A Software-based Method-Level Speculation Framework for the Java Platform[⋆]

Ivo Anjo and João Cachopo

ESW
INESC-ID Lisboa/Instituto Superior Técnico/Universidade Técnica de Lisboa
Rua Alves Redol 9, 1000-029 Lisboa, Portugal
{ivo.anjo,joao.cachopo}@ist.utl.pt

Abstract. With multicore processors becoming ubiquitous on computing devices, the need for both parallelizing existing sequential applications and designing new parallel applications is greatly intensified. With our work, we intend to tackle the former issue.
In this paper, we present the design of a software-based automatic parallelization framework for sequential applications that run on the Java platform: the JaSPEx-MLS framework.
Our framework employs Method-Level Speculation: It uses method invocations as fork points and converts those invocations to return futures that can be stored in local variables in place of the original values. The support for speculative execution is provided by automatically modifying application bytecode to use a custom lightweight Software Transactional Memory (STM), and we present a novel approach to integrate futures representing speculative executions with the STM. Thread state transfer is done by employing a Java Virtual Machine that provides support for first-class continuations.
We present preliminary results from our implementation of the proposed techniques on the JaSPEx-MLS framework, which works on top of the OpenJDK Hotspot VM.

Keywords: Automatic Parallelization, Method Level-Speculation, Software Transactional Memory, Continuations, OpenJDK Hotspot JVM

1 Introduction

With the move to multicore processors, many new applications are being developed with concurrent architectures in mind. Yet, many existing applications are still sequential, and fail to take advantage of the full computing potential promised by the multicore age.

Unfortunately, it is not feasible for a vast majority of sequential applications to be rewritten to work in parallel within a reasonable time frame. Thus, an enticing option is to use an automatic approach to parallelization.

[⋆] This work was supported by FCT (INESC-ID multiannual funding) through the PIDDAC Program funds and by the RuLAM project (PTDC/EIA-EIA/108240/2008).

Parallelizing compilers [4,19] are one such approach that attempts to extract concurrency from sequential programs automatically by proving that parts of an application can be safely executed in parallel. The problem is that they fail to parallelize many irregular applications [8,9,12] that employ dynamic data structures, loops with complex dependences and control flows, and polymorphism, which are very hard or even impossible to analyze in a fully static way.

Thread-level speculation (TLS) systems [5,10,11,13,15,21] attempt to work around this issue by optimistically running parts of the application in parallel, even if the TLS system is not able to prove statically that there will be no dependences. Instead, correctness is dynamically ensured at runtime, during or after execution of the parallel tasks. Incorrect operations and memory changes are prevented by buffering and tracking the execution of such operations, followed by validation before they are propagated to the global program state.

There are multiple ways of identifying tasks from a sequential application to be executed in parallel. Most TLS proposals concentrate only on loops [5,10,11,15], whereas on our system we chose to use method calls as spawning points, as proposed by [6,12,13,18].

In this paper, we present the design of our method-based speculation system, which was implemented on top of the JaSPEx [1,2] speculative parallelization framework. Our system needs no special hardware extensions, instead relying on Software Transactional Memory (STM) for transactional support, and it works on top of a modified version of the OpenJDK Hotspot Java Virtual Machine (JVM), allowing it to benefit from a state-of-the-art, production-level managed runtime with dynamic optimization, garbage collection, and support for Java 6.

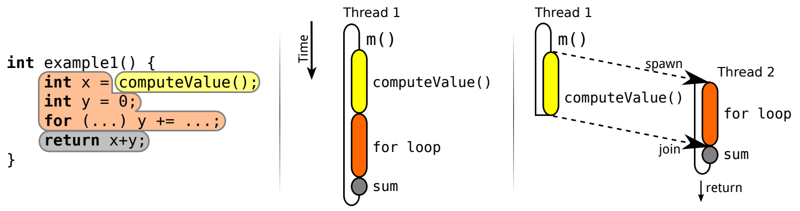
The rest of this paper is organized as follows. Section 2 introduces the Method-Level Speculation technique, and Section 3 introduces the JaSPEx-MLS parallelization framework. The static modifications done by the JaSPEx-MLS classloader are described in Section 4, whereas in Section 5 we detail the runtime creation and coordination of speculative tasks. Section 6 presents preliminary experimental results for our system. Section 7 discusses the related work, and we finish in Section 8 by presenting conclusions and future research directions.

2 Method-Level Speculation

Method-level speculation (MLS) is a speculative parallelization strategy first discussed in the context of Java by [6], and shown to be a promising source for parallelism by [12,13,18]. This technique works by speculatively executing the code that follows the return of a method call — its continuation — in parallel with the method call itself.

