J. Parallel Distrib. Comput. 87 (2016) 13–25

Contents lists available at ScienceDirect

J. Parallel Distrib. Comput.

journal homepage: www.elsevier.com/locate/jpdc

# Design of a Method-Level Speculation framework for boosting irregular JVM applications

Ivo Anjo *, João Cachopo

*ESW/INESC-ID Lisboa/Instituto Superior Técnico/Universidade de Lisboa, Rua Alves Redol 9, 1000-029 Lisboa, Portugal*

## HIGHLIGHTS

- JaSPEx-MLS: an automatic parallelization framework for JVM applications.
- Uses Method-Level Speculation.
- Custom STM extended with support for futures, value prediction, and captured memory.
- Optimized execution with custom thread pool buffering and task freezing mechanisms.
- Works on top of the HotSpot JVM and is able to obtain speedups over the Oracle JVM.

## ARTICLE INFO

*Article history:*
Received 1 November 2014
Received in revised form
23 August 2015
Accepted 18 September 2015
Available online 30 September 2015

*Keywords:*
Automatic parallelization
Method-Level Speculation
First-class continuations
Software Transactional Memory
OpenJDK HotSpot JVM

## ABSTRACT

Despite the ubiquity of multicores, many commonly-used applications are still sequential. As a consequence, many chip designers are still investing on the creation of chips with a small number of ever-more-complex cores, showing that sequential performance is still a very important issue in some of today's computing systems. To tackle this issue, we have developed JaSPEx-MLS: a software-based automatic parallelization framework targeted at sequential irregular Java/JVM applications, that is based on Method-Level Speculation and Software Transactional Memory, and works atop the OpenJDK HotSpot JVM, a state-of-the-art managed runtime. We aim our framework as a software implementation of the boost feature in modern CPUs, allowing sequential applications to execute faster on multicores whenever parallel versions of those applications are not yet available. In this work, we describe the design of our framework, and introduce several techniques that when combined allow it to parallelize applications successfully with minimal overheads on commonly-available multicores.



## 1. Introduction

Despite multicore processors having reached near-ubiquity, most CPU manufacturers are still focused on improving their ever-more-complex cores: a decade after Intel and AMD introduced multicore computing to their desktop chips, most commonly-available multicores are still in the single-digit number of cores. This happens because although many new applications and frameworks are being built to take advantage of multicores, a large number of *existing* applications are still sequential. This observation leads chip manufacturers, while continuing their push for developers to target parallel designs, to still spend considerable resources to extract even a small amount of extra sequential performance. Unfortunately, it is not feasible for a vast majority of sequential applications to be rewritten to work in parallel within a reasonable time frame. Additionally, it is still generally harder and more costly to develop correct and efficient parallel software than it is to do sequential software. To tackle both of these issues, an enticing option is thus to use an automatic approach to parallelization.

Parallelizing compilers [4,26] work by breaking up an application into several independent tasks. If the compiler is able to prove non-interference of the tasks, the application is changed to execute them in parallel. More recently, speculation-based parallelization systems were proposed, that work by running parts of the application in parallel even if the parallelization system is not able to prove statically that the result will be correct; correctness validations are instead dynamically performed at run-time. There have been numerous speculative parallelization systems proposed: many (e.g. [15,16,20,23]) target parallelization of loops and very small code blocks; others (e.g. [8,28]) depend on non-standard hardware with custom speculative features. This leaves

* Corresponding author.
*E-mail addresses:* ivo.anjo@ist.utl.pt (I. Anjo), joao.cachopo@ist.utl.pt (J. Cachopo).

http://dx.doi.org/10.1016/j.jpdc.2015.09.005
0743-7315/

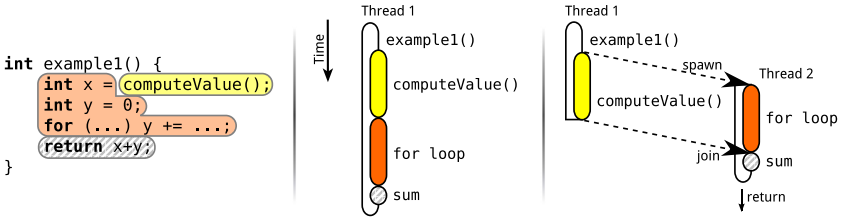


**Fig. 1.** Normal (center) and parallelized with MLS (right) executions of `example1()`.

out most commodity multicores and many dynamic and irregular applications.

