) or through Hybrid Approaches (Section 2.4) that combine both. Each approach has its advantages and disadvantages: HTMs offer very small overheads for transaction start and commit, but are limited by the features provided by the hardware, that often limit transaction size and duration; STM designs offer big or even unbounded transaction size and duration, and do not need any kind of special hardware support, but can impose big overheads; and hybrid designs try to balance both approaches by allowing transactions to execute in HTM mode if possible, falling back to STM mode otherwise.

Thread-Level Speculation (Section 2.5) systems combine speculative parallelization with hardware support for transactions to try to parallelize applications at a very fine-grained level. A common approach is to parallelize loops by executing each loop iteration inside its own transaction. Other approaches include grouping application code into tasks that are run in parallel.

Parallel Programming Languages (Section 2.6) present an alternative approach to application development, where application design is entirely thought out to support and encourage concurrent execution. They also tackle issues that a speculative parallelization system has to resolve, like task scheduling and atomic execution.

The Java Concurrent Programming (Section 2.7) and Transactional Java Execution (Section 2.8) sections describe other approaches taken for concurrent programming inside the Java platform, some of which share traits with how the system described in this dissertation is implemented, providing insights

into the possibilities offered by the platform.

Finally, in Section 2.9, the Java Versioned Software Transactional Memory (JVSTM) — the STM used by the JaSPEx system — is introduced and discussed.

2.1 Parallelizing Compilers

The Polaris compiler [1] is a prototype parallelizing compiler for Fortran that combines many techniques to try and overcome the limitations of other parallelizing compilers. 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 are used to try and break data dependencies between different iterations of loops, so that they may execute in parallel. Symbolic dependence analysis is used to determine what statements or loops can be safely executed in parallel. Scalar and array privatization identifies scalars and arrays that are used as temporary work space by an iteration of a loop, and allocates local copies of them. Run-time analysis is also included, where some loops that cannot be analyzed statically are augmented with a test at run-time to determine if there are cross-iteration dependencies; during run-time execution, parallel versions of these loops write into a temporary location, and if they pass the dependencies test, their result is stored in permanent locations, otherwise a sequential version of the loop is reexecuted.

SUIF [2] is an infrastructure for experimental research on parallelizing and optimizing compiler technology, aiming to provide researchers with a modular and flexible platform on which to test new ideas. It supports ANSI C and Fortran (via translation to C) front-ends, a loop-level parallelism and locality optimizer, and an optimizing MIPS back-end. Parallelization is done in multiple passes: first, scalar optimizations are applied to help expose parallelism, such as constant propagation, forward propagation, induction variable detection, constant folding, and scalar privatization analysis; array dependence analysis is then applied, and its results are used to transform loops. The current loop analyzer recognizes sum, product, minimum and maximum reductions. Ongoing research projects based on this infrastructure include global data and computation decomposition, array privatization, interprocedural parallelization and efficient pointer analysis.

2.2 Hardware Transactional Memory

Transactional Memory [9] was initially proposed by Herlihy and Moss as a multiprocessor architecture capable of making lock-free synchronization as efficient and easy to use as conventional techniques based on mutual exclusion, while avoiding pitfalls like priority inversion, convoying, and deadlocks. The implementation was based on extensions to multiprocessor cache-coherence protocols, addition of some new instructions to the processor, and a small transactional cache where transactional changes were kept prior to committing. It was a direct generalization of Load-Linked/Store-Conditional, providing the same semantics for multiple memory words, instead of being restricted to just one. But, it was not *dynamic*: memory usage and the transactions had to be statically defined in advance.

Software transactional memories have since been proposed that overcome many of these problems, at the cost of bigger overheads.

2.3 Software Transactional Memory

Software transactional memory [6] was introduced as an alternative to hardware transactional memory that could be implemented on top of Load-Linked/Store-Conditional of a single memory word, as provided by most current hardware architectures. The authors recognized that although hardware transactional memory was a very promising tool for implementation of non-blocking concurrent programs, applications using it were not portable, and most architectures being introduced and planned at that time (and today still) did not incorporate HTM into their designs. As it was the case with Herlihy's original HTM, this STM was limited to static transactions, where the data set is known in advance, providing only a k-word compare-and-swap operation.

The Dynamic Software Transactional Memory (DSTM) [7] was the first unbounded STM, allowing it to be used in the implementation of dynamically-sized data structures such as lists and trees. It allowed transactions to detect if they would cause some other transaction to abort before doing so, therefore allowing the decision of whether to proceed or to give the other transaction a chance to complete its work first. Such decisions were made by pluggable contention managers.

The McRT-STM [10] is an STM implementation that is part of McRT, an experimental multicore runtime. This STM supports advanced features such as nested transactions with partial aborts, object based conflict detection for C/C++ applications, and contention managers. The authors also present an analysis of STM design tradeoffs such as optimistic versus pessimistic concurrency, write buffering versus undo logging, and cache line based versus object based conflict detection. Unlike many designs, the McRT-STM does not provide non-blocking guarantees, because the authors argue that the implementation is both simplified and more efficient this way. Memory changes are done in-place on writes; the location's original value is saved on an undo log, which is used in case the transaction aborts.

2.4 Hybrid Approaches

Hybrid Transactional Memory (HyTM) [11] 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 using HTM or using the STM library. An important part of this work is reconciling concurrent hardware and software transactions, especially detection of conflicts between them. Additionally, this hybrid approach allows 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 an unbounded (dynamic) hardware solution from the start. The authors also recognized the importance of the contention managers proposed by Herlihy et al. [7].

Rajwar, Herlihy and Lai [12] present the Virtual Transactional Memory (VTM) 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 the "hardware-only" mode (HTM) to the unbounded transaction mode.

2.5 Thread-Level Speculation

POSH [3] presents a Thread-Level Speculation (TLS) infrastructure on top of the GNU Compiler Collection (GCC) that 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 dependencies — the hardware ultimately guarantees correct execution; spawn hoisting tries to place the transaction spawn instruction as early as possible, with some restrictions; and in the task refinement phase the final set of tasks is selected. The profiler runs applications with a training input set, and provides the compiler with a list of tasks that are beneficial for performance, allowing the compiler to eliminate non-beneficial tasks. 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 are already in cache, and so a squashed task also works as a prefetching mechanism.

In [4] the authors present a reverse compilation framework that translates binary code to static single assignment (SSA) form, from there performing optimizations and adding support for speculative execution. Profiling is also used to identify important program areas, and to predict likely values, reducing the number of speculative mis-predictions and their penalties. The reverse compilation approach allows speculative parallelization of existing legacy applications, because no access to program source code is needed, only to program binaries.

Bridges et al. [13] describe a framework that combines whole-program analysis and speculative execution with extensions to the sequential programming model that further enable parallelization. The suggested extensions allow the programmer to specify that multiple legal outcomes of the program execution are possible:

- The `Y-branch` extension is similar to a common `if-then-else`, but it allows the true path to be taken regardless of the condition of the branch; the compiler is free to generate code to pursue this path when it is profitable to do so (Figure 2.1).
- The `Commutative` extension informs the compiler that calls to a function (or group of functions) can occur in any order; this is particularly useful for functions that have internal state — and as such generate dependencies — but on which the order of the calls are not relevant to the application as a whole, like the `random` function shown in Figure 2.2.

Jrpm [5], the Java runtime parallelizing machine is a Java virtual machine that does TLS on a multiprocessor with hardware support. Buffer requirements and inter-thread dependencies are analyzed at runtime to identify loops to parallelize. Once sufficient data is collected, the selected loops are dynamically recompiled. As Jrpm works at the Java bytecode level, no changes need to be made to the source binaries or code.

Oplinger et al. [14] 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

```
while (b = readByte()) {
    profitable = compress(b, dictionary);

    @YBranch(probability=.0001)
    if (!profitable)
        dictionary.restart();
}
```

Figure 2.1: Example use of a Y-Branch on a simplified dictionary-based compression algorithm. The compiler is free to restart the dictionary, even if `profitable` is `true`, giving the compiler the ability to break dependencies caused by the restart operation. The probability argument informs the compiler that the dictionary should not be restarted until at least 10000 bytes are compressed.

```
private long seed;

@Commutative
protected synchronized int next(int bits) {
    seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
    return (int) (seed >>> (48 - bits));
}
```

Figure 2.2: Implementation of the `java.util.Random.next()` method, which generates a pseudorandom number based on a seed, annotated with the Commutative extension.

the use of speculative procedure execution, coupled with procedure return value prediction. The results that they present were obtained by simulating the execution of applications on variations of an optimal speculative thread-level parallelism (STP) machine; it is argued that this optimal machine can be used to derive an upper bound on the performance achievable on any real machine of a similar design. Speculative procedure execution is done by executing a called procedure in parallel with the code following the return of the procedure. This approach is then applied recursively. As it is common that the return value of a procedure is immediately used, this is where the proposed procedure return value prediction is applied. This prediction works by keeping a cache of past values returned by a procedure, which are then used to run speculatively the code following the procedure return immediately; in case of mis-speculation, the speculative thread is aborted, and the code rerun with the real value.

2.6 Parallel Programming Languages

The Fortress [15, 16] programming language is designed to make parallel programming simple and painless. In Fortress, a number of constructs are *implicitly parallel*. These include tuple expressions, `also do` blocks, function calls, `for` loops, comprehensions, sums, and generated expressions. Implicitly parallel constructs are automatically run in threads that are managed entirely by the compiler, runtime, and libraries of the implementation. Threads can also be explicitly started using the `spawn` construct, but its use is discouraged. Iteration is done by the use of *generators* that drive *reducers*. Atomic expressions allow operations to be executed transactionally, where all other threads either observe that the computation of the atomic block has completed, or that it has not yet begun; handling of conflicts is left to the implementation. Additionally, functions that perform primitive input/output operations — that have

```
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
    }
}
```

Figure 2.3: General pseudo-code form of a fork/join algorithm.

externally visible effects — must be explicitly declared with the `io` modifier, which cannot be used in combination with the `atomic` modifier.

X10 [17] is an object-oriented Java-inspired language aimed at *Non-Uniform Cluster Computing* (NUCC) systems, where multiple nodes containing multicore SMP chips with non-uniform memory hierarchies are interconnected in horizontally scalable cluster configurations. It supports parallel programming across multiple cores in a chip, and across multiple systems in a cluster. Each system represents a *place*, and multiple *activities* (similar to threads/tasks) can be run in each place. The `async` keyword is used to schedule execution of an activity on a local or remote place. Parallel iteration can be done across multiple activities in a single place by use of `foreach`, and across multiple places using `ateach`. Atomic blocks are also supported, but isolation only occurs under normal execution; if an atomic block terminates abruptly (e.g. with an unhandled exception), transaction semantics are explicitly violated, and it becomes the responsibility of the programmer to clean up any side-effects performed inside it. Support for futures, conditional atomic blocks and partitioned multi-dimensional arrays is also included.

2.7 Java Concurrent Programming

`FJTask` [18] is a framework due for inclusion on the upcoming 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 Figure 2.3. The framework uses a pool of worker threads, which are reused for multiple tasks; tasks themselves are lightweight objects. A special queueing 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 deque is empty, threads try to obtain tasks by work-stealing [19] from other threads. The author argues that standard thread frameworks are too heavy to support this style of programming, imposing unneeded overheads and having too generic scheduling algorithms.

JCilk [20] 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. 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

```
cilk int fib(int n) {
    if (n <= 1) return n;
    int x = spawn fib(n-1);
    int y = spawn fib(n-2);
    sync;
    return x+y;
}
```

Figure 2.4: Recursive JCilk implementation of the Fibonacci function.

reasoning about them. An important detail is that if the JCilk keywords are elided from a program, a syntactically correct serial Java application results. Figure 2.4 shows a JCilk implementation of the Fibonacci function.

Welc et al. [21] introduce safe futures for Java, which are futures that work as semantically transparent annotations on methods, designating opportunities for concurrency. Execution of a method can be replaced with execution of a future, the safe future model guaranteeing that sequential execution semantics are respected, and observed behaviour of serial and concurrent tasks are the same. The implementation is based on the Jikes RVM [22] virtual machine, and includes features very similar to those provided by STMs.

2.8 Transactional Java Execution

Carlstrom et al. [23] 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. The authors argue that strong transactional atomicity semantics are 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 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 the pessimistic lock-based implementation of the application.

In [24], the authors present the Atomos programming language, the first 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 RVM [22] Java virtual machine, and on the Transactional Coherence and Consistency (TCC) hardware transactional memory model [25]. 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. [26]. Also included are open-nested transactions [27], 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,[1] while minimizing the risk of data dependency violations, by limiting

[1] In Java, declaring a variable `volatile` means that it will never be cached by the current thread; reads and

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 dependency violations without rolling back. Benchmarks were done on a PowerPC CMP system simulator with support for the TCC HTM, and show a clear advantage of Atomos over Java for the tested applications.

2.9 Java Versioned Software Transactional Memory

The Java Versioned Software Transactional Memory (JVSTM) [28, 29] is a pure Java software transactional memory library that is used to provide transactional execution semantics for code that is being speculatively executed by the JaSPEx framework.

The JVSTM introduces the concept of versioned boxes [30], which are containers that keep the history of the values of an object, each of these corresponding to a change made to the box by a committed transaction. This arrangement allows read-only transactions to never conflict with any other concurrent transaction, favouring applications that have a high read/write transaction ratio.

The authors recognize that read-only and read-write transactions present different implementation overheads, and the importance of distinguishing between them. Also proposed are speculative read-only transactions in which the JVSTM speculatively assumes that a transaction is read-only when it starts, but later the transaction is aborted and restarted as a read-write transaction if it tries to change a versioned box; garbage collection of old history values that are not needed anymore, because all transactions that could reach them have either committed or been permanently aborted; and consistency predicates, which allow domain-based validation of transactions at commit-time.

Detection of conflicts caused by read-write transactions is done only at transaction commit time.