An example of method-level speculation is shown in Figure 1. When the `computeValue()` method call is reached, the current thread (`T1`) begins executing it, while at the same time triggering the speculative execution (by `T2`) of the code following the return of that method.

In this example, both the original parent thread and the speculative child thread have to join to produce the result of the method. If the value of the



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

Fig. 1. Execution of `example1()` method when run normally (center) and parallelized with MLS (right).

variable `x` was never used, it would be possible to speculate past the return of `example1()`, and continue execution of the method that invoked it.

Alternatively, even if `x`'s value is needed to proceed with the execution, we can employ return value prediction [7, 14] to guess a probable value of `x`, and continue speculation using this assumption.

3 The JaSPEx-MLS Parallelization Framework

JaSPEx-MLS is a fully software-based speculative parallelization framework that provides both a Java classloader that modifies application code as it is requested by the virtual machine, and a runtime Java library that orchestrates speculative execution. It is based on the JaSPEx framework [1,2], but with both an entirely new MLS-based speculation model, and a new transactional backend.

The JaSPEx-MLS classloader (Section 4) is responsible for preparing application code for speculative parallelization. This includes transactifying the code, adding hooks to allow the framework to correctly handle non-transactional operations, and inserting into the application the spawn points that will be used at runtime to create speculative tasks.

The runtime orchestration library (Section 5) is responsible for controlling the creation of speculative tasks, establishing the commit order for the underlying transactional system, deciding when to validate and to commit speculative tasks, correctly handling aborting and retrying, and controlling the execution of non-transactional operations. Speculative work is submitted to a thread pool, which we attempt to keep busy at all times. Nested speculation is supported.

Almost all of JaSPEx-MLS is implemented in Java, and modifications to applications are done via bytecode rewriting.[1] The lone exception to this is that JaSPEx-MLS relies on having first-class continuation support, which is provided by a modified version of the OpenJDK virtual machine.

The OpenJDK VM is the result of the open-sourcing of Oracle's Java technology, including the Hotspot JVM. By working on top of OpenJDK, JaSPEx-MLS has access to all the features and optimizations of a modern production JVM: just-in-time compilation and adaptive optimization, state-of-the-art garbage collection algorithms, support for Java 6 and optimized concurrency primitives.

We believe that the combination of software-only speculation on top of a modern production JVM sets our system apart from previous work: Our ap-

[1] To simplify presentation, the examples in this paper instead appear in Java.

proach can work on commonly available modern hardware, and on top of the same codebase regularly used to run the sequential versions of the applications that we are targeting.

Our first-class continuation implementation is based on previous work by Yamauchi [20], which itself was based on the work of Stadler et al. [16]. We have developed a library that currently includes backends for two different JVM implementations of first-class continuations, and that will allow JaSPEx-MLS to easily adapt to future developments in this area.

4 JaSPEx-MLS Classloader: Static Code Preparation

As introduced in Section 3, the JaSPEx-MLS classloader handles the static preparation of classes for speculative parallelism. An important assumption that we make is that any class that is prepared and loaded by this classloader is fully safe to invoke with transactional semantics. The modifications described in the following subsections allow a class to fulfill this assumption.

4.1 Transactification

The first part of static application processing is concerned with allowing application code to run with transactional semantics. This allows JaSPEx-MLS to control memory read and write operations during speculative execution, to have a means of validating them, and to decide if they should be kept or not.

Rather than modifying the virtual machine to obtain this transactional support, we intercept any Java bytecodes that may access and mutate heap-allocated memory locations—that is, accesses to object slots and to array elements.

As such, an application is modified to use an STM-like API whenever it must read or write to slots and arrays.[2] This API is very lightweight, type-specific, and static, allowing the JVM to easily inline it into hot paths of the code.

4.2 Handling Non-Transactional Operations

In any transactional system, there are always some operations that cannot be made to behave transactionally, as they are outside the control of the system.

For JaSPEx-MLS, and in the JVM platform, we consider as non-transactional two types of operations: (1) `native` methods, which are implemented with precompiled binary code, making them hard to analyze and transactify; and (2) code belonging to the JDK (any classes in the `java.*` package namespace).