In this article, we describe our work on designing JaSPEx-MLS, a speculative parallelization framework employing Method-Level Speculation that targets irregular and dynamic applications that run on the Java platform. Our framework is entirely software-based, needing no custom hardware extensions, and relies on Software Transactional Memory to enable speculative code execution. We position our framework as a fully software-based application boost mode similar to the hardware boost modes provided by modern chips such as Intel's Turbo Boost [6] and AMD's Turbo Core that aims to take advantage of unused CPU cores to improve single-threaded code execution performance. Unlike other proposals aimed at managed runtimes (e.g. [19,22]), we base our work on top of the OpenJDK HotSpot Java Virtual Machine – a state-of-the-art, production-quality managed runtime with dynamic optimization and garbage collection – allowing us to achieve speedups over the normal execution run times of applications using the widely-deployed Oracle JVM.

In this article we make the following contributions: (1) We present a technique for automatic conversion of returned values from methods into futures, allowing them to be stored both in place of the original local variables and also on the heap, with on-demand resolving whenever needed (Section 3); (2) We describe the run-time life-cycle of tasks, and propose two novel techniques to make better use of the resources available on a multicore machine: *hybrid thread pool buffering* and *task freezing* (Section 4); (3) We introduce a new STM model that is optimized for speculative parallelization: our model distinguishes between the thread running in program-order and other speculative threads; supports keeping futures in the write-set as placeholders for values not yet resolved; includes support for Return Value Prediction (RVP); and adopts a variant of captured memory, allowing low-overhead access to objects created by the current transaction, while still allowing safe concurrent accesses from other transactions (Section 5). We present in Section 6 experimental results obtained with our framework, explore related work in Section 7, and end in Section 8 with our conclusions.

## 2. The JaSPEx-MLS framework

JaSPEx-MLS is a fully software-based parallelization framework employing Method-Level Speculation (MLS) and Software Transactional Memory (STM) that is aimed at sequential irregular and dynamic applications that target the Java/JVM platform. Note that while our framework is also able to tackle method-based regular applications, in this exposition we will focus on the framework features that are aimed at irregular and dynamic applications, which may employ both pointer-based data structures with access patterns that are normally not known until run-time and abstractions such as inheritance, polymorphism, and encapsulation, which also complicate static analysis. Our framework is implemented in Java, and modifications are done via bytecode rewriting.

MLS is a parallelization strategy shown to be a promising source for parallelism by several researchers (e.g. [7,18,19,25]) that works by using method calls as speculative task spawn points—speculatively executing the code following the return of a method call in parallel with the method call itself. An example is shown in Fig. 1: when the `computeValue()` method call is reached, it triggers the spawn of a new task to execute the code immediately following the return of the method. We refer to the existing task that goes on to run the method as the *parent* task, whereas the newly-created task that runs the continuation of the method is its *child*. This relation implicitly imposes a global order on all tasks that mirrors the sequential execution order of the original application.

The JaSPEx-MLS framework is divided into two parts: a class-loader and a runtime component. Before an application class is loaded, its bytecode goes through two sets of modifications in preparation for parallelization. The first set of modifications – *transactification* – prepares application code to execute speculatively, changing it so that all memory accesses are done under the control of our STM, and additionally adding hooks to allow non-transactional operations to execute safely and to disallow dangerous operations from executing, as described in [1]. The second set, detailed in Section 3, comprises the selection and transformation of method call sites into speculative task spawn points that return futures. After transformation, the modified bytecode is loaded into the JVM and starts executing under the guidance of the JaSPEx-MLS runtime.

To implement MLS, our framework relies on our own implementation, based on [27], of first-class continuations on top of the OpenJDK HotSpot JVM, as described in [2]. OpenJDK is the result of the open-sourcing of Oracle's Java technology, and by working on top of HotSpot, JaSPEx-MLS benefits from all the features of a modern production JVM: just-in-time compilation, adaptive optimization, advanced garbage collection, and optimized concurrency primitives. We believe that this combination of software-only speculation on top of a modern production JVM is the distinguishing feature of our work.

## 3. Spawn and future insertion

In this section, we describe the modifications performed to an application so that it can be parallelized using MLS. As MLS employs method calls as speculative task spawn points, the JaSPEx-MLS framework needs to instrument method call sites so that they can be used to trigger the creation of new speculative tasks.

As a first step of the MLS modifications, JaSPEx-MLS must select call sites for conversion into speculative task spawn points. This selection is performed taking into account both global profiling information and local static analysis.