Nested transactions are supported, and follow the *linear nesting* model [27]. In this arrangement, concurrency within a transaction is disallowed, and each top-level transaction can only have at most one nested child transaction at any given time, although each nested child transaction can also have at any given time at most one other nested child transaction. Child transactions are executed by the same thread of their parents, which means that if a child transaction is executing, then its parents are not. Additionally, sibling transactions cannot be executed simultaneously.

The JVSTM was also designed to work well with medium (using hundreds to thousands of java objects) and long-running transactions (using thousands to millions of objects) [28].

2.9.1 Transactional Model

A transaction is a sequence of steps that is executed by a single thread. 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 effects done by the transaction become visible to the rest of the system, all at the same time, or with an **abort**, in which case effects done during the transaction are discarded, never becoming visible to the rest of the system.

writes to this variable will always be forced to go to main memory.

Both of these behaviours are very important for speculative execution: the ability to execute a number of instructions without their results becoming immediately visible allows reordering of code execution without affecting the original application semantics; and the ability to discard results allows invalid executions, where the parallelization done by the system did not follow the original application semantics, to be discarded without impacting the rest of the system.

2.9.2 Versioning and Conflict Detection

Every transaction in the JVSTM has a version number that is assigned when the transaction is created, which represents the version number of the latest read-write transaction that successfully committed.

When a value of a versioned box is read inside a transaction, the box returns not always its latest value, but a value that has an equal or smaller version than the current transaction version. When a value is written to a VBox inside a transaction, it is put on the transaction's write-map, and is only visible to other threads after the transaction is successfully committed.

Read-only transactions are always allowed to commit, because they do not change the application state.

A read-write transaction is only allowed to commit if all of the versioned boxes that it read have the same or a smaller version than the transaction's, meaning that no other read-write transaction has committed changes to those boxes while this transaction was executing. Only the successful commit of a read-write transaction can cause the version to be incremented.

2.9.3 API

The JVSTM has a very simple API, and most applications need only to access two classes: `jvstm.VBox<E>` and `jvstm.Transaction`.

The `VBox` class implements the versioned box concept. Figure 2.5 shows the relevant API for this class. The `get` method returns the value of the VBox for the current transaction, and `put` modifies the value of the VBox for the current transaction.

```
package jvstm;

public class VBox<E> {
    public VBox() { ... }
    public VBox(E initial) { ... }

    public E get() { ... }
    public void put(E newE) { ... }
}
```

Figure 2.5: Relevant `jvstm.VBox<E>` class API.

There are also specific `VBox` versions for each primitive Java type, which can be used to avoid the need to box and unbox these types: `VboxBoolean`, `VboxChar`, `VboxByte`, `VboxShort`, `VboxInt`, `VboxLong`, `VboxFloat` and `VboxDouble`.

The `Transaction` class controls the start, commit and abort of transactions. Figure 2.6 shows the relevant API for this class. The `begin` method starts a new transaction, and sets it as the current transaction for this thread; if a transaction was already active, a nested transaction is created. The `current` method returns the current transaction, if any. The `commit` method tries to commit the current transaction; if this operation fails, a `CommitTransaction` is thrown. Finally, the `abort` method aborts the current transaction.

```
package jvstm;

public abstract class Transaction {
    public static Transaction begin() { ... }
    public static Transaction current() { ... }

    public static void commit() { ... }
    public static void abort() { ... }
}
```

Figure 2.6: Relevant `jvstm.Transaction` class API.

Chapter 3

Problem Statement and Solution Overview

To make full use of a multicore processor, an application needs to split up its workload, and assign it to be run by multiple processing cores concurrently. This model differs from the sequential programming model, where an application only uses a single processing core at a time.

Most existing applications execute sequentially, so they cannot benefit from running on a multicore processor. Writing a new application that works concurrently is hard, and retrofitting concurrency on existing applications is even harder.

A possible solution to these problems is automatic parallelization. As I have introduced in the previous chapter, parallelizing compilers were the first approach to automatic parallelization, but their static analysis approach is unsuccessful for most applications. Thus, the idea of speculative parallelization was born: the speculative parallelization system does not have to be sure that a parallelization is valid, and can use heuristics to try and parallelize an application.

We may parallelize the execution of a Java method like the one shown in Figure 3.1 by executing the calls to `doA` and `doB` in parallel, as exemplified in Figure 3.2. The problem is that these methods might modify and access some shared state, and as such may not be able to run in parallel.

The key difference between earlier automatic parallelization approaches and speculative parallelization is that instead of statically analyzing `doA` and `doB` to figure out if their parallel execution is safe, a speculative parallelization system runs them anyway, and relies on having both the ability to detect when the execution violates sequential execution semantics and the ability to reverse the changes done by a speculative execution when such a violation occurs. Both of these abilities are provided by a transactional memory. Yet, current speculative parallelization systems rely on the presence of hardware transactional memory, which is not available on any of the current mainstream computer architectures.

Speculative parallelization may be applied at the source-code level, or at the application binary level. It may also require the programmer to change the application, or work independently without needing changes to the original application.

I argue that having a system that works both without access to the original application source code, and that does not need input from the programmer(s) is a desirable property. This way, parallelization

```
void method() {
    doA();
    doB();
}
```

Figure 3.1: Example method to be parallelized.

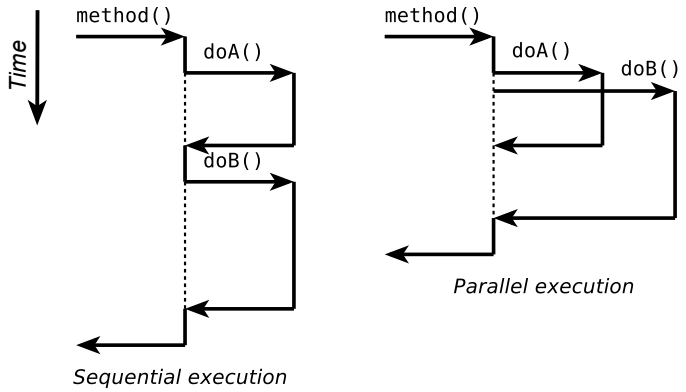

Figure 3.2: Sequential and parallel executions of `method`.

can be done independently from the creation of the original application.

Not all application code is parallelizable. Because a speculative parallelization system can only know at runtime if it is being successful, and success rates may change with differing inputs to the application, we are generally unable to know in advance if parallelization is going to be possible or even worth the added overheads. Despite this uncertainty, there is research [14] that points to the existence of untapped parallelism opportunities in most applications.

As the speedups achieved by speculative parallelization are application-specific and can even be input-specific, besides being dependent on the techniques used, the system that I propose should be, above all, a framework for research on speculative parallelization techniques. As such, this framework should be modular and should facilitate the experimentation of various implementations of subsystems such as execution task selection and task scheduling.

A fundamental requirement for such a system is that, even if a parallelization technique is unsuccessful, the platform should always be able to run an application successfully, preserving the original sequential application execution semantics. That is, the system should degrade gracefully, rather than not working at all for some applications.

By developing such a framework, I hope to provide a basis for incremental research on speculative parallelization, while still being able to run normal application code.

For this work, I have chosen to work on top of the Java Virtual Machine platform. The Java language [31] and virtual machine [32] have the goal of providing a portable, object oriented, garbage collected, high performance environment for applications to execute.

Java applications are typically compiled to a low-level standard bytecode representation that can

be run on any Java Virtual Machine (JVM). This standard bytecode representation makes it simpler to analyze and modify current applications; it also allows custom virtual machine implementations to seamlessly integrate support for hardware and software transactions, while still preserving the original sequential code semantics [33].

The Java memory model is also clearly defined [34], providing clear semantics for the legal behaviours of parallel applications such as STM libraries. This way both the original java applications and their speculative execution versions can be run on all architectures that feature a compliant virtual machine.

Unlike related work that modifies the Java Virtual Machine for speculative execution, I have chosen to work on top of unmodified JVMs. By targeting an unmodified Virtual Machine platform, my approach becomes as platform-independent as other applications that run on the JVM platform, and can be applied on any JVM on any platform that complies with Sun's JVM specification. This also allows my approach to benefit from current research in Java Virtual Machine implementations [22, 35].

As the JVM platform does not offer any transactional support (hardware or software based), I used the Java Versioned Software Transactional Memory (JVSTM) to support this work. Because the JVSTM is a normal Java library, applications have to be modified to work with it, a process I call *transactification* of an application.

Thus, to develop a speculative execution system that takes into account these problems, and tries to solve them, I propose the following requirements for my work:

- *Automatic and transparent to the programmer.* The system developed should work automatically and without the need of input from the programmer(s) of the application. It should not need access to the application's source code, also.

- *Applicable to current execution environments.* Parallelized applications should be able to run in current execution environments, and should not need support for processor extensions that are currently not available on mainstream computers.

- *Correct execution of applications.* Any sequential application that runs on the JVM platform should be able to be correctly executed by the system. Even if the execution of the application does not benefit from the speculative parallelization, original sequential execution semantics should always be preserved.

- *Modularity and support for incremental development.* By extension of the previous requirement, the system should allow incremental development and research into speculative parallelization techniques, while still allowing applications to run correctly. Transactification and speculative execution should be treated separately, and the components needed for speculative execution, such as task selection and scheduling should be easily replaceable to facilitate experimentation of various implementations.

- *Work on an unmodified Java Virtual Machine.* The system should work with any compliant JVM implementation, and should not need a special Virtual Machine implementation to work. This allows it to be portable and to benefit from new JVM implementations.

3.1 Proposed Solution Overview

JaSPEx automatically parallelizes applications that run on the Java Virtual Machine. Applications are first modified to execute transactionally, under the control of a Software Transactional Memory; and then further modified to add support for speculatively executing method calls in parallel with their callers. All modifications to applications are done by rewriting their Java bytecode.

The JaSPEx system is organized into two big components that address two distinct and complementary aspects of my proposed solution.

- The Transactification (Chapter 4) part of the JaSPEx system takes care of adapting an existing application to run under the control of the chosen Software Transactional Memory. It does transactification of class and instance fields (Section 4.1); transactification of arrays is also covered (Section 4.2), although it is currently disabled, for reasons that will be explained later. Finally, it also handles the identification of non-transactional operations (Section 4.3).
- The Speculative Parallelization (Chapter 5) part of the system further modifies the application for speculative execution (Section 5.2), inserting entry points to allow the system to control the methods executed and identifying tasks for speculative parallel execution. During application execution, it controls the start, commit, and abort of speculative executions, as well as their scheduling (Section 5.3).

I decided to separate these two aspects, because they may be used independently. If the Java Virtual Machine being used already supports transactional execution of applications, only the speculative parallelization part of JaSPEx would be used. Transactification itself may be useful without speculative parallelization, because nowadays applications that make use of a Software Transactional Memory have to be manually transactified by the programmer; with the separation between transactification and speculation, this job could be performed instead by the automatic transactification part of JaSPEx.

Modifications to the application are applied at class load-time, via Java bytecode rewriting, using a Java class loader that transforms and prepares classes as they are requested by the application. Bytecode rewriting is done using the ASM bytecode manipulation framework [36].

Chapter 4

Transactification of JVM Applications

Because the JVM runtime has no support for transactional execution of code, an application must first be modified to run transactionally, so that the automatic parallelization system is able to detect when a speculative execution violates sequential execution semantics, and is able to reverse the changes made by a speculative execution when such a violation occurs.

To solve this problem, I propose the use of a software transactional memory [6, 7] to allow (parts of) the program memory to act transactionally. Execution of different parts of the application is then mapped to different transactions each executing on their own thread, and when there is a conflict between two transactions we know that there may have been a violation of sequential execution semantics, and abort the one that comes later in the original program execution.

Coming back to the example in Figure 3.1, we can parallelize execution of `method` by running `doA` and `doB` in separate threads, each with a different transaction. If the STM system detects a conflict between the speculative execution of `doA` and `doB`, we abort `doB`, and schedule it for reexecution, because the original program order puts `doA` before `doB`; if no conflict is detected, the two methods are run in parallel, and this should result in a speedup over the sequential version.

The software transactional memory currently used for JaSPEx is the Java Versioned Software Transactional Memory (JVSTM) [30, 28], which was already described in Section 2.3. The JVSTM was chosen for its features and due to my familiarity with it, but this approach can also be used with other STMs.

Because the JVSTM is a normal Java library, applications have to explicitly relinquish control of accessible memory locations to it. On the JVM platform, an accessible memory location must either contain a primitive type,[1] a reference to an object, or a `null` reference. The only accessible memory positions in the JVM are fields (class or instance), array positions, and local variables. Field and array positions will have to be modified to hold `jvstm.VBox` instances, instead of simply the original values or references. This process, which I call *transactification*, has to be applied to all classes of a target application, so that it runs entirely under the control of the JVSTM. Modification of local variables is

[1] In the JVM, the following types are considered primitive, and are handled separately from objects: `boolean`, `byte`, `char`, `short`, `int`, `float`, `long`, and `double`. Of these, `long` and `double` have to be handled differently, as they occupy two "slots" on the stack, instead of one, as is used by everything else, including objects. See also `http://java.sun.com/docs/books/jvms/second_edition/html/Concepts.doc.html#19511`.

currently not needed, for reasons explained in Section 5.1.

The rest of this chapter is organized as follows. Section 4.1 describes transactification of fields. Transactification of arrays and the problems it poses are examined in Section 4.2. Finally, Section 4.3 will describe the issue of non-transactional operations, and how it can be resolved.

4.1 Transactification of Class and Instance Fields

The transactification of class and instance fields replaces all fields in a class with VBox instances, so that accesses and modifications to them behave transactionally. Because the types of the fields change, transactification of fields may be visible to the outside of the class, and so classes that access those fields must be changed accordingly.