Code belonging to the JDK is considered to be non-transactional because the OpenJDK JVM, like Oracle's JVM, does not allow alternative versions of JDK classes to be loaded at runtime. To reduce the number of non-transactional operations resulting from this limitation, we use a semi-manually–compiled whitelist that includes immutable classes and methods that do not change any state, and

[2] An exception is the access to `final` fields, which do not change after initialization.

```
java.util.List l = ...;
if (!(l instanceof Transactional)) nonTransactionalActionAttempted();
l.clear();
```

Listing 1.1. Runtime check for `Transactional` instances.

that do not access state from non-transactional classes nor arrays. Yet, in the future, we intend to explore either the feasibility of modifying the VM to remove this restriction, or a more limited offline modification of these base classes before they are loaded.

To protect an application from executing a non-transactional operation while performing speculative execution, we prepend any such operation with a call to the framework method `nonTransactionalActionAttempted()`, which validates the current speculation, waiting if needed, before allowing the operation to proceed, or aborts the execution if the speculation is not valid.

In addition, as it is not always possible to distinguish statically when, for instance, a reference `l` of type `java.util.List` refers to a user-provided `MyList` or a non-transactional `java.util.ArrayList`, a runtime test is added. This runtime test relies on the fact that any class processed by our classloader implements the `Transactional` interface, and thus, at runtime, we can avoid stopping speculation unless really needed, as shown in Listing 1.1.

4.3 Modifications for MLS

Whereas the previous modification steps of the JaSPEx-MLS classloader prepared application code to run with transactional semantics, the final step readies the code for MLS.

To add support for MLS, JaSPEx-MLS replaces normal method calls with a call to a special `spawnSpeculation()` method. This method receives a `Callable` object, representing the original method invocation and its arguments, and returns a `Future`, representing the value that will be returned by the target method.

The `Callable` object is an instance of an automatically generated class that includes slots for each argument to the method call. When the `call()` method is invoked, it proceeds to call the original method.

The most complex part of the insertion of `spawnSpeculation()` is dealing with the returned future. The main objective of the transformation performed is to delay to as late as possible the retrieval of the result from the future, as it would entail waiting if the value is not yet computed. Thus, the trivial case where the future is immediately needed is not useful,[3] as nothing would be gained from just transferring execution to another thread, and so JaSPEx-MLS rejects this case. The other trivial case, where the value returned from the method is discarded, or the method is `void`, is useful, but needs no further modifications other that popping the future off the stack.

A more interesting case however, is the common pattern of saving the result of a method on a local variable for later use. The JVM bytecode specification

[3] JaSPEx-MLS currently does not employ return value prediction [7, 14].

```
void example() {
  int x = 0;
  if (condition) {
    Future f0 = spawnSpeculation(...); // original method: compute()
    x = f0;
  }
  int y = x + 1; // error: is x an int or a future ???
  { ... code that uses y ... }
}
```

Listing 1.2. Example of problematic replacement of a returned value with a future.

```
void example() {
  int x = 0;
  if (condition) {
    Future f0 = spawnSpeculation(...); // original method: compute()
    x = f0;
    goto x_is_a_Future;
  }
  int y = x + 1; // x is an int
rest_of_the_method:
  { ... code that uses y ... }
  return;
x_is_a_Future:
  int y = x.get() + 1; // x is a future
  goto rest_of_the_method;
}
```

Listing 1.3. Valid version of the code from Listing 1.2, obtained by duplicating part of the method.

allows any type to be stored in any local variable (and this type can change during execution of a method), so we are allowed to write the future to the same local variable as the original return value would have.

The problem with this substitution is what happens when the return value is accessed. Consider for instance the code shown in Listing 1.2: This transformation is not valid, because the `x` local variable may be of type `int` in a possible path through the method, and of type `Future` on another path.

To solve this problem, we construct the control flow graph of the method and duplicate code blocks where both a future and the original return type may be present. As an example, Listing 1.3 shows the correct version of the transformation shown in Listing 1.2.

To avoid spawning speculative executions that would run only a small number of instructions before needing to synchronize with other threads, JaSPEx-MLS does a number of passes that perform simple analysis to try to avoid these cases. In addition, the MLS modification pass can use a list of methods that are known not to be profitable for speculation: This list may either be manually provided, or be the result of profiling performed on the application.