Global profiling information is gathered using JaSPEx-MLS's profiling mode. The profiling mode works by modifying most call sites to spawn speculative tasks, and then executing the application sequentially, taking a number of measurements: time spent executing speculatively vs program mode, commit and abort rates, etc. In addition to the profiler output, knowledge from profiling with third party tools or a programmer's own knowledge about the application may also be supplied as input for the call site selection process.

```
1  int example1() { /* Original */      int example1() { /* Conversion for MLS */
2      int x = computeValue();              Future<Integer> x = spawn computeValue();
3      int y = 0;                           int y = 0;
4      for (...) y += ...;                  for (...) y += ...;
5      return x+y; }                        return x.get() + y; }
```

**Fig. 2.** Conversion of `computeValue()` into a speculative task spawn point returning a future.

```
1   public int fieldx;
2   int example2() { /* Original */
3       fieldx = computeValue();
4       ...
5
6   int example2() { /* Unsuccessful naïve conversion for MLS */
7       Future<Integer> temp_fieldx = spawn computeValue();
8       // Retrieve value from Future temp_fieldx, and store it in fieldx
9       Transaction.storeInt(this, temp_fieldx.get(), $offset_fieldx);
10      ...
```

**Fig. 3.** In this example, the return value from the `spawn` is saved onto a field, causing a naïve MLS conversion to be unsuccessful, as `get()` gets immediately called after the `spawn`.

Local static analysis is used to determine if within a given method there is a non-trivial amount of computation between a task spawn point and its expected synchronization point. We consider the work performed as non-trivial if between a candidate spawn point and either its synchronization point (an operation that operates on its return value) or stall (due to a non-transactional operation being found) we find one of the following: (1) *Method calls*. As these calls will execute in parallel with the call from the spawn point, we consider them as performing enough work. These method calls may themselves be candidate spawn points, and we evaluate each one separately; (2) *Backwards edges*. These are indicative of loops, that we optimistically consider as performing enough work; (3) *End of the method*. Whenever the method ends without a synchronization point or a stall being found, this means that the method will return and be able to continue speculatively executing its caller. Categorization of candidate spawn points is performed by a flow analysis algorithm that simulates code execution, tracing where futures would be created and used inside a method.

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

### 3.1. Spawn insertion

Returning to the example from Fig. 1, let us consider that the call to `computeValue()` has been selected for conversion into a speculative task spawn point. To do so, JaSPEx-MLS replaces the call to `computeValue()` with a call to a special framework-internal method that receives as argument an automatically-generated `Callable` that represents the original method call. As a simplification, in this article we represent this modified call using the conceptual `spawn` keyword.

### 3.2. Future insertion

After transforming a call site into a spawn point, a new issue arises: dealing with the return value from the method. As a method call and its continuation are executed concurrently, the return value is not yet available when the speculative task is spawned—it will only be available later, when the method finishes working. Because application code expects method calls to return values, a challenge presents itself: how can we deal with their absence?

A possible solution for this challenge is to predict the return value of the method (Section 5.6). Unfortunately, this technique is not always useful, as for instance a predictor cannot replicate a method that always returns a new object instance. The alternative solution employed by our framework is to have the task spawn operation return a placeholder for the computed value, in the form of a future. As an example, consider Fig. 2: the conversion entails changing `x` from an `int` into a `Future<Integer>`, and then changing accesses to `x` to instead call the `get()` method from the future. In this example the conversion is very straightforward, but depending on its type and how the return value from the invoked method is used, the JaSPEx-MLS framework may need to perform additional modifications:

*Return value is `void` or not needed.* Whenever the target of the `spawn` operation is a `void` method, or a value is returned but never actually assigned, no further modifications need to be made to the method other than adding the `spawn` call.

*Return value is consumed immediately.* On some call sites, the result from the invoked method is immediately used without being explicitly saved to the heap or onto local variables. A `get()` that immediately follows a `spawn` makes the `spawn` useless, only adding overhead, and JaSPEx-MLS's call site selection algorithm rejects such cases unless when, optionally, RVP is enabled.