A class may be categorized according to the changes that transactification may cause either to that class or to other classes:

- *Contained IN*. A class is considered *Contained IN* if all of its fields are `private`. Transactification of this kind of class never necessitates changes to classes that use it.
- *Contained OUT*. A class is *Contained OUT* if it never accesses fields from an outside class. Transactification of a class `c1` that is used by a *Contained OUT* class `c2` never causes changes to `c2`.
- *Fully-Contained*. If a class is both *Contained IN* and *Contained OUT*, it is *Fully-Contained*. Transactification of a *Fully-Contained* class never causes changes to outside classes, and transactification of outside classes never causes changes to a *Fully-Contained* class.
- *Spilled*. If a class fits none of the other categories, it is considered *Spilled*. Transactification of a *Spilled* class may cause changes to outside classes, and transactification of outside classes may cause changes to a *Spilled* class.

These distinctions are important if only partial transactification is applied, and especially if there is interaction with unmodifiable classes. These issues shall be further discussed in Section 4.3.

The transactification process does the following:

- Replaces each original field *field* with type *type* of each class with a `private VBox<`*type*`>` named `$box_`*field*.
- Creates the get and put methods, `$box_`*field*`_get` and `$box_`*field*`_put`, which mediate accesses to the corresponding `VBox`. These methods have the same access level as the original field.
- Adds `VBox` slot initializations to the class constructors.
- Replaces accesses to the original fields, either from the same class or from outside classes, with calls to the `$box_`*field*`_get` and `$box_`*field*`_put` methods.

Figures 4.1 and 4.2 show the normal and transactional versions of a sample class, respectively.

Because a Java `interface` can have fields but it cannot have methods, and because all fields belonging to an `interface` are always implicitly `public static final`, direct accesses to those fields are preserved.

When a Java class `implements` an `interface`, it inherits its fields. To ease the transactification process, get and put methods for the fields inherited from interfaces are also inserted into those classes.

```
public class A {
    private String s;
    public List<Integer> l;

    public A(String s) {
        this.s = s;
    }

    public String s() {
        return s;
    }
}
```

Figure 4.1: Original `A` class.

4.2 Transactification of Arrays

4.2.1 VBoxes on Arrays

Apart from fields, arrays also have to be transactified. In the JVM, arrays are always one-dimensional, strongly typed, have a fixed size once created, and can either contain primitive types or object types. Arrays themselves are objects, and are types on their own. For instance, `int[]` is an object type of an array of primitive type `int`, `int[][]` is an object type of an array of objects of type `int[]`. Array transactification poses a multitude of problems, some of which cannot be solved fully without support from the JVM, as discussed in Section 4.2.5.

For an array to be transactional, all of the memory positions it contains must become transactional. This means that an `Object[]` becomes a `VBox<Object>[]`, an array with a `VBox<Object>` in each position. Multidimensional arrays in Java are just arrays that contain other arrays (but they are always *type-safe*), so an `Object[][]` is an array that contains in each position either an `Object[]` or `null`; likewise, a transactional version uses a `VBox<VBox<Object>[]>[]` that contains a `VBox<Object>[]` instance in each position (or `null`). An example of multidimensional array transactification is presented in Figures 4.3 and 4.4.

The alternative of implementing transactification of multidimensional arrays by changing an `Object[][]` to become a `VBox<Object>[][]`, a multidimensional array of VBoxes, does not work because whereas each position (i, j) on the array would become transactional, modifications to the array structure would not behave transactionally, as shown in Figure 4.5.

Note that these modifications are independent of the transactification of fields, which still has to be applied. A class containing a field `arr` of type `Object[]` has to be modified to hold a field of type `VBox<VBox<Object>[]>`, that is, a box of objects with type `VBox<Object>[]`, combining the transactification of the field itself with the changed type of the transactified array.

```
public class A {
    private VBox<String> $box_s;
    private VBox<List<Integer>> $box_l;

    public A(String s) {
        $box_s = new VBox<String>();
        $box_l = new VBox<List<Integer>>();
        $box_s_put(s);
    }

    public String s() {
        return $box_s_get();
    }

    private String $box_s_get() {
        return $box_s.get();
    }

    private void $box_s_put(String s) {
        $box_s.put(s);
    }

    public List<Integer> $box_l_get() {
        return $box_l.get();
    }

    public void $box_l_put(List<Integer> l) {
        $box_l.put(l);
    }
}
```

Figure 4.2: Transactified version of class `A` (Figure 4.1). Note that the field `s` was originally `private`, so `$box_s_get()` and `$box_s_put()` are `private`; the field `f` was originally `public` but became `private`, as all `VBox` fields are, and instead its accessor methods `$box_l_get()` and `$box_l_put()` are `public`.

```
Object[][] array = ...;  // Array creation

for (int i = 0; i < array.length; i++)
    for (int j = 0; j < array[i].length; j++)
        array[i][j] = new Integer(i+j);  // Write into array position

Object[] otherArray = ...;  // Unidimensional array creation
array[0] = otherArray;  // Write into array position
```

Figure 4.3: Normal multidimensional array access and manipulation.

```
VBox<VBox<Object>[]>[] array = ...;  // Array creation

for (int i = 0; i < array.length; i++)
    for (int j = 0; j < array[i].length; j++)
        (array[i].get()[j]).put(new Integer(i+j));  // Transactional modification

VBox<Object>[] otherArray = ...;  // Unidimensional array creation
array[0].put(otherArray);  // Transactional modification
```

Figure 4.4: Transactified version of the code presented in Figure 4.3, showing multidimensional array access and manipulation. All changes made are transactional because only VBoxes are modified.

4.2.2 Array Creation

In the JVM, when an array is created (using the opcodes `NEWARRAY`, `ANEWARRAY` or `MULTIANEWARRAY`), all of its positions are set to the default value of the array type. This operation is extremely fast and efficient on most JVMs, having little overhead.

A transactified array must contain boxes for each array position. Deciding when these boxes are created presents a tradeoff. Boxes can be created for each position at array creation time, adding a lot of overhead to array creation — consider that for an `Integer[1000]` instead of just allocating and zeroing a block of memory, a block of `VBox[1000]` must be allocated and zeroed, a thousand individual

```
VBox<Object>[][] array = ...;  // Array creation

for (int i = 0; i < array.length; i++)
    for (int j = 0; j < array[i].length; j++)
        array[i][j].put(new Integer(i+j));  // Transactional modification

VBox<Object>[] otherArray = ...;  // Unidimensional array creation
array[0] = otherArray;  // Non-transactional modification
                        // Change is not done to a VBox, so access to the old array[0]
                        // value is lost and the operation cannot be undone
```

Figure 4.5: Incomplete transactified version of the code shown in Figure 4.3, done by substituting the original multidimensional array with a multidimensional `VBox` array.

```
// Original version
Object[][] array = new Object[3];
Object o = array[0];

// Allocation on array creation
VBox<Object>[] array = VBoxArray.initializeArray(new VBox<Object>[3]);
Object o = array[0].get();

// Lazy allocation
VBox<Object>[] array = new VBox<Object>[3];
Object o = (array[0] != null ? array[0] : VBoxArray.initialize(array, 0)).get();
```

Figure 4.6: Allocation of VBoxes at array creation time versus lazy VBox allocation.

VBox objects must be created, and their constructors run, and each `VBox` object reference needs to be assigned to each array position. The alternative, where boxes are lazily allocated, adds no overhead to array creation, but each and every access must check for a `null` value where a VBox is expected, and the VBox must then be created and its reference put in the array, while taking care of possible parallel VBox creation data races. Figure 4.6 exemplifies these two options.

The first solution was chosen for ease of implementation: each array creation bytecode is replaced by a call to an initialization method that takes care of array creation and placing a VBox at each array position.

4.2.3 Array Access

Array accesses also have to be changed. When reading from an array position, after doing the normal array load to obtain the VBox from the VBox array, `VBox.get()` needs to called. For multidimensional arrays, at each intermediate access a `get()` is called, so that an access to `arr[1][2][3]` must be transformed into `((arr[1].get())[2].get())[3].get()`.

For array writes, an array load must be used to obtain the VBox which is going to be written to, and then `put()` is called for the write. For example, `arr[0] = "Hello"` becomes `arr[0].put("Hello")`. For multidimensional arrays, like for reads, `get()` must still be called for intermediate accesses, and `put()` is called for the final write — `arr[1][2][3] = "Hello"` is replaced by `((arr[1].get())[2].get())[3].put("Hello")`.

4.2.4 Arrays as Method Arguments

Passing an array as an argument in a method call is a common operation. Unfortunately, unlike the transactification of slots, in which a class of type `A` before transformation keeps its type `A` after the transformation, transactified arrays are not of the same type of their non-transactified counterparts. A method that accepted the type `String[]` cannot, without modification, accept the type `VBox<String>[]` in its place instead. Additionally, a transactified array cannot be temporarily converted to a regular array for method calls, and then converted back after the method returns, because the called method may save references to the temporary converted array, and further changes to it would not be reflected on the real array.

Thus, the only possible solution is changing method signatures of all methods receiving or returning arrays. As an example, method `m(String[])` becomes `m(VBox<String>[])`. This poses another problem, related to the implementation of generics in Java, which are done using *type erasure* [8]. During compilation, all generic type information is erased, and the previous example becomes `m(VBox[])`.[2] Because this information is removed, it is no longer possible to distinguish between two different versions of `m`, let's say `m(String[])` and `m(Byte[])`, as after erasure both share the same signature, `m(VBox[])`.

There are various possible solutions to this problem. A solution that presents no overhead is renaming the method, so that its name reflects the original argument types before transactification. Coming back to the previous example of `m`, the transactified versions become `m$$_java_lang_String(VBox[])` and `m$$_java_lang_Byte(VBox[])`. This scheme almost fully solves the problem: constructors cannot be renamed, and always have to be called `<init>`. For constructors, a low overhead alternative is re-adding the original argument types to the argument list, after all the other arguments — `<init>(A[])` becomes `<init>(VBox[],A[])`, and all calls to the constructor are modified to pass null values for those extra arguments.

4.2.5 Putting it All Together

In the previous section, we concluded that because a transactified array is not of the same type of a non-transactified array, *the only possible solution is changing method signatures of* ***all*** *methods receiving or returning arrays.*

And herein lies the real problem. It is not possible to change all method signatures of methods receiving or returning arrays on the Sun JVM, because it reserves the `java.*` package namespace and does not allow runtime loading of modified versions of classes within this package or any of its subpackages.

Because of this, array transactification is essentially unusable without either escape analysis to guarantee that an array is never going to be passed to an unmodifiable method, or some level of cooperation from the JVM.

Array transactification is thus currently disabled in the JaSPEx prototype.

4.3 Non-Transactional Operations

As already mentioned in brief in the previous sections, not all things can be transactified. `native` methods, which use the Java Native Interface [37] to interface with native code that runs outside the JVM, cannot be analyzed or modified easily. I/O operations cannot generally be undone. Another source of problems is the use of reflection, because it eludes the static transformation of accesses to fields. So, reflection has to be forbidden, or else modified to be transactification and speculation aware.

Also, the Sun JVM reserves the `java.*` package namespace and does not allow runtime loading of modified versions of classes within this package or any of its subpackages, which besides disallowing array transactification as described in Section 4.2.5, makes calls to built-in Java classes non-transactional operations, because side-effects potentially caused by them cannot be undone.

[2] Notice that all information between angle brackets has been removed.

I refer to classes that either cannot be changed, such as those living in the `java.*` package namespace, or should not be changed, such as the JVSTM libraries and the JaSPEx framework, as **unmodifiable** classes.

Because not all things can be transactified, the parallelization system must be able to detect all of these cases and make sure that such invocations are forbidden during speculative execution.

There are two main approaches to prevent the execution of nontransactional operations within a speculative execution: (1) static identification of these operations, as discussed in Section 4.3.1; and (2) dynamic, runtime prevention of their execution, as discussed in Section 4.3.2.

4.3.1 Static Identification of Non-Transactional Operations

Static identification consists of building a graph of possible method invocations, starting from the initial entry point of an application, with the objective of identifying both `native` methods and methods that *may* call `native` methods.

A method is considered as *may invoke native* if any of the methods it may call are also marked with either `native` or *may invoke native*, or if it can be overridden by a method in a subclass that *may invoke native*. Method override semantics have to be taken into account because of polymorphism — Figures 4.7 and 4.8 show an example where both `A.method()` and `A.anotherMethod()` are considered as *may invoke native* because `B.method()` overrides `A.method()` and is `native`.