4.4 STM Support for Futures

To further delay the moment when we need to obtain the return value from the future, we added to our STM support for writing futures to memory locations.

```
// Original Method
void doCompute(Object[] results) {
  for (int i = 0; i < results.length; i++) {
    results[i] = compute(i);
  }}

// Attempted parallelization
void doCompute(Object[] results) {
  for (int i = 0; i < results.length; i++) {
    Future f0 = spawnSpeculation(...); // original method: compute(i)
    TM.storeObjectArray(results, i, f0.get()); // get() called immediately
  }}
```

Listing 1.4. Unsuccessful parallelization of `doCompute()`.

```
void doCompute(Object[] results) {
  for (int i = 0; i < results.length; i++) {
    Future f0 = spawnSpeculation(...); // original method: compute(i)
    TM.storeFutureObjectArray(results, i, f0); // f0 is handed to the STM
  }}
```

Listing 1.5. Successful parallelization of `doCompute()`, with the added support for futures in the STM.

Consider, for instance, Listing 1.4: In this case, the transformation performed in Section 4.3 to add the `spawnSpeculation()` call would not be useful, as the resulting code would immediately obtain the return value from the future, so that it can be written into the array.

In reality, due to the transactification step performed in Section 4.1, the write to the array is not done directly, but instead the value to be written is handed over to the STM. We can take advantage of this behavior to extend the STM with support for futures, allowing the resulting code to behave as shown in Listing 1.5.

Note that if each execution of `compute()` that is being replaced by `spawn-Speculation()` is fully independent, the example method (and the loop contained therein) has gone from not being parallelizable, to being fully parallel, as the entire loop can be executed without stopping speculation, and the `doCompute()` method can even return to its caller, allowing other work to be done, while the computation of the values proceeds in parallel.

5 Runtime Orchestration of Speculative Executions

The JaSPEx-MLS runtime library is responsible for the creation and coordination of speculative executions.

A speculative execution starts when an application reaches a call to the `spawnSpeculation()` method, which was previously inserted by the JaSPEx-MLS classloader as a replacement for a normal method call. Inside this method, the framework dynamically decides weather a new speculation should be spawned by taking into account the current workload of the system. Because in our system nested speculations are supported, speculative executions can spawn further speculative executions.

If JaSPEx-MLS chooses to spawn a speculation, it starts by capturing a first-class continuation representing the stack and execution state of the current thread. Remember from Figure 1 in Section 2 that this execution state will be resumed on another thread, while the current thread will continue by executing the method call contained in the `Callable` received by `spawnSpeculation()`. To represent the task being spawned, JaSPEx-MLS creates a new instance of `SpeculationTask` and submits it for execution by the thread pool.

The created `SpeculationTask` instance is the link between the *parent* task — the task that reached the call to `spawnSpeculation()` — and the *child* task, which will resume the continuation and start its execution of the code following the call to `spawnSpeculation()`. This parent/child relation implicitly imposes a global order on all tasks on the system that mirrors the original order on the sequential application.

After submitting the child `SpeculationTask` for execution, the parent task cleans its current thread's stack by resuming an empty continuation, and proceeds to execute the method represented by the `Callable`. When the method returns, its return value is stored inside the child `SpeculationTask`, so that the child task will be able to retrieve it, and the thread running the parent task returns to the thread pool.

When the child `SpeculationTask` is picked up for execution by a thread, we first test if the task's parent already finished by checking if its result is available. If it is not, a new STM transaction is started, otherwise, because its parent already committed, no transaction is started and the task is executed in program-order mode. The thread then resumes the first-class continuation captured by the parent task: Upon resuming the continuation, execution restarts inside the `spawnSpeculation()` method, and JaSPEx-MLS returns a future to the caller method — representing the promise of a return value from the parent task — and the child task continues its execution.

This design where the thread that reaches the `spawnSpeculation()` throws away its stack and executes the parent task, while the child task will start by restoring the very same stack was chosen so as to allow tasks to be queued even when there are no free threads to execute them. The inverse option, where the thread that reaches the `spawnSpeculation()` would execute the child task and queue the parent for execution could in many cases delay the application, as the child task would not be able to commit its work before its parent was finished.

5.1 Committing a Speculation

There are three conditions that trigger the commit of a speculative task:

1. The task completed its work
2. The task needed to obtain a result from a future, and noticed that it was the oldest-running task in the system
3. The task attempted to execute a non-transactional operation

A speculative task is allowed to commit its work only if it is the oldest-running task in the system. In our design, every task has a parent that spawned it, and that parent is responsible for writing its result onto the child's `SpeculationTask`.

Every parent commits before its child task and before passing its result to the child. Thus, when a child receives the result from its parent, it can also commit because it is guaranteed that its parent has finished its work.

When a task wants to commit, but no result from its parent is available, it waits on its own `SpeculationTask` for this value to arrive. Note that it may be possible that its parent is in the same situation, and that a sequence of speculative executions are all waiting for their own parents. When a parent sets its value on the child, it wakes up the child, so that the child can resume working.