*Heap Write: Return value is written onto a field or an array position.* An example of this case is shown in Fig. 3, where a `spawn` operation is immediately followed by a `get()`. In this case, we take advantage of the value itself not yet being needed, as any write operation to a given location only needs to take effect before the next read operation to that same location (or discarded before the next write). Thus, we can reorder the real write until a read operation needs to be executed. To allow this reordering to occur, our STM model was designed to allow the registration of intents to write instead of concrete values, thus enabling our framework to parallelize methods that would normally not be able to be parallelized by other MLS frameworks. Under this model, futures can be written onto heap memory locations as temporary replacements for the original values. We can see this reordering in Fig. 4: although the modification performed is similar to the previous version, in the new version the actual return value from `computeValue()` is not required for the field write to succeed. We chose to perform this reordering via the STM and not by reordering the write operation inside the method as it has a number of advantages: (1) It allows the reordering to work lazily, if, for instance, the write can be ignored in some paths through the code; (2) It enables the reordering to work correctly in the presence of method calling, where some of them may also share access to the same heap location (e.g. the future may be written in a method call and only accessed in another); (3) It simplifies bytecode transformation.

```
1  int example2() { /* Conversion for MLS with STM support for Futures */
2      Future<Integer> temp_fieldx = spawn computeValue();
3      // Store the Future in temp_fieldx directly into fieldx
4      Transaction.storeFutureInt(this, temp_fieldx, $offset_fieldx);
5      ...
```

**Fig. 4.** Successful conversion for MLS of `example2()`, using STM support for futures. In contrast with Fig. 3, the STM stores the future as if it was written to `fieldx`.

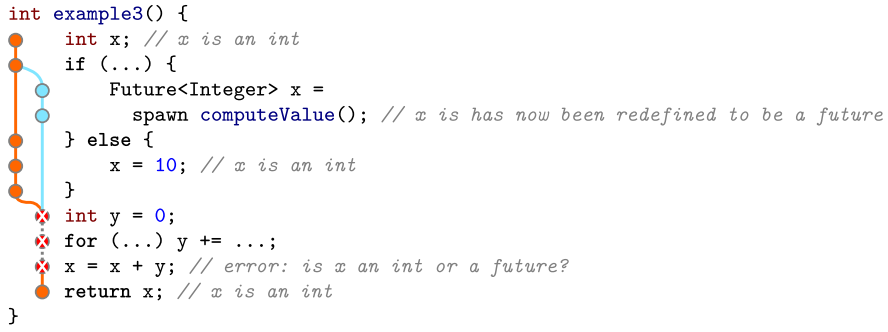


**Fig. 5.** Problematic replacement of a returned value with a future. Execution may reach line 9 via lines 4 and 5, in which case x is a `Future<Integer>` ( ) or via line 7 in which case x is still an `int` ( ), resulting in an error. tags lines where there is an inconsistency.

*Stack Write: Return value is kept on the stack or is saved onto a local variable.* JaSPEx-MLS takes advantage of the JVM specification allowing local variable slots to have any type to substitute the future for the original value in the same local variable or stack slot. After this substitution, variables now contain futures, and thus any reads must be prepended with a call to the contained future's `get()` method so as to retrieve its value. This modification is non-trivial: it entails tracking every read of a local variable (or stack slot) containing a future and adding a call to `get()` to obtain the concrete value from the future immediately before it is used. Unfortunately, when used blindly this substitution may lead to issues when multiple branches lead to different types being possible for the same variable, as exemplified in Fig. 5. To solve this issue, we again turn to our semantic bytecode analyzer. As a first step of the future insertion algorithm, JaSPEx-MLS identifies regions of the method under analysis where a variable's type is inconsistent—where the type may either be the original type or a future depending on the control flow followed through the method. To detect inconsistencies, we simulate the state of the stack and the local variables, keeping for each position a set of the expected types for that position. Afterwards we check for problematic instructions: those that attempt to operate on a value where the set of possible types for that value is bigger than one (e.g. the original and the `Future` types). To fix the problem, our framework duplicates basic blocks containing problematic instructions, afterwards changing the method by diverting one of the branches that created to jump to the newly-added duplicate region, as shown in Fig. 6. The fixing process iterates over each problematic block (if there are multiple) and may have to be repeated several times if several `spawn` operations are involved. As a safeguard, if too much code gets duplicated, our framework undoes the placement of the `spawn` and the subsequent modifications, as the VM will not optimize huge methods, negating any performance gains from parallelization.

## 4. Task management

Our framework splits a program into a set of tasks. At any given time, most of these tasks are speculative, and one is not: the non-speculative task that is executing code in program-order. Tasks being executed by JaSPEx-MLS may create further tasks, as part of the `spawn` operation. These speculative tasks may be successful and *commit*; or they may fail or no longer be needed and *abort*.

### 4.1. Spawning a new speculative task

When program execution hits one of the previously-inserted spawn points, the `spawn` operation gets triggered, resulting in a call to a framework-internal method (as described in Section 3.1). Before proceeding, the framework performs an approximate check of the state of the thread pool, to avoid wasting work if the pool is full. If the pool is suspected to be full, we *early reject* the new task, and execute the method call and its continuation normally.