A less trivial to analyze example, that uses the `java.util.ArrayList` Java library class is shown in Figure 4.9. In this example, no `native` calls are visible, but some of the `ArrayList`'s functionalities may trigger the execution of `native` methods, as shown in Figure 4.10.

```
class A {
    void method() { }

    void anotherMethod() { method(); }
}

class B extends A {
    native void method();
}

A o = new A();
o.anotherMethod();

A o = new B();
o.anotherMethod();
```

Figure 4.7: Example class `B` that extends `A` and overrides `method()` with a native version. If `anotherMethod()` is invoked on a reference with type `A`, a native method *may* be called, because a `B` is also an `A`, so `A`'s `method()` and `anotherMethod()` are considered *may invoke native* methods. (See also Figure 4.8)

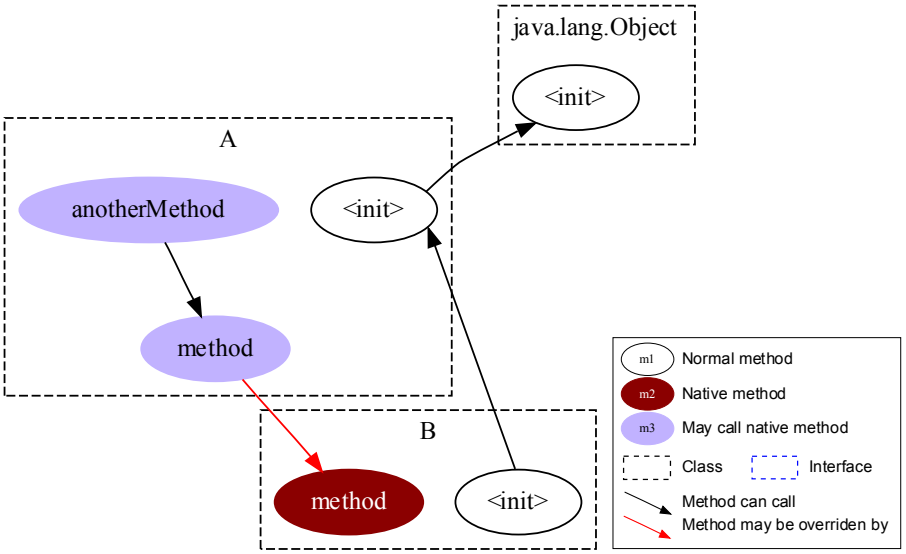

Figure 4.8: Static identification graph for the classes presented in Figure 4.7. If `anotherMethod()` is invoked on a reference with type `A`, a native method *may* be called, because a `B` is also an `A`, so `A`'s `method()` and `anotherMethod()` are considered *may invoke native* methods.

```
class StaticIdExample {
    private StaticIdExample() {
        List<String> lst = new ArrayList<String>();
        lst.add("Hello");
    }

    public static void main(String[] args) {
        new StaticIdExample();
    }
}
```

Figure 4.9: Example class that uses the `java.util.ArrayList` Java library class. (See also Figure 4.10)

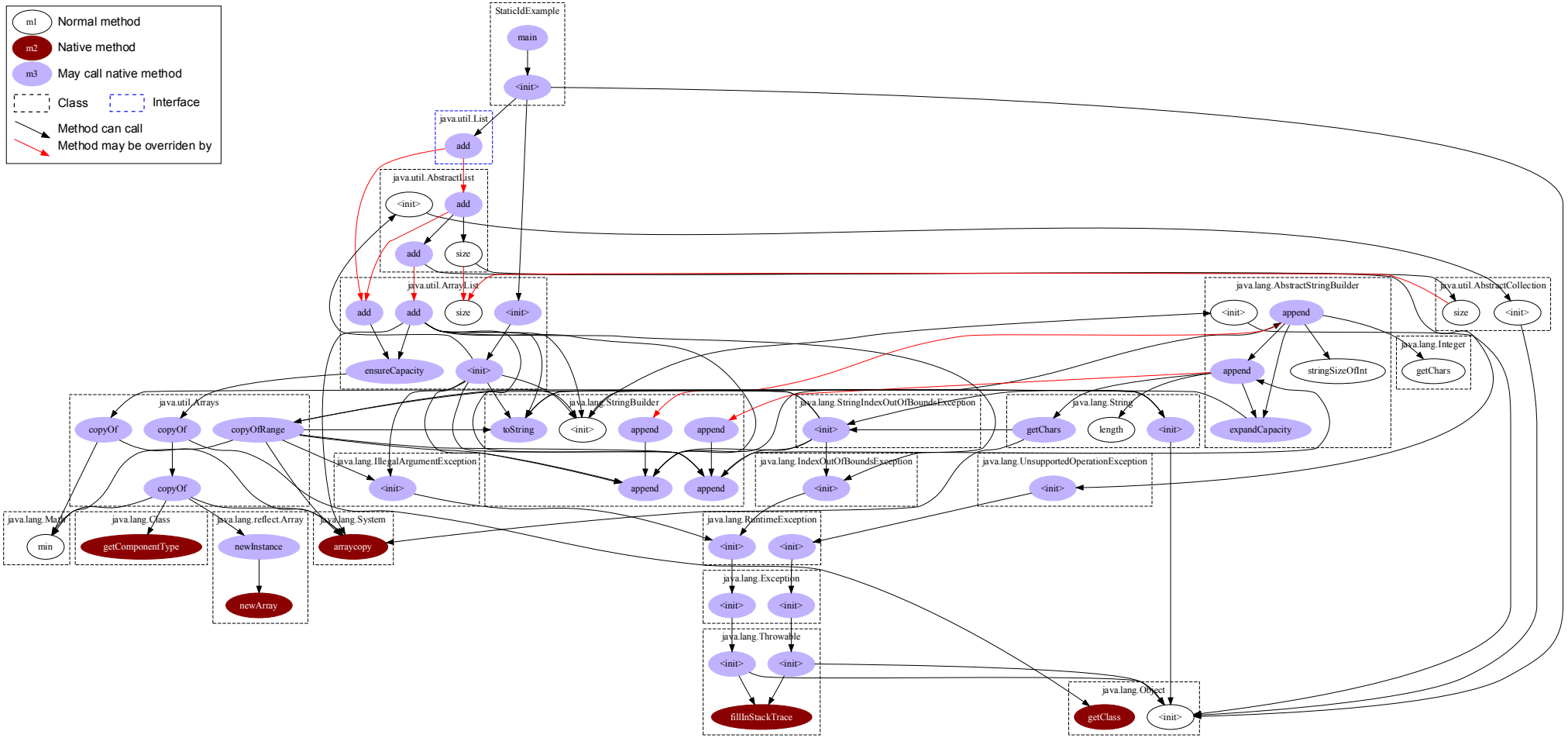

Figure 4.10: Static identification graph for the class presented in Figure 4.9. Most of the methods are considered *may invoke native*, even though there are comparatively few `native` methods.

Looking at the graph in Figure 4.10, we see that most methods are marked as *may invoke native*, even though there are comparatively few `native` methods.

Figures 4.11 and 4.12 show the breakdown between `native` methods, *may invoke native* methods, and normal methods for two large Java software applications: the NetBeans Java IDE,[3] and Jython,[4] a Java implementation of Python. As we can see, using a static approach means that a big percentage of methods are considered *may invoke native*, and as such, would not be able to run speculatively.

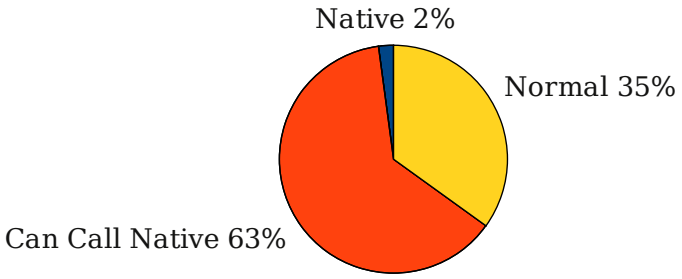

Figure 4.11: Breakdown of method classifications for the NetBeans IDE (version 6.7).

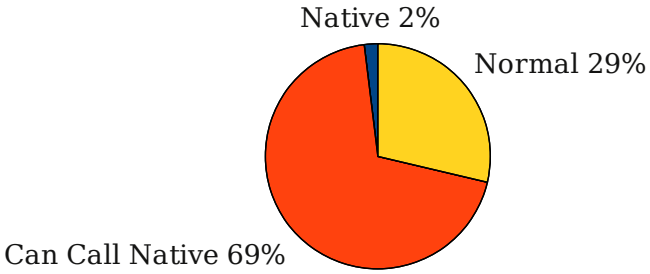

Figure 4.12: Breakdown of method classifications for the Jython interpreter (version 2.5).

Note that even if a method `A` may call a `native` method `B`, it does not mean that `A` calls `B` every time it executes. So, as this approach is very conservative, and disallows speculative execution of many methods, I opted for a dynamic, runtime approach, which is presented in the next section.

4.3.2 Runtime Prevention of Non-Transactional Operations

Because of the limitations of the static identification of non-transactional operations presented in the previous section, the current JaSPEx prototype uses runtime prevention instead.

Runtime prevention works by identifying all non-transactional operations in the bytecode, and by prepending a call to the speculation runtime before such operations. It is then up to the speculation runtime to decide if the operation can proceed, and when the call returns, the non-transactional operation may proceed; alternatively, the runtime may throw an exception, aborting the speculation and causing the operation to never proceed. This mechanism allows the framework to prevent the execution of these operations, or to take steps to ensure that they can safely proceed.

To support speculative execution, JaSPEx creates a speculative version of each method `M`, called

[3]See more information at `http://www.netbeans.org/`.
[4]See more information at `http://www.jython.org/`.

```
public class NonTransExample {
    public NonTransExample(String[] strings) {
        System.out.println(strings[0]);
    }
}
```

Figure 4.13: Example class to which we will apply the modifications needed for runtime prevention of non-transactional operations. (See also Figures 4.15 and 4.16)

```
public class NonTransExample {
    public NonTransExample(String[] strings) { ... }  // Same as original

    public NonTransExample(String[] strings, SpeculativeCtorMarker dummy) {
        SpeculationControl.nonTransactionalActionAttempted();
        System.out.println(strings[0]);
    }
}
```

Figure 4.14: Example class after application of the modifications needed for runtime prevention of non-transactional operations. (See also Figures 4.15 and 4.16)

`M$speculative`, except for constructors, which always have to be named `<init>`. In this latter case, an alternative scheme is used: A new parameter of type `SpeculativeCtorMarker` is added at the end of every speculative constructor. The speculative version of each method is a copy of the original method with invocations to other methods replaced by calls to their `$speculative` versions, if possible — this way, when doing speculation, the program always flows through `$speculative` methods, as expected. The modifications needed for runtime prevention of non-transactional operations are then applied to the `$speculative` version of a method.

Currently, the following operations are considered non-transactional:

- Invocation of a method on an unmodifiable class
- Accessing a field on an unmodifiable class
- Invocation of a `native` method
- Operations involving arrays, both creation and read/write accesses

When a bytecode corresponding to one of these operations is found in the classes being transactified, a call to `SpeculationControl.nonTransactionalActionAttempted()` is inserted before it, as shown in Figures 4.13-4.16.

Additionally, if the original method was `native`, its `$speculative` counterpart consists of a call to the JaSPEx runtime — `SpeculationControl.nonTransactionalActionAttempted()` — followed by a call to the original version. Similarly, because a class may inherit methods from an unmodifiable class, the framework needs to add `$speculative` versions of inherited methods that call the runtime and then the original method on the superclass. Both of these modifications are needed because when a class `c1` has

```
ALOAD_0
INVOKESPECIAL   java/lang/Object/<init>()V
GETSTATIC       java/lang/System/out Ljava/io/PrintStream;
ALOAD_1
ICONST_0
AALOAD
INVOKEVIRTUAL   java/io/PrintStream/println(Ljava/lang/String;)V
RETURN
```

Figure 4.15: JVM bytecode for the constructor method `NonTransExample()` shown in Figure 4.13, before application of the modifications needed for runtime prevention of non-transactional operations.

```
ALOAD_0
INVOKESPECIAL   java/lang/Object/<init>()V
INVOKESTATIC    speculation/runtime/SpeculationControl/nonTransactionalActionAttempted()V
// Access to a field on an unmodifiable class (java/lang/System.out)
GETSTATIC       java/lang/System/out Ljava/io/PrintStream;
ALOAD_1
ICONST_0
INVOKESTATIC    speculation/runtime/SpeculationControl/nonTransactionalActionAttempted()V
// Array access (load position 0)
AALOAD
INVOKESTATIC    speculation/runtime/SpeculationControl/nonTransactionalActionAttempted()V
// Invocation of a method on an unmodifiable class (java/io/PrintStream.println())
INVOKEVIRTUAL   java/io/PrintStream/println(Ljava/lang/String;)V
RETURN
```

Figure 4.16: JVM bytecode for the constructor method `NonTransExample()` shown in Figure 4.13, after application of the modifications needed for runtime prevention of non-transactional operations.

been transactified, all operations called on an object of type `c1` must be transactional: the first case protects against `native` methods in `c1`, and the second against methods inherited from unmodifiable classes. Figures 4.17-4.20 show examples of these modifications being applied.

```
public class NativeNonTransExample {
    public native void doSomething();
}
```

Figure 4.17: Example class with a `native` method.

```
public class NativeNonTransExample {
    public native void doSomething();

    public void doSomething$speculative() {
        SpeculationControl.nonTransactionalActionAttempted();
        doSomething();
    }
}
```

Figure 4.18: Transactified version of the class shown in Figure 4.17.

```
public class InheritedNonTransExample extends java.util.ArrayList {
    public void newFeature() { ... }
}
```

Figure 4.19: Example class that extends an unmodifiable class (`java.util.ArrayList`).

```
public class InheritedNonTransExample extends java.util.ArrayList {
    public void newFeature() { ... }  // Same as original
    public void newFeature$speculative() { ... }  // $speculative version

    // Overrides for methods inherited from ArrayList
    public boolean containsAll$speculative(java.util.Collection c) {
        SpeculationControl.nonTransactionalActionAttempted();
        return super.containsAll(c);
    }

    public boolean removeAll$speculative(java.util.Collection c) { ... }
    ...
    public java.lang.Object[] toArray$speculative(java.lang.Object[] o) { ... }
    public void trimToSize$speculative() { ... }
}
```

Figure 4.20: Transactified version of the class shown in Figure 4.19. The `$speculative` versions of the inherited methods invoke the JaSPEx framework before forwarding the method call to the superclass.

Chapter 5

Speculative Parallelization

After all the changes described in the previous chapter, code can now execute transactionally, and the JaSPEx framework has control over the execution of non-transactional operations. From here, we can now delve into the speculative execution of code.

JaSPEx supports two modes of code execution: (1) transactified execution, where code is run transactionally but no speculation occurs; and (2) a speculative execution mode, where code runs both transactionally and speculatively. The first mode is useful to establish a baseline execution time that can be compared to the second mode, speculative execution.

This chapter is organized as follows. Section 5.1 introduces the design chosen for speculative execution. Section 5.2 describes the modifications done to an application so that it can execute speculatively. Section 5.3 describes the runtime behaviour of speculations: how they are spawned, how the framework handles them internally, how they are executed and when they can commit their results. Finally, Section 5.4 finishes with a discussion of some of the issues faced by the current implementation.

5.1 Design

The idea of speculative code execution may be applied at different granularity levels, the smallest of which is running a single instruction speculatively. As the JaSPEx system runs without help from the JVM runtime, and it uses a a software transactional memory that imposes overheads on transaction spawning and commit, it is unpractical to do such a small granularity speculation.