After a child task receives the result from its parent, we first check for two special cases: an exception and an order to abort. If the result from the parent is an exception, then whatever work the current speculative task has done is invalid: In the original application, this code would never run, because the exception would be thrown before the code was reached. So, when this happens, the child task aborts its current STM transaction, signals its own child speculation (if any) that it should abort, and retries execution by re-resuming the continuation (which re-initializes the current execution stack back inside the `spawnSpeculation()` method) and by re-throwing the exception thrown by the parent. This scheme simulates the way the exception would appear in the original application. In case the speculation receives an order to abort, either because its parent (or any grandparent) aborted due to an exception or because a mis-speculation was detected by the STM, the current task aborts its own transaction, signals its child (if any) to abort also, and the thread is returned to the thread pool. Any computation done was wasted, because it was based on invalid assumptions.

When a thread attempts to commit, but the validation of the STM transaction fails, the transaction is aborted, and the task is retried by re-resuming the continuation received from the parent. For the re-execution, no transaction is started, as the task will be running in program-order, rather than speculatively.

Finally, whenever a task finishes its work and is able to commit successfully its STM transaction, it writes its return value on its child `SpeculationTask` and the thread hosting it is returned to the thread pool.

5.2 Custom Relaxed STM Model

The STM used in JaSPEx-MLS was designed to be very lightweight, so as to impose minimal overheads on the transactified application. It clearly distinguishes between two modes of execution: program-order mode and speculation mode.

A task that is executing in program-order mode always reads from and writes to memory directly, with no additional validation nor synchronization: At any given time, only one task is working in program-order; any other threads executing tasks are performing speculative execution.

When the program-order task attempts to write a future into a memory location, the return value from the future is immediately retrieved as, per the structure imposed by the method-level speculation scheme, that future represents the result of a previous speculation that must already have finished.

Like most STMs, for tasks running in speculative mode, JaSPEx-MLS keeps both a (value-based) read-set, which contains each heap-allocated memory location read by the transaction and its value at the time, and a write-set, which maps memory locations to values to be written to them upon commit.

In our STM model, tasks running speculatively always read directly from the requested memory location, and may observe changes being done concurrently by a task running in program-order mode. This strategy can, of course, cause inconsistent reads, but unlike normal applications that use STM, inconsistent reads are always present in the execution model, and are handled by the framework.

When a read of a memory location is attempted, and there is already an entry in the write-set for that location, the value from the write-set is returned. If the entry in the write-set contained a future in place of the real value, the result from the future is first retrieved (by waiting, if necessary), and then returned.

Whenever a speculative task wants to perform a write, a new mapping is added/updated to the write-set: a pair $(location, newvalue)$ for normal values, and a pair $(location, future)$ for futures.

Because the task coordination part of JaSPEx-MLS always commits transactions in the same order as the original sequential application, only one transaction will be trying to commit at any one time, dismissing the need for synchronization during the commit operation. As such, the commit operation consists of only two simple steps: (1) validating the the read-set by re-reading the values from the memory locations and comparing them to the ones originally read and kept in the read-set; and (2) performing the write-back of values from the write-set to the memory locations, including retrieving and writing the results from any futures.

5.3 Thread and Task Management

When a new speculative task is created, JaSPEx-MLS submits it for execution to a thread pool. The current design of the thread pool allocates a limited number of threads based on the number of CPUs on the machine, and accepts speculations only when there are idle threads.

This design is very simple, and we intend to improve it in the future in two ways: (1) by returning threads to the pool instead of waiting; and (2) by integrating a task scheduler.

The idea of returning threads to the pool instead of waiting is applicable when there is a great imbalance between the size of speculative tasks that threads are working on. If, for instance, the oldest task in the system is executing a very long-running code section, and all the other threads have, in the meantime, finished their work, and are waiting for permission to commit, no further speculations are accepted, and the application would be executing sequentially. Instead, we plan to have waiting threads capture a first-class continuation with their current state, which would then be associated with their parent. Then, when the thread running the parent task finishes its work, instead of immediately returning to the thread pool it would switch to and finish execution of its child task. This

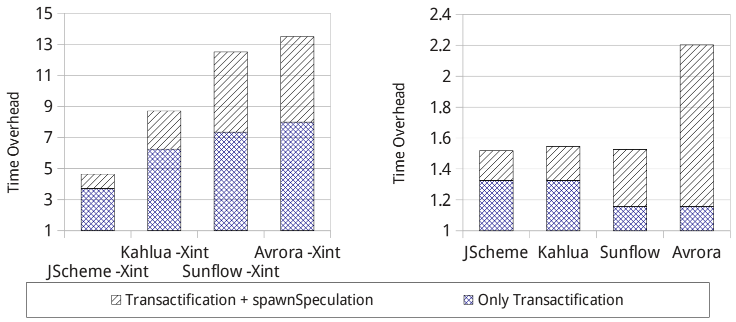


Fig. 2. Slowdown introduced by the bytecode modifications performed by JaSPEx-MLS (speculation was disabled) with the JVM in interpreter-only mode (left), and normal optimizing mode (right). *Only transactification* accounts for the modifications described in Sections 4.1-4.2, whereas *Transactification + spawnSpeculation* accounts for all bytecode modifications. Results were normalized to the runtimes of the unmodified applications.

way, waiting threads would be free to return to the thread pool, where they may accept new speculative tasks that are submitted by the very busy thread, speeding up its execution once again.

Baptista [3] was able to integrate a conflict-aware scheduler into an older version of the JaSPEx framework. His work shows promising results, and we intend to adapt it to JaSPEx-MLS in the future.

6 Experimental Results

In this Section, we present preliminary experimental results obtained with the JaSPEx-MLS framework. We tested our prototype on an Intel Core i5 750 machine with 8GB of RAM, running Ubuntu Linux 12.04 64-bit, and our modified OpenJDK VM.

We tested several JVM applications: the JScheme R4RS Scheme implementation, the Sunflow ray tracing engine, the Avrora hardware simulator and analysis framework, the Kahlua Lua scripting language interpreter, and some benchmarks from the the Java Grande Forum (JGF) benchmark suite. Apart from Sunflow, the chosen benchmarks are single-threaded, although some of them employed locking and thread-local variables in some places, which we removed; we modified Sunflow to use only the single main thread to perform its rendering work.

We first measured the overheads imposed by our system when speculation is disabled. Figure 2 shows the overhead we measured, when comparing our system to the original sequential application runtime, in two cases: (1) running the JVM in interpreter-only mode (`-Xint` mode), and (2) with the full Hotspot VM optimizations enabled. Using only the interpreter, our bytecode modifications impose heavy overheads, but when running with optimizations enabled, we can see that the VM is successfully able to optimize away many of the added indirections, showing that our lightweight STM imposes minimal overhead on application code running in program-order. The results also show that blind con-

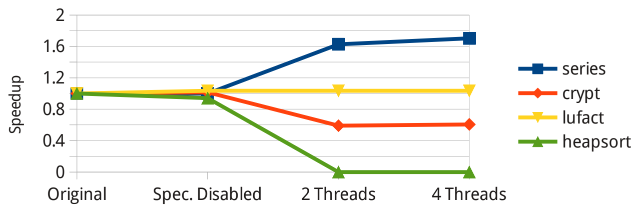


Fig. 3. Results from benchmarking with JGF benchmarks, normalized to the original application runtimes. Values above 1 correspond to a speedup, whereas below 1 correspond to a slowdown.

version of normal method calls into `spawnSpeculation()` also imposes non-trivial overheads, suggesting the importance of the integration of a profiling pass to remove unprofitable calls in our framework.

Results from testing with benchmarks from the JGF benchmark suite are shown in Figure 3. The input applications were not modified, but some manual profiling was done and a blacklist of methods unsuitable for speculation was provided to JaSPEx-MLS. The `series` benchmark was able to obtain a speedup of 1.7x, `lufact` was not able to perform any meaningful speculation, `crypt` was not able to extract speedup from the speculation done, `heapsort` had an overwhelming number of aborted transactions, and both `sor` and `sparsemult` (not shown) are not suitable for MLS parallelization as all their computation is done in a single method. These results again underline the need for a semi-automatic profiling pass that can avoid unprofitable executions, but also that it is possible to extract parallelism using JaSPEx-MLS.

7 Related Work

Many TLS proposals depend on some kind of hardware transactional support; unfortunately, while such support remains absent from common architectures these systems remain impractical. Jrpm [5] is a Java VM that does speculative loop parallelization on a multiprocessor with hardware support for profiling and speculative execution. At runtime, applications are profiled to find speculative buffer requirements and inter-thread dependences; once sufficient data is collected, the chosen loops are dynamically recompiled to run in parallel. Helper Transactions [21] rely on special hardware support to perform method-level speculation. The authors introduce the concept of implicit commit, allowing a thread that finished working to signal its sibling speculation to commit; this sibling speculation should then validate itself, commit its current work and continue executing non-speculatively. The advantage of this approach is that the oldest speculation in the system automatically switches off speculative execution as soon as possible, lowering execution overheads.