In the JVM (and in Java), code is divided into multiple methods. To execute parts of a method speculatively, local variables would need to be transactified, imposing yet another overhead to code being run by the speculation system; the framework would also need to be able to jump to the middle of a method, starting execution from that point. As the JVM runtime does not allow jumping to the middle of a method — methods are called and execute from start to end, with jumps occurring only intra-method — a method would also need to be split up into multiple parts, to introduce the entry points needed for speculative execution.

To try and strike a balance between imposed overheads and granularity of speculations that would allow the system to successfully achieve speedups for applications, the JaSPEx framework does speculative

method execution. The idea of speculative method execution is similar to the example originally presented in Figures 3.1 and 3.2, and shown here again in Figures 5.1 and 5.2. When a method (the *caller*) is entered, some of the methods that it calls (the *callees*) may be run speculatively. In the example shown, when `method` starts, `doA` and `doB` are run speculatively.

```
void method() {
    doA();
    doB();
}
```

Figure 5.1: Example method to be parallelized. The method `method` is the caller, `doA` and `doB` are the callees.

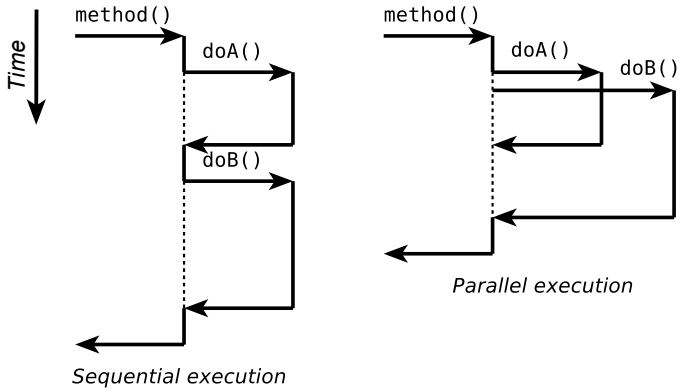

Figure 5.2: Sequential and parallel executions of `method`.

This strategy of spawning speculative executions when the caller is entered avoids both the need for transactification of local variables and the need to split a method into smaller methods, so that the speculation system has entry points to the middle of methods. Note that method invocations inside loops are speculated at most once; further invocations are executed normally in the thread of the caller (but may still spawn speculations of their own).

But why start the speculative execution of a called method at the beginning of the caller, and not further down at the real invocation site of the callee method? To answer this question we again need to take into account the limitations imposed by not having any support from the JVM. Consider the example shown in Figure 5.3: The current design will queue `doC` for speculative execution at the beginning of `someMethod`, resulting in the execution of `doC` being speculative, because it might run *before* or in parallel with the `Code Block 1`, whereas originally it would be run *after* it.

If instead, we waited until we reach the invoke site for `doC`, `doC` would not be running speculatively, and instead it would be `Code Block 2` that would speculatively run *before* or in parallel with `doC`, instead of *after it*. This means that we would again need to be able to revert changes made to local variables, because local changes done by `Code Block 2` might need to be undone, and also we would need to be able to rerun only `Code Block 2` instead of the whole method, so it would have to be replaced with a `do { ... } while (!speculationSuccessful);` block.

Until now, the design being described ignores the issue of function parameters for methods that are

```
void someMethod() {
    { ... } // Code Block 1
    doC();
    { ... } // Code Block 2
}
```

Figure 5.3: Method which will spawn speculative executions.

speculatively executed. If `doC` receives some parameters as input, these need to be available, also, for a speculative execution of `doC`. To solve this problem, there are two possible strategies. The first one is to perform an analysis on `someMethod`, and try to determine statically how to obtain the parameters needed for a speculative execution. The simplest example for this is shown in Figure 5.4, where `doD` receives a constant value (2) as a parameter.

```
void anotherMethod() {
    doD(2);
}
```

Figure 5.4: Example of simplest case for discovering the parameters of a method that will be speculatively executed: in this case, `doD` always receives as a parameter the `int` 2.

The second possible strategy for identifying the parameters for a speculative execution is the usage of value prediction, similar to the approaches proposed in [38, 14]. The idea is to speculate also on the values of the parameters, and to later verify if they matched the real values obtained at the method invocation site. Multiple schemes like last-value prediction (LVP) or stride-value prediction (SVP) might then be used to obtain a possible value for the parameters.

Currently, JaSPEx relies only on the first strategy of static analysis to obtain the parameters for the speculative executions. It supports speculation when constants and parameters of the caller are used (if they are not written to before the function call), and all arithmetic, logic, or bitwise operations on these. Figure 5.5 shows an example, based on Figure 2.2, where JaSPEx is able to determine the arguments for the function call to `setRandom`.

```
void yetAnotherMethod(long seed, int bits) {
    { ... }
    setRandom( ((seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1)) >>> (48 - bits) );
    { ... }
}
```

Figure 5.5: This example, based on Figure 2.2, shows a complex case where JaSPEx is able to statically determine the argument for the speculative execution of `setRandom`.

As a final example, consider a recursive implementation of the Fibonacci function shown in Figure 5.6.

At each call to `fib`, JaSPEx speculatively launches the execution of `fib(n-1)` and `fib(n-2)` and then proceeds with the execution of the method: In the case where `n` $\leq$ `1`, the speculative executions that

```
public int fib(int n) {
    if (n <= 1) return n;
    return fib(n-1) + fib(n-2);
}
```

Figure 5.6: Recursive implementation of the Fibonacci function.

may be running are discarded; otherwise, their results are retrieved and the transactions that they are running in are committed.

5.2 Transformations for Speculative Execution

As described in section 4.3.2, for each method in each transactified class, a version for speculative execution (the `$speculative` version) is created. This version is fully transactional, containing all of the changes mentioned in the previous chapter.

I shall now describe the final round of modifications needed for speculation. These allow the speculation system to know when it can spawn a speculative execution, and when the speculation results should be applied or discarded. Before these changes are made, a copy of each `$speculative` method, the `$non_speculative` version, is created. The `$speculative` version can then receive the final round of modifications. Having both versions of a method allows the JaSPEx framework to skip speculation on a method, and just run a transactified version of it. This is useful, if for example, too many speculations are active already.

Methods that will spawn speculations call the JaSPEx runtime when they are started, before they terminate, and to get results from the speculative executions. When the caller method starts, it invokes the framework method `SpeculationControl.entryPointReached`, passing as argument an entry-point id that uniquely identifies the caller method. The call to `entryPointReached` returns either an instance of `SpeculationId`, that identifies the current dynamic execution context uniquely, signalling that speculation can proceed, or `null`, signalling that speculation should not proceed, and that the `$non_speculative` version of the method should be invoked.

If the speculation can proceed, a call to `SpeculationControl.startSpeculation` is done, passing as arguments both the `SpeculationId` and an array of arrays with the arguments for each function call that is to be executed speculatively within that method. For instance, in the `fib` example in Figure 5.6, `this.fib(n-1)` and `this.fib(n-2)` will be speculatively executed.[1] Thus, `startSpeculation` will receive an array `arr` of type `Object[2][]`, where `arr[0]` contains the arguments `this` and `n-1`, and `arr[1]` contains `this` and `n-2`.

Before a method exits, it calls `SpeculationControl.exitPointReached`, to inform the runtime that the method will terminate, and that speculative executions that may be queued or running for this method should be discarded. The current form in which the call to `exitPointReached` is injected does not take into account exceptions yet; if a thrown exception causes this method to never be called, the application will continue to behave correctly, but pending speculative executions may be left in the system.

[1] Because `fib` is not a static function, each recursive call also implicitly includes as an argument the current object instance, `this`.

Method invocations for methods that are executed speculatively are replaced by calls to `SpeculationControl.getResult`, which, given the current `SpeculationId` and an identifier that identifies the function call, returns a `Future` object that represents the result of the speculative execution. Finally, to obtain the result, `get()` is called on the Future; if the underlying method execution resulted in an exception being thrown (an instance of `java.lang.Throwable` or any of its subclasses), that exception will be rethrown by `get()`. Figure 5.7 shows the `fib$speculative` method with these modifications.

```
public int fib$speculative(int n) {
    SpeculationId specId = SpeculationControl.entryPointReached(ENTRY_POINT_ID);
    if (specId == null) {
        return fib$non_speculative(n);
    }
    SpeculationControl.startSpeculation(specId, new Object[] { new Object[] { this, n-1 },
                                                               new Object[] { this, n-2 } });
    if (n <= 1) {
        SpeculationControl.exitPointReached(specId);
        return n;
    }
    Future f0 = SpeculationControl.getResult(specId, INV_ID_0);
    Future f1 = SpeculationControl.getResult(specId, INV_ID_1);
    int temp = f0.get() + f1.get();
    SpeculationControl.exitPointReached(specId);
    return temp;
}
```

Figure 5.7: The final speculative version of the Fibonacci function. `ENTRY_POINT_ID` uniquely identifies the `fib` method. The symbols INV_ID_* identify the method calls that they replace: In this case, INV_ID_0 represents the call to `fib(n-1)`, whereas INV_ID_1 represents the call to `fib(n-2)`.

5.3 Doing Speculation

As seen in the previous sections, calls to methods belonging to the framework class `SpeculationControl` are added at various points of the speculative methods, allowing control of speculation start and end, decision on how to proceed when nontransactional actions need to be executed, and fetching of results from speculative executions. I shall now describe how the framework internally manages these operations.

5.3.1 Caller Method

When the caller method — the method for which we will speculatively execute the callees — starts, it tries to obtain an instance of `SpeculationId`. This instance of `SpeculationId`, which uniquely identifies each dynamic execution context where speculation is being done, is created by a call to `SpeculationControl.entryPointReached`.

Each `SpeculationId` points to an instance of a class called `MethodSpeculationInfo`, which contains, for each method in the entire application for which we may start speculative execution, a list of the methods that it invokes; this list is harvested during the transformation of a class for speculative execution, and is converted to a list of `java.lang.reflect.Method` instances, which can be invoked using the JVM *reflection* support.

After the call to `SpeculationControl.entryPointReached` returns a `SpeculationId`, speculation can start; `SpeculationControl.startSpeculation` is then called with the `SpeculationId`, and with the arguments for the speculative executions. The `startSpeculation` method stores these arguments inside the `SpeculationId`, and creates an empty `ExecutionTask` array, that will keep the individual speculative execution tasks — each task represents a speculative execution of a callee. For instance, in the `fib` example in Figure 5.6, two tasks will be created to represent the executions of `this.fib(n-1)` and `this.fib(n-2)`.

Finally, `SpeculationControl` hands over the `SpeculationId` to the `Scheduler`, which saves it in a queue for further processing. At this point, execution of the caller method may proceed — remember that all of these steps are injected at the beginning of the method. Figure 5.8 shows a sequence diagram of these steps.

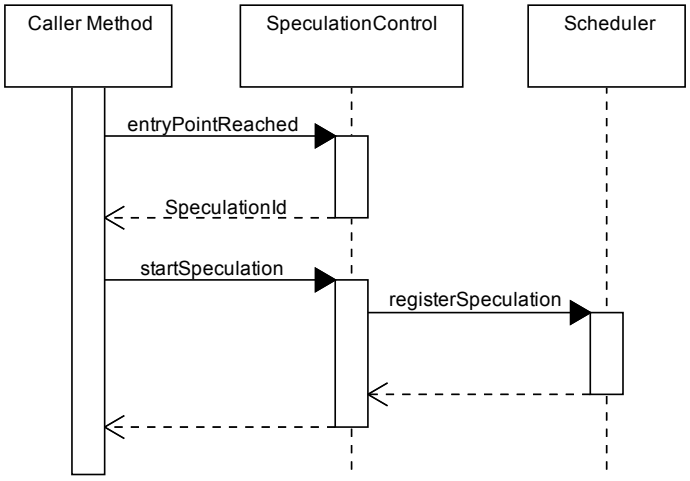

Figure 5.8: Sequence diagram showing the steps needed to queue a speculation for execution.

5.3.2 `ExecutionTask` Generation and Execution

When a `SpeculationId` is registered for execution on the `Scheduler` class, it is put in a queue. This queue is processed by a thread, the `speculationIdSplitThread`, that splits the `SpeculationId` into multiple `ExecutionTask` objects, one for each callee that is going to be speculatively executed. These tasks are then submitted for execution by the `Executor` class.

The `Executor` class extends the `java.util.concurrent.ThreadPoolExecutor` class, and provides an unbounded pool of threads to execute tasks. This class tries to reuse idle threads that have no tasks to process, and if no threads are available, it spawns a new thread to execute the submitted task.

The `ExecutionTask` class encapsulates a speculative execution. It implements the `java.util.concurrent.RunnableFuture` interface, allowing it to act both as a `Runnable` object, which can be run by the `Executor`, and as a `Future`, that the caller uses to obtain the value or exception returned from the speculative execution.

Internally, an `ExecutionTask` keeps a state machine to control the different phases of speculation. The current state can be changed by the worker thread that is running the task, the `speculationIdSplitThread` when it wants to signal speculation abortions, and the caller thread, when it wants to obtain the value

from the task. The state machine mechanism also allows threads to wait for certain states to appear: for example, a caller thread that wants to obtain the result of the speculative execution will need to wait until the task is in the `DONE` state. Table 5.1 describes the possible states of the `ExecutionTask` state machine, and Figure 5.9 shows the possible transitions between states. The `ExecutionTask` also keeps two `boolean` flags, `hasCommitToken` and `abortRequested`, that as we will see are used by other threads to communicate with the worker thread that is running the task.

A new task starts in the `NEW` state. When a worker thread picks up a task, it moves the task to the `RUNNING` state, starts a new STM transaction, and uses reflection to invoke the target method. Because the method executes within an STM transaction, none of its changes are visible to the outside until the transaction commits. In the `RUNNING` state multiple things may happen: the method being speculatively run returns or throws an exception; a non-transactional operation needs to be executed; or this speculation triggered the spawning of other speculative tasks, and it needs to obtain values from them. For all of these cases, the `waitCurrentTransactionCommit` method is called.

The `waitCurrentTransactionCommit` method first verifies if there is an STM transaction active for the current thread. If there is no transaction, it means that this worker thread has already committed its transaction, and is running in program order, as I will explain in the next section. If there is a transaction, then it checks if it has the *commit token* that allows it to commit. This *commit token* is set by the caller thread on a worker thread when the caller invokes `get` on the future representing the task. If no token is found, it checks if there has been a request to cancel this task — if for example, its result will not be needed — and aborts the transaction if such a request is found. If a task can neither commit nor needs to abort, it goes into the `WAIT` state, where it waits for one of these conditions to change. When another thread changes one of these conditions (`hasCommitToken` or `abortRequested`), it changes the state to `PARENT`, to inform the worker thread that something changed, and it should execute the steps in `waitCurrentTransactionCommit` again.

Inside `waitCurrentTransactionCommit`, when a worker thread possesses the *commit token* it tries to commit its STM transaction. This operation may fail because the STM detects that the speculative execution read a box that was written to in the meanwhile, and a `RetrySpeculationException` is thrown to signal that this task should be reexecuted. If the transaction successfully commits, further calls to `waitCurrentTransactionCommit` will simply return right away, and the execution will run until the method being speculatively run returns, and its return value or thrown exception is saved. After this, the worker thread moves the task to the `DONE` state, and goes on to execute another task; the only remaining step is for the caller thread to collect the saved results of the execution, and to move the task to its final state, `CLOSED`.

5.3.3 Transaction Commit and Original Program Order

As detailed in the previous section, an `ExecutionTask` needs the *commit token* before it can commit. When a task commits, its results become visible to code executing outside a transaction, and to other future transactions; its results stop being speculative and become definite.

To maintain the original serial execution semantics of an application, changes done by speculative executions have to be made visible to the rest of the application in the same order that they would originally be. This is where the idea for the *commit token* comes from: a *commit token* is only given to a task at the point where it would be executed in the original program order, and, so, after committing, a task keeps executing in the program order until it finishes.

`NEW`	New, unstarted task
`RUNNING`	Task is executing
`WAIT`	Task is waiting for the *commit token*, allowing it to commit its results so far, or for an abort request
`PARENT`	Either the *commit token* or an abort request has been handed out by the parent thread
`DONE`	Task has finished executing and is waiting for the caller to collect the results
`CLOSED`	Task has finished executing and results have been collected
`ABORT`	Task execution has been aborted

Table 5.1: Listing of the states in the `ExecutionTask` state machine.

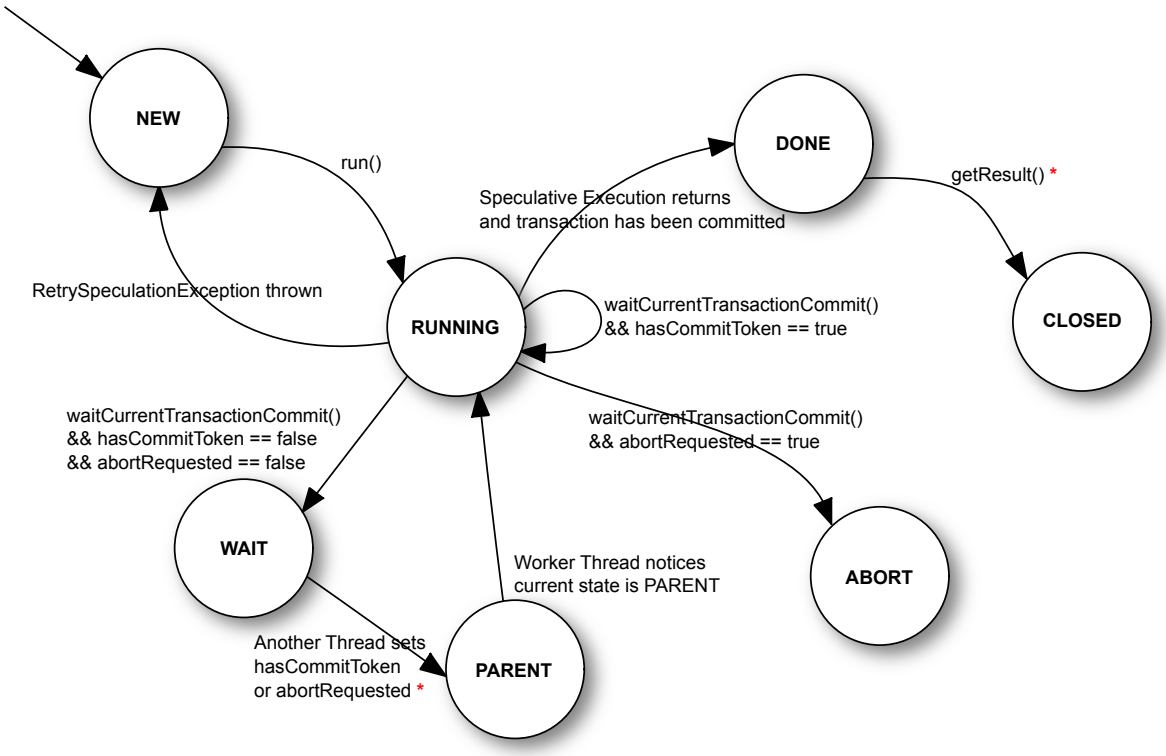

Figure 5.9: `ExecutionTask` state machine. Transitions are caused by the worker thread that is running the task, except for the ones marked with an *.

The only way a thread that is executing in program order can get the results from a speculative execution that it spawned is to "give" the callee `ExecutionTask` its *commit token*, and to wait until it reaches the `DONE` state. This scheme mimics the original sequential method call semantics: the caller invokes the callee, and waits for it to return; only one of them is executing at a given time.

This scheme results in only one thread executing in program order at any given time — other running threads, if any, are executing code speculatively — and original program ordering semantics is always respected.

5.3.4 Obtaining Results from a Speculation

After spawning a speculative execution, the caller method execution continues normally, until it reaches the call site for a method that has been speculatively executed. To obtain the result from the speculative execution, a call to `SpeculationControl.getResult` is made, that returns a `Future` object representing the result of the speculative execution (which can be `void`). This `Future` object is just the `ExecutionTask` object.

To obtain the result from the `Future`, `get` is called, which further calls the `getResult` method shown in Figure 5.9. Inside this method, `waitCurrentTransactionCommit` may be called, if the current caller is itself running speculatively; if not, the *commit token* is set on the task, and the caller waits until the task reaches the `DONE` state.

Upon reaching the `DONE` state, the task is moved to the `CLOSED` state, and an `ExecutionResult` object is retrieved, that wraps the final return value or exception for the callee. This object is then unwrapped: if the callee returned void, no value is returned; if the callee returned a primitive type, its value is unwrapped from its correspondent object type, and put on the caller's stack; if the callee returned a plain object, it is also put on the caller's stack; and if the callee threw and exception, the exception is rethrown inside the `get` method.

5.3.5 On Reusing Waiting Threads

I briefly mentioned in the description of the `Executor` class in Section 5.3.2, that "The `Executor` class (...) provides an **unbounded** pool of threads to execute tasks".

"Why an unbounded pool?" we may ask. This decision stems from the fact that each thread runs a single task until completion. Imagine that a fixed-size pool is used, such that there is exactly one thread for each one of n cpu cores, with n being the total number of cores in the system, which would be the expected solution. Because there are a lot of non-transactional operations that may cause a speculative execution to need to commit, and there can be only one thread running non-speculatively (in program order) at any given time, it means that a lot of the time, just one of the n threads would be working.

Consider the example shown in Figure 5.10, and consider that the method `yaMethod` is run on a machine where $n = 4$ and with a bounded pool of threads. In this case, four tasks could be generated, each representing a call to `startExpensiveCalculation`, and a worker thread could be allocated to each of these tasks. The problem is that because before starting the real calculation, a non-transactional operation needs to be executed, only one of the threads would be doing work at any given time, because

only one thread can be executing in program order, and the others would be stuck at the `println` operation.

To get around this, JaSPEx also speculatively spawns tasks to execute `doExpensiveCalculation` while their callers are doing their work — that in this case is just printing a simple string. The problem is that if a bounded pool of threads is used, there would be no available worker threads to speculatively run `doExpensiveCalculation`, because they were all stuck waiting for permission to execute `println`. So the solution chosen is to allow an unbounded pool of threads, that in this case would cause four other threads to be spawned, that would all be doing useful work while their parents were waiting for the *commit token* that they needed. To avoid spawning too many threads, JaSPEx stops submitting tasks for speculative execution when it observes that most threads in the thread pool are busy, and that the number of active threads has grown over a configurable value.

But isn't the entire point of the thread pool to reuse threads? Why doesn't the speculative execution system reuse threads that are just waiting, much like the OS switches the executing process on a cpu core when it too is waiting for some event? Despite that being the best solution, especially if the JVM does not use green threads,[2] the answer is that it is very hard to give a new task to a thread and then switch between the newer and older tasks that have been assigned to that thread.

One of the reasons it is not possible to reuse threads that are waiting can be shown with the example in Figure 5.11. Consider that two threads are used. When the thread running in program order reaches the beginning of `yaMethod2`, two tasks are created: one for running `doExpensiveA` and another for running `doExpensiveB`.

As the system is using only two threads, there is only one worker thread, and it starts execution of `doExpensiveA`; meanwhile, the thread running `yaMethod2` runs `Code Block 1`, and reaches the call site for `doExpensiveA`. If the worker thread is still not done with its work, we could set its *commit token*, and instead of waiting for results, the thread that was running `yaMethod2` could pick up the remaining task for execution, in this case, the speculative execution of `doExpensiveB`.

[2]Green threads differ from native operating system threads in that they are entirely managed in user space, and the switching is done by the JVM or by a library without the intervention of the operating system. It is possible for multiple green threads to co-exist inside a single native thread, with of course only one of them being active at a time.

```
public void yaMethod() {
    startExpensiveCalculation(1);
    startExpensiveCalculation(2);
    startExpensiveCalculation(3);
    startExpensiveCalculation(4);
}

public void startExpensiveCalculation(int i) {
    System.out.println("Starting expensive calculation");
    doExpensiveCalculation(i);
}

public int doExpensiveCalculation(int i) { ... }
```

Figure 5.10: Example to illustrate the problem of using a bounded thread pool.

```
public void yaMethod2() {
    { ... } // Code Block 1
    doExpensiveA();
    { ... } // Code Block 2
    doExpensiveB();
}

public void doExpensiveA() { ... }

public void doExpensiveB() {
    { ... } // Expensive Calculation
    System.out.println("Ending expensive calculation B");
}
```

Figure 5.11: Example to illustrate the problem of reusing waiting threads.

The problem is what needs to be done upon reaching the `println` method call at the end of the call to `doExpensiveB`. At this point the second task does not have the *commit token*, and is still speculative, because `Code Block 2` inside `yaMethod2` hasn't yet been executed. So the system would have to jump back to the point where the initial call to `doExpensiveA` occurred, retrieve the results from the speculative execution, execute `Code Block 2` until reaching the call site of `doExpensiveB`, jump back inside `doExpensiveB`, commit its results so far, and then continue with executing the `println` method call and finishing execution of the task.

And here we reach the issue of the limitations imposed by the JVM again: the complete inability of an application to manipulate or examine the stack of a JVM thread. Because there is no way to manipulate the stack of the current thread in the JVM, the only solution to the need to switch between contexts as in the example given is to run things inside different threads, and leave the switching up to the JVM or the OS.

5.3.6 Using the JVSTM for Speculative Execution

As described before, the JVSTM is used to supply the transactional support for the JaSPEx framework. Applications are modified to use VBox instances, making management of concurrent accesses, commit of changes, and undo of changes made by an aborted transaction transparent.

Calling `Transaction.begin()` starts a transaction and associates it with the current thread, `Transaction.abort()` aborts the current transaction, and `Transaction.commit()` tries to commit the current transaction — it can either return, indicating success, or throw a `CommitException`, indicating failure. VBoxes are accessed with `put()` and `get()`, as we saw in section 2.9.

A VBox can be accessed and changed with or without an active transaction on the current thread. If `get` is invoked outside a transaction, the JVSTM does the equivalent of starting a normal read-only transaction, reading the value from the box, and committing the read-only transaction, which, due to JVSTM's design, always succeeds.[3] If, instead, `put` is invoked outside a transaction, the JVSTM again does the equivalent of starting a read-write transaction, writing the value to the box, and committing

[3]Originally, the JVSTM did exactly this. An optimized implementation of this operation later improved the speed of this operation, while guaranteeing the same semantics.

the transaction, which again cannot fail because a commit of a read-write transaction that does not read anything always succeeds.[4] This model means that a read or a write operation outside a transaction always reads or writes the latest value of a VBox.

We have seen in section 2.9 that every transaction in the JVSTM has a version number. When a value of a VBox is read inside a transaction, the VBox returns not always its latest value, but a value that has an equal or smaller version than the current transaction version. When a value is written to a VBox inside a transaction, it is put on the transaction's write-map, and is visible to other threads only after the transaction is committed.

The approach used by JaSPEx is for code that is running in program order (not speculatively) to run outside a transaction, causing its changes to always be observed by transactions started after the changes; code that is executing speculatively executes inside a transaction.

Unfortunately, JVSTM's linear nesting model is too limited for it to be usable for speculative execution, because a nested transaction always has to execute in the same thread of the parent, and there cannot be any concurrency between sibling transactions that share the same parent. This poses a problem for speculations that are spawned by other speculations, because ideally they should have access to the modifications that are done by their parent, even if they are not committed yet (because their commit is dependent on their parent's commit), but they need to be able to execute independently from and concurrently with their parents, and it is also very common for a single parent to spawn multiple sibling speculations.