Because executing code transactionally can impose very large overheads, recent TLS proposals, similarly to JaSPEx-MLS, try to optimize the transactification and transactional model as much as possible: Oancea, Mycroft and Harris [11] proposed SpLIP, a software speculation system that targets mostly-parallel loops. The authors concentrated on avoiding performance pitfalls present on other software TLS proposals by having speculations commit their work in parallel, and using in-place updates. This contrasts with our approach of keeping changes in the write-set until commit, and increases the penalty for bad speculation decisions, which involve costly rollback operations, prompting a very careful analysis when choosing loops to parallelize. In [10] the authors propose STMlite, a lighter STM model targeting loop parallelization. STMlite aims at using a small number (2-8) of speculative threads to extract parallelism from loops, avoiding the need to transactify the whole program. During execution, transactional read and write operations are encoded using hash-based signatures that are then checked by a central bookkeeping and commit thread. Fastpath [15] is also aimed at extracting parallelism from loops using speculation. The transactional system distinguishes between the thread running in program order, and other speculative threads: The lead thread always commits its work, and has minimal overhead, whereas speculative threads suffer from higher overheads and may abort. The authors also propose two different STM-inspired algorithms for conflict detection: value and signature-based. In both conflict detection algorithms, the lead thread is always allowed to change memory locations in-place. The Fastpath system as presented did not yet support automatic parallelization; results from a hand-instrumented benchmark showed that the value-based algorithm presented the best results. The JaSPEx-MLS relaxed STM model is very similar to the Fastpath value-based algorithm, the biggest difference being our inclusion of support for futures in the STM.

The idea of using futures in Java coupled with speculative execution was also explored in a different context by Welc et al. [17]: In their work on safe futures for Java, the authors extend Java with support for futures that are guaranteed to respect serial execution semantics. Because of this, futures can be thought of as semantically transparent annotations on methods: Execution of a method can be replaced with the execution of a future, the safe future model guaranteeing that sequential semantics is respected. In contrast with our automatic approach, to use safe futures programmers manually change their code to employ futures instead of normal method calls, including solving cases where the return value from a method is used immediately. Zhang and Krintz's Safe DBLFutures [22] also support a similar approach, and includes safe handling of exceptions that respect sequential semantics.

SableSpMT [13] is a Java MLS-based automatic parallelization framework. To allow speculation even when the return value of a function call is needed immediately, SableSpMT employs return value prediction [14]. Nested speculation is not allowed, limiting some of the achievable parallelism: Although the main thread is allowed to spawn multiple speculative tasks, the tasks themselves cannot spawn further speculative tasks. Before running an application, SableSpMT performs a static analysis and modification pass on the input application: This pass inserts fork points into the application bytecode, and gathers information to be used by the return value predictor. SableSpMT is based on the SableVM virtual machine, which is a research VM, employing only an interpreter and a simpler garbage collection algorithm. The system was benchmarked using the SPECjvm98 benchmark suite on a quad-cpu machine, but no speedup was achieved over the original application runtimes due to the added overheads. In further testing with fork and join overheads factored out by considering a baseline execution where every speculation fails to commit at the end, SableSpMT was able to achieve a mean relative speedup of 1.34x. In contrast with SableSpMT, JaSPEx-MLS fully supports nested speculation, and in our system the garbage collector works normally, whereas in SableSpMT it invalidates all running speculations. The base SableVM is also a much simpler VM, with no support for Java 6 and none of the advantages of OpenJDK as introduced in Section 3.

8 Conclusions and Future Work

In this paper, we introduced the design of JaSPEx-MLS, an automatic parallelization framework for the Java platform. Our framework needs no hardware transactional support, and works atop a modern production-quality managed runtime supporting JIT compilation and advanced garbage collection facilities: the OpenJDK Hotspot virtual machine.

We described how the JaSPEx-MLS classloader transactifies applications, and also how it converts method calls into speculation spawn points. We also presented a novel approach to enable further speculation by integrating support for the futures returned at spawn points into the STM, allowing the application to behave as if the future itself was written to a memory location. We then described how speculative tasks are orchestrated at runtime, and the design of our lightweight relaxed STM model.

Our preliminary results show that an optimizing VM can hide much of the overhead introduced by our static bytecode preparation, and also that JaSPEx-MLS is already able to extract parallelism in some benchmarks.