Thus, JaSPEx starts only top-level transactions for all speculations, and there are no nesting relationships inside transactions, even if the speculations themselves have implicit parent/child relationships. Still, note that the commit scheme used by `ExecutionTask` (Section 5.3.2) implicitly enforces these relationships. This means that there is no sharing between a parent speculation and its child: if the child starts its own transaction before the parent commits its own, the child will not see any of the changes made by his parent, and will abort if the parent writes to a value that the child reads. If the parent is running in program order, this problem is alleviated, because its speculative children will see all of the changes the parent made up until the time when their transaction started.

The JaSPEx framework uses an unmodified, upstream version of the JVSTM. The only change needed to the original behaviour of the JVSTM is implemented as an extension that changes the semantics of a read-only transaction. The default semantics of the read-only transaction define that a transaction that only does reads always commits successfully, because it can be linearized at the time when it started. The problem with this semantics is that it could happen that a method produces invalid results based on a stale value that it read, and the transaction it executed in would still be able to commit, which would result in a violation of the original program order.

As an example, consider Figure 5.12. Assume that the code at the bottom is being executed, and that a speculation is spawned to execute `startOperation`. When the instance of `TransactionSemanticsExample` is initialized, as no value is given to n, the JVM uses the default value for the `int` type, 0. Now, at the start of the method `startOperation`, a speculative execution of `doOperation` is spawned. The problem is that because a child will not be able to see the modifications that its parent made, when `doOperation` starts, n will still be 0, and the `AssertionError` will be triggered. As the transaction containing `doOperation` did not do any writes, it will happily commit its result when it receives the *commit token*, and the parent will receive the `AssertionError`, causing a deviation from the sequential execution semantics.

[4] This operation, too, was optimized, by using a special lightweight read-write transaction.

To solve this problem, JaSPEx changes the semantics of JVSTM's read-only transactions to force their linearization to occur at the point the commit is made, similarly to the way it that is done when both reads and writes are done. In the presented example, this would cause the JVSTM to detect that `doOperation` read an old version of n and cause the task to reexecute.

5.4 Discussion

In the design section (5.1) I already argued for some of the design compromises that I had to make to do speculation on top of and unmodified plain Java Virtual Machine that offered no such support, but I will now expand a bit more on some of these issues.

The Java Virtual Machine is a complex platform, designed for performance and portability. Its latest versions also carry a considerable backward-compatibility luggage, in terms of obsolete library classes, methods, and interfaces; and of features such as *generics* that are implemented in more limited ways so as to not disturb older code. This all means that the VM and its behaviours are not easily extended, and there is little meta-programming support, mainly limited to inspecting objects and classes, and to invoking methods.

For all these reasons, designing a speculation system for Java is complex. Even small applications normally exercise large amounts of features of the JVM platform, making it hard to partition these features into subsets that can be incrementally dealt with, while still successfully and correctly allowing speculative executions to run.

There are quite a lot of changes that are made before a class is ready for speculative execution. From transactification and handling of non-transactional operations to spawning speculative executions, and allowing speculation to be skipped for individual executions of a method, a class goes through many

```
public class TransactionSemanticsExample {
    private int n;

    public void startOperation(int arg) {
        if (arg > 0) n = arg;
        else n = 1;

        doOperation();
    }

    private void doOperation() {
        if (n <= 0) throw new AssertionError("This should never happen");
        { ... }
    }
}

TransactionSemanticsExample e = new TransactionSemanticsExample();
e.startOperation(2);
```

Figure 5.12: Example to illustrate the problem with JVSTM's original semantics for read-write transactions, when used by the JaSPEx system to do speculative execution.

changes, most of them adding overheads to code execution.

The design of the speculation part of the system tries to balance some of these overheads with the flexibility needed for successfully tapping parallelism in an application, by doing speculative method execution at the beginning of caller methods. Unfortunately, there are inherent limitations in this scheme, especially concerning the issue of dealing with the parameters needed for speculative executions. The use of an unbounded thread pool is also a non-optimal solution, that had to be adopted because of another limitation of the JVM.

The JVSTM, being a general-purpose software transactional memory, provides features that go unused on a speculative execution system such as the support for speculative read-only transactions, and lacks features that would help extract more performance on such a system, such as supporting a more advanced nested transaction model.

All of these limitations point to future work opportunities for the JaSPEx framework that are discussed in Section 7.2.

Chapter 6

Experimental Results

In this chapter I will present experimental results obtained with the current development version of the JaSPEx framework. I will start by presenting, in Section 6.1, measurements of the overheads incurred by applications after applying both transactification and the modifications needed for speculative execution, but without doing any speculation; then, in Section 6.2 I will present results that include speculative executions.

All of the results were obtained on a dual-quadcore system with two Intel Nehalem-based Xeon E5520 processors, running Ubuntu Linux 9.04, and Java SE version 1.6.0_13. To simulate execution on a machine with fewer processing cores, the extra cores were completely disabled by the operating system.[1]

6.1 Transactification

Before application code can be run speculatively, it must first be modified to run transactionally, so that the JaSPEx system is able to detect conflicting speculative executions, and reverse the changes made during them.

Transactification of an application imposes overheads to its execution. To measure those overheads, I have benchmarked execution of multiple applications, comparing their original execution times to those of their transactified counterparts.

The applications tested were:

- Java Grande Forum Benchmark Suite [39, 40] (version 2.0).[2] This benchmark suite encompasses multiple benchmarks, divided into three sections: low level operations (section 1), kernels (section 2), and large scale applications (section 3). Benchmarking was done with a subset of section 3 applications (`MolDyn`, a molecular dynamics simulation and `MonteCarlo`, a monte carlo simulation).
- Nativegraph. Nativegraph is a part of JaSPEx that is currently unused for transactification and speculation, but that does static identification of non-transactional operations and outputs the graphs shown in Figures 4.8 and 4.10.

[1] See `Documentation/cpu-hotplug.txt` on the Linux source tree.
[2] See also `http://www.epcc.ed.ac.uk/research/java-grande/`.

Benchmark	Original Runtime (s)	Transactified Runtime (s)
JGrande `MolDyn`	2.06	24.82
JGrande `MonteCarlo`	3.16	37.47
Nativegraph	23.9	96.29
JScheme `quicksort`	1.42	22.22
JScheme `fibonacci`	1.56	51.79
JScheme `ackermann`	3.08	117.25

Table 6.1: Original runtime versus transactified runtime for the presented benchmarks.

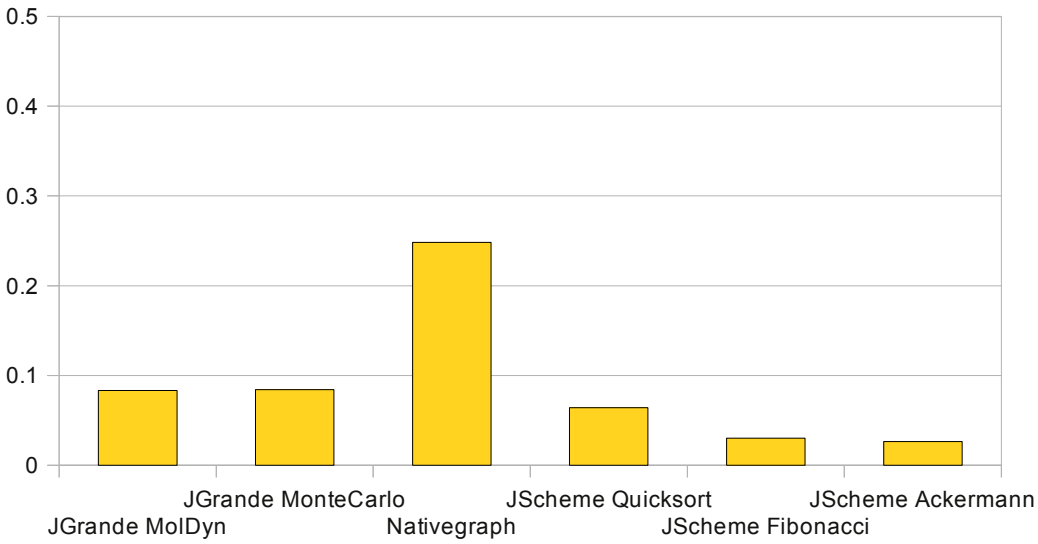

Figure 6.1: Relative slowdown for each of the benchmarks presented in Table 6.1.

- JScheme (version 7.2).[3] JScheme is an almost-compliant R4RS Scheme implementation that works on top of the JVM platform. Testing was done with example Scheme implementations of the `quicksort` algorithm, a recursive implementation of the `fibonacci` function and the `ackermann` function.

Table 6.1 shows the original and transactified runtimes for the various benchmarks, and Figure 6.1 shows the relative slowdown for each of the benchmarks.

As we can see, the transactification process imposes quite big overheads for applications, in account of all the extra method calls and indirections that are added.

6.2 Speculative Execution

I now present some preliminary results of the automatic parallelization performed by JaSPEx.

Figure 6.2 compares the transactified execution time of the Nativegraph application, with the execution times when speculative parallelization is used.

In Figures 6.3 and 6.4 I present the benchmark results of applying automatic parallelization to the

[3]See also `http://jscheme.sourceforge.net/jscheme/main.html`.

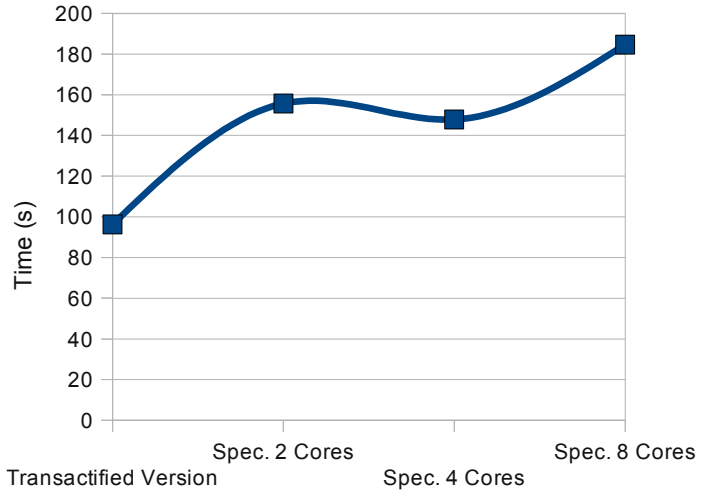

Figure 6.2: Transactified and speculative execution times for the Nativegraph benchmark. Shown is the runtime for the single-threaded transactified version, and the runtimes for speculative parallel execution with an increasing number of processing cores. (See also section 6.1).

STMBench7 Benchmark [41, 42] benchmark.[4] The STMBench7 is a benchmark for software transactional memory implementations; its underlying data structure consists of a set of graphs and indexes similar to many complex applications, e.g., CAD/CAM. Unlike most applications, this benchmark can already be executed transactionally, so this benchmark allows me to test only the speculative parallelization part of JaSPEx, and compare its execution times to those of a normal single-threaded run of the benchmark. Figure 6.3 shows results with a read-dominated workload, while Figure 6.4 shows with a write-dominated workload.

Finally, I present the results of executing a modified version of the recursive implementation of the Fibonacci function originally shown in Figures 5.6 and 5.7. Because `fib` does very little computation at each step, I have modified it to do speculative execution only up to a threshold, and from then on to run the rest of the computation entirely without speculative execution on the same thread, as shown in Figure 6.5. Note that the resulting application is still a normal, valid single-threaded Java application, that can be executed without speculative parallelization or JaSPEx.

Figure 6.6 presents the time needed for calculating `fib(50)` using this version with 1 to 8 cores. The single-core execution time shown is for a normal execution of `fib`, without the JaSPEx framework.

The results of this benchmark are interesting, because they show that with better scheduling decisions, it is possible to successfully extract parallelism from a sequential program with the approach that I propose; the threshold was only introduced to avoid the task overcreation that the framework scheduler originally did.

[4]See also `http://lpd.epfl.ch/site/research/tmeval`.

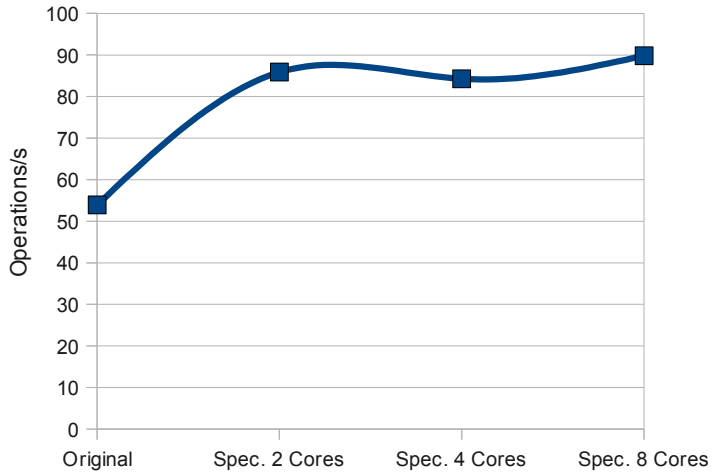

Figure 6.3: Original and speculative number of operations per second for the STMBench7 Benchmark, with a read-dominated workload. Shown are the original operations/second for the benchmark running in single-threaded mode, and the operations/second for speculative parallel execution with an increasing number of processing cores.

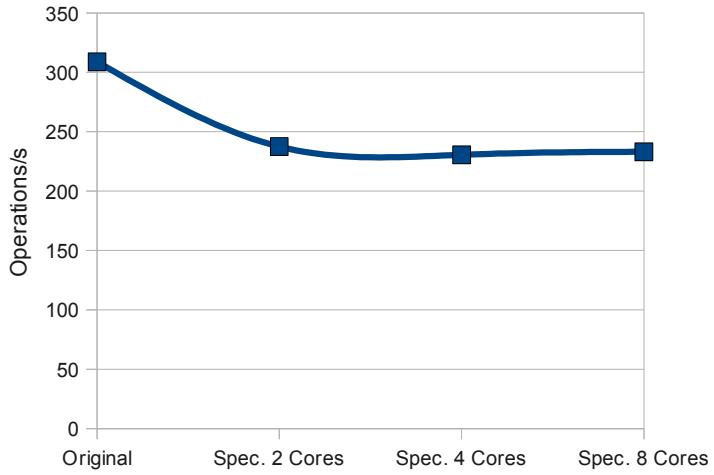

Figure 6.4: Original and speculative number of operations per second for the STMBench7 Benchmark, with a write-dominated workload. Shown are the original operations/second for the benchmark running in single-threaded mode, and the operations/second for speculative parallel execution with an increasing number of processing cores.

```
public long fib(long n) {
    if (n < threshold) return fib_nospeculation(n);
    return fib(n-1) + fib(n-2);
}

// This version of fib will not spawn speculative executions
public long fib_nospeculation(long n) {
    if (n <= 1) return n;
    return fib_nospeculation(n-1) + fib_nospeculation(n-2);
}
```

Figure 6.5: Modified version of `fib`. When the value of `n` that `fib` receives goes below the threshold, execution jumps to the `fib_nospeculation` method, for which no speculative execution is done.

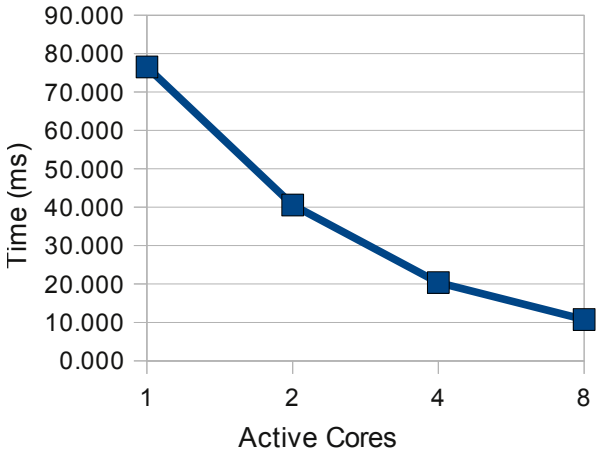

Figure 6.6: Time for calculating `fib(50)` using speculative parallelization, as the number of available cores is increased. The single-core execution time shown is for a normal execution of `fib`, without the JaSPEx framework.

Chapter 7

Conclusion

In this dissertation, I proposed the use of an STM-based approach to thread-level speculation, so that we may extract more parallelism from sequential programs, benefit from the results of the transactional memory research community, and target current mainstream hardware.

I have incorporated my proposal into a running system, the Java Speculative Parallel Executor (JaSPEx), that automatically parallelizes a program that was compiled to run in the Java Virtual Machine. To accomplish that, JaSPEx transforms the program, without the intervention of the programmer, so that some parts of it may execute speculatively in parallel, while still respecting the original sequential execution semantics.

JaSPEx was built and designed both as a framework on which future speculative execution research can build upon, and as a working proof-of-concept implementation of such a system.

One of the challenges in speculative execution is the transactification of the program. In this work I described some of the difficulties inherent to the transactification of a JVM program if there is no support from the JVM runtime. Because of those difficulties, the transactification performed by JaSPEx is currently limited.

I also described how JaSPEx modifies applications for speculative execution, how tasks are selected and when speculation is introduced, how speculative executions are managed internally by the framework, and what is the API used by a parallelized application.

In its current state, JaSPEx provides a working framework for speculative execution, including transactification, but its current speculative execution techniques have proved effective only for a small number of applications.

7.1 Main Contributions

The main goal for this work was the creation of a speculative parallelization system for existing sequential applications that run on the Java Virtual Machine platform, without needing to modify the Virtual Machine. The parallelization system should also work automatically without any input from the application programmer.

A secondary goal was that the resulting system should also be a usable platform for future speculative execution research.

The main contributions done by this work are the following:

- *Transactification system for the JVM platform.* I have analyzed the issues underlying the transactification of an application — modifying it so that it can run transactionally under control of an STM — on the JVM platform and developed techniques to solve some the presented problems. I then implemented those techniques as part of the JaSPEx system.
- *Identification of dependencies on non-transactional methods.* I have investigated and described two approaches to handling identification of non-transactional methods, and shown that static identification is too conservative for real-world applications, causing most methods to be identified as possibly dependent on native methods, and that a runtime approach can lead to more parallelism being uncovered.
- *Speculative execution on an unmodified JVM platform.* To the best of my knowledge, I have developed the first fully-automatic speculative parallelization system that works on top of an unmodified JVM platform, and that does not rely on any kind of transactional support provided by the hardware to work.
- *Speculative parallelization framework.* I have developed a speculative parallelization framework that may be used as the basis for future speculative execution research and development.
- *Benchmarking results.* I have performed preliminary benchmarking that show promising results for the approach I proposed.

7.2 Future Research Directions

As I have shown in the previous sections, the JaSPEx system is hindered by the many limitations of the Java Virtual Machine platform, and incurs into high overheads from the usage of a Software Transactional Memory.

Starting with the latter, by integrating the Software Transactional Memory with the Java Virtual Machine, less overheads would be incurred, and the STM could better collaborate with other sections of the VM, especially garbage collection, to improve performance. The usage of STM could also be made transparent for the user application by this integration, avoiding some of the work that currently has to be done on transactification, and some of its limitations, such as array handling and reflection.

Transactification at the Virtual Machine level could also be expanded to allow transactification of the Java base libraries, which would greatly reduce the set of methods that are considered non-transactional.

Integrating the JaSPEx system itself at the Virtual Machine level would further reduce overheads and improve performance, because the VM has full control of what happens at all times, so speculation could be done at a finer-grain level, without all of the overheads that are imposed by the hooks that are currently needed to give JaSPEx a moderate level of control over code executed. This kind of integration could also allow worker threads to be reused while they are waiting for the *commit token*, instead of the current arrangement where the framework has to create new threads to be able to switch between different execution contexts.

Modifying the JVSTM model to support concurrency in nested transactions would also serve to reduce the number of failed speculative executions by allowing parent speculations to share some of their preliminary results with their child speculations.

Besides these issues, investigation and implementation of better parallelization techniques would also benefit JaSPEx:

- *Task scheduling.* How to better map the different speculative tasks to the available processing resources of a system.
- *Task selection.* How to divide an application into parts that are as independent as possible.
- *Statistics gathering and profiling.* Keep track of past speculative execution success rates and speedup gains, and try to avoid possible but unprofitable executions.
- *Value prediction.* Employ value prediction techniques such as those proposed by [38, 14] whenever results that are not yet available are needed to proceed with speculative execution, allowing JaSPEx to uncover further parallelism in applications.
- *Loop optimizations.* Implement loop parallelization, unrolling, and other transformations. Techniques for loop optimization have been the focus for most parallelizing compilers, with some successful results, and it would be interesting to investigate if JaSPEx could mirror and improve their success.
- *Better static analysis.* Add support for more advanced static analysis and optimization that can generate better code and richer metadata to be used for speculation. As the usage of a dynamic approach does not preclude the usage of a static one, JaSPEx could benefit from the existing research on static compiler optimizations.

Bibliography

[1] Blume, B., Eigenmann, R., Faigin, K., Grout, J., Hoeflinger, J., Padua, D., Petersen, P., Pottenger, B., Rauchwerger, L., Tu, P., et al.: Polaris: The Next Generation in Parallelizing Compilers. In: Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing. (1994)

[2] Wilson, R.P., French, R.S., Wilson, C.S., Amarasinghe, S.P., Anderson, J.M., Tjiang, S.W.K., Liao, S.W., Tseng, C.W., Hall, M.W., Lam, M.S., Hennessy, J.L.: Suif: an infrastructure for research on parallelizing and optimizing compilers. SIGPLAN Not. **29**(12) (1994) 31–37

[3] Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., Torrellas, J.: POSH: a TLS compiler that exploits program structure. In: PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, New York, NY, USA, ACM (2006) 158–167

[4] Yang, X., Zheng, Q., Chen, G., Yao, Z.: Reverse compilation for speculative parallel threading. Parallel and Distributed Computing Applications and Technologies, International Conference on **0** (2006) 138–143

[5] Chen, M., Olukotun, K.: The Jrpm system for dynamically parallelizing Java programs. In: Proceedings of the 30th annual international symposium on Computer architecture, ACM New York, NY, USA (2003) 434–446

[6] Shavit, N., Touitou, D.: Software transactional memory. Distributed Computing **10**(2) (1997) 99–116

[7] Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.: Software transactional memory for dynamic-sized data structures. In: Proceedings of the twenty-second annual symposium on Principles of distributed computing, ACM Press New York, NY, USA (2003) 92–101

[8] Bracha, G.: Generics in the Java programming language. Sun Microsystems, java. sun. com (2004)

[9] Herlihy, M., Moss, J.: Transactional memory: architectural support for lock-free data structures. In: Proceedings of the 20th annual international symposium on Computer architecture, ACM New York, NY, USA (1993) 289–300

[10] Saha, B., Adl-Tabatabai, A., Hudson, R., Minh, C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM New York, NY, USA (2006) 187–197

[11] Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 2006 ASPLOS Conference. Volume 41., ACM New York, NY, USA (2006) 336–346

[12] Rajwar, R., Herlihy, M., Lai, K.: Virtualizing transactional memory. In: Computer Architecture, 2005. ISCA'05. Proceedings. 32nd International Symposium on. (2005) 494–505

[13] Bridges, M., Vachharajani, N., Zhang, Y., Jablin, T., August, D.: Revisiting the sequential programming model for multi-core. In: Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Computer Society Washington, DC, USA (2007) 69–84

[14] Oplinger, J., Heine, D., Lam, M.: In search of speculative thread-level parallelism. In: Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on. (1999) 303–313

[15] Allen, E., Chase, D., Luchangco, V., Maessen, J., Ryu, S., Steele Jr, G., Tobin-Hochstadt, S., Dias, J., Eastlund, C., Flood, C., et al.: The Fortress language specification. Sun Microsystems **139** (2005) 140

[16] Steele, G.: Parallel programming and parallel abstractions in fortress. Lecture Notes in Computer Science **3945** (2006) 1

[17] Charles, P., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C., Sarkar, V.: X10: an object-oriented approach to non-uniform cluster computing. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming systems languages and applications. Volume 40., ACM New York, NY, USA (2005) 519–538

[18] Lea, D.: A Java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande, ACM New York, NY, USA (2000) 36–43

[19] Blumofe, R., Leiserson, C.: Scheduling multithreaded computations by work stealing. Journal of the ACM (JACM) **46**(5) (1999) 720–748

[20] Danaher, J., Lee, I., Leiserson, C.: The JCilk language for multithreaded computing. In: OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL). (2005)

[21] Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. In: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications, ACM New York, NY, USA (2005) 439–453

[22] Alpern, B., Attanasio, C., Barton, J., Burke, M., Cheng, P., Choi, J., Cocchi, A., Fink, S., Grove, D., Hind, M., et al.: The Jalapeño virtual machine. IBM Journal of Research and Development **39**(1) (2000) 211

[23] Carlstrom, B., Chung, J., Chafi, H., McDonald, A., Minh, C., Hammond, L., Kozyrakis, C., Olukotun, K.: Transactional Execution of Java Programs. In: OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL). (2005)

[24] Carlstrom, B., McDonald, A., Chafi, H., Chung, J., Minh, C., Kozyrakis, C., Olukotun, K.: The Atomos transactional programming language. In: Proceedings of the 2006 PLDI Conference. Volume 41., ACM New York, NY, USA (2006) 1–13

[25] McDonald, A., Chung, J., Chafi, H., Minh, C., Carlstrom, B., Hammond, L., Kozyrakis, C., Olukotun, K.: Characterization of TCC on Chip-Multiprocessors. In: Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on. (2005) 63–74

[26] Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming, ACM New York, NY, USA (2005) 48–60

[27] Moss, J.E.B., Hosking, A.L.: Nested transactional memory: model and architecture sketches. Sci. Comput. Program. **63**(2) (2006) 186–201

[28] Cachopo, J.: Development of Rich Domain Models with Atomic Actions. PhD thesis, Technical University of Lisbon (September 2007)

[29] Cachopo, J.: JVSTM - Java Versioned Software Transactional Memory (2008) `http://web.ist.utl.pt/~joao.cachopo/jvstm/`.

[30] Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Science of Computer Programming **63**(2) (2006) 172–185

[31] Arnold, K., Gosling, J.: The Java programming language. (1998)

[32] Lindholm, T., Yellin, F.: Java Virtual Machine Specification. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA (1999)

[33] Goetz, B.: Optimistic thread concurrency: Breaking the scale barrier. Technical report, Technical Report AWP-011-010, Azul Systems, Inc., Mountain View, CA, USA, Jan. 2006

[34] Manson, J., Pugh, W., Adve, S.: The Java Memory Model

[35] Mukhi, S., Rottstädt, N.: Survey Open-Source JVMs

[36] Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems (2002)

[37] Liang, S.: The Java native interface: programmer's guide and specification. Addison-Wesley Professional (1999)

[38] Lipasti, M., Wilkerson, C., Shen, J.: Value locality and load value prediction. ACM SIGOPS Operating Systems Review **30**(5) (1996) 138–147

[39] Mathew, J., Coddington, P., Hawick, K.: Analysis and development of Java Grande benchmarks. In: Proceedings of the ACM 1999 conference on Java Grande, ACM New York, NY, USA (1999) 72–80

[40] Bull, J., Smith, L., Westhead, M., Henty, D., Davey, R.: A benchmark suite for high performance Java. Concurrency Practice and Experience **12**(6) (2000) 375–388

[41] Guerraoui, R., Kapalka, M., Vitek, J.: STMBench7: a benchmark for software transactional memory. ACM SIGOPS Operating Systems Review **41**(3) (2007) 315–324

[42] Dragojević, A., Guerraoui, R., Kapałka, M.: Dividing transactional memories by zero. In: Proc. of the 3rd ACM SIGPLAN Workshop on Transactional Computing, Salt Lake City, UT. (2008)