In the future, we intend to lift some of the limitations imposed by our handling of non-transactional operations, to add support for a profiling pass that gathers information on the most profitable methods for speculation, and to improve the runtime scheduling of tasks and reuse of threads in the thread pool.

References

1. Anjo, I.: JaSPEx: Speculative Parallelization on the Java Platform. Master's thesis, Instituto Superior Técnico (2009)
2. Anjo, I., Cachopo, J.: JaSPEx: Speculative parallel execution of Java applications. In: Proceedings of the Simpósio de Informática (INFORUM 2009). Faculdade de Ciências da Universidade de Lisboa (2009)
3. Baptista, D.: Task Scheduling in Speculative Parallelization. Master's thesis, Instituto Superior Técnico (2011)
4. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T.: Parallel programming with Polaris. Computer 29(12), 78–82 (1996)
5. Chen, M., Olukotun, K.: The Jrpm system for dynamically parallelizing Java programs. ACM SIGARCH Computer Architecture News 31(2), 434–446 (2003)

6. Chen, M., Olukotun, K.: Exploiting method-level parallelism in single-threaded Java programs. In: 7th International Conference on Parallel Architectures and Compilation Techniques (PACT-1998). pp. 176–184. IEEE (1998)
7. Hu, S., Bhargava, R., John, L.: The role of return value prediction in exploiting speculative method-level parallelism. Journal of Instruction-Level Parallelism 5(1) (2003)
8. Kulkarni, M., Burtscher, M., Inkulu, R., Pingali, K., Cascaval, C.: How much parallelism is there in irregular applications? ACM SIGPLAN Notices 44(4), 3–14 (2009)
9. Lam, M., Wilson, R.: Limits of control flow on parallelism. ACM SIGARCH Computer Architecture News 20(2), 46–57 (1992)
10. Mehrara, M., Hao, J., Hsu, P., Mahlke, S.: Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory. ACM SIGPLAN Notices 44(6), 166–176 (2009)
11. Oancea, C., Mycroft, A., Harris, T.: A lightweight in-place implementation for software thread-level speculation. In: Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09). pp. 223–232. ACM Press (2009)
12. Oplinger, J., Heine, D., Lam, M.: In search of speculative thread-level parallelism. In: 8th International Conference on Parallel Architectures and Compilation Techniques (PACT-1999). pp. 303–313. IEEE (1999)
13. Pickett, C., Verbrugge, C.: Software thread level speculation for the Java language and virtual machine environment. In: Proceedings of the 18th International Conference on Languages and Compilers for Parallel Computing (LCPC 2005). pp. 304–318. Springer-Verlag (2006)
14. Pickett, C., Verbrugge, C.: Return value prediction in a Java virtual machine. In: Proceedings of the 2nd Value-Prediction and Value-Based Optimization Workshop (VPW2). pp. 40–47 (2004)
15. Spear, M., Kelsey, K., Bai, T., Dalessandro, L., Scott, M., Ding, C., Wu, P.: Fast-path speculative parallelization. In: Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing (LCPC 2009). pp. 338–352. Springer-Verlag (2010)
16. Stadler, L., Wimmer, C., Würthinger, T., Mössenböck, H., Rose, J.: Lazy continuations for Java virtual machines. In: 7th International Conference on Principles and Practice of Programming in Java (PPPJ 2009). pp. 143–152. ACM Press (2009)
17. Welc, A., Jagannathan, S., Hosking, A.: Safe futures for Java. ACM SIGPLAN Notices 40(10), 439–453 (2005)
18. Whaley, J., Kozyrakis, C.: Heuristics for profile-driven method-level speculative parallelization. In: Proceedings of the 2005 International Conference on Parallel Processing (ICPP '05). pp. 147–156. IEEE Computer Society (2005)
19. Wilson, R., French, R., Wilson, C., Amarasinghe, S., Anderson, J., Tjiang, S., Liao, S., Tseng, C., Hall, M., Lam, M., Hennessy, J.: SUIF: An infrastructure for research on parallelizing and optimizing compilers. ACM SIGPLAN Notices 29(12), 31–37 (1994)
20. Yamauchi, H.: Continuations in servers. In: JVM Language Summit 2010 (2010)
21. Yoo, R., Lee, H.: Helper transactions: Enabling thread-level speculation via a transactional memory system. In: 2008 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures (PESPMA 2008). pp. 63–71 (2008)
22. Zhang, L., Krintz, C.: As-if-serial exception handling semantics for Java futures. Science of Computer Programming 74(5-6), 314–332 (2009)