If, instead, the framework decides to spawn a new speculative task, it first creates a new speculative task object. In the current arrangement, the parent task (that triggered the `spawn` operation) will execute the `Callable`, while the new task being created will execute the code that follows the return of the `spawn` operation. As seen in Fig. 1, the parent task's thread state will become the child task's starting state, and will need to be transferred onto the new task's host thread. To transfer state between threads, we leverage on our JVM with support for continuations: the framework captures a continuation containing the current state of the parent task's thread, and then attaches it to the newly-created child task. Afterwards, the new task is submitted to the thread pool. Between the earlier approximate check and the submission of the new task for execution by the thread pool, the state of the pool may change, and the pool may reject the new task whenever it is full—we call such cases *late rejects*. The cost of an early reject is comparatively small: it entails the creation of the `Callable`, the `spawn` operation, and the approximate thread pool check. Late rejects are costly as they include the creation of the new task object, the continuation capture operation, and task submission to the pool. As avoiding late rejects is important for performance, we present in Section 4.3 a technique to reduce them.

After the new speculative task is successfully submitted to the thread pool, the parent proceeds to execute the `Callable` it received. Note that because the JaSPEx-MLS framework supports nested speculation, the parent task may itself be speculative. Whenever a thread from the pool picks up the new speculative task for execution, it resumes the continuation containing the state received from the parent task, and starts a new STM transaction: the `spawn` is now complete, and execution of the code that follows the `spawn` point begins.

Note that the parent/child relationship between tasks is dynamic, and can change during a task's execution. JaSPEx-MLS supports both *out-of-order* and *in-order* task spawn: *out-of-order* spawning allows the same parent to create several child tasks, its

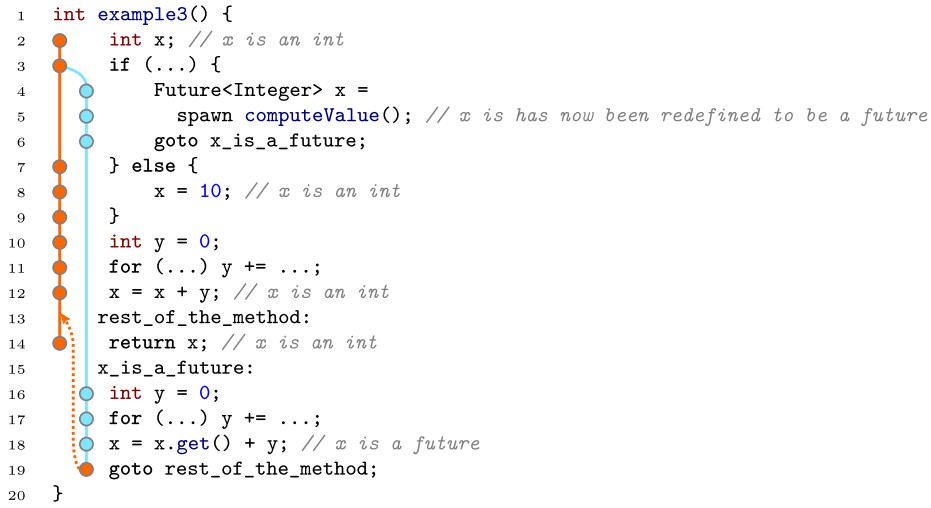


**Fig. 6.** Corrected version of `example3()`. Lines 10–12 are duplicated onto lines 16–18, so that either the sum is performed with `x` containing an `int` (line 12) or a future (line 18).

name coming from the observation that newer tasks are created in an order that does not follow the original sequential program order. *In-order* spawn happens when the most speculative child task spawns a further child task, and that task repeats the same process: new tasks are created, but as their creation order follows from the execution order in the original sequential program, parenting relationships are not changed after task spawns. Combining both models creates a very flexible model for extracting parallelism from applications, but also poses challenges to task ordering, as discussed in Section 4.3.

### 4.2. Completing and committing a speculative task

Validation and posterior commit/abort of a speculative task happens whenever: (1) the task finishes its work; (2) the task attempts to execute a non-transactional operation; (3) the STM detects that the task's transaction is doomed to abort; or (4) the task's parent finishes its own execution and signals that the child can also commit its work. A speculative task may only validate its work and attempt to commit if it is the oldest-running task in the system—that is, if all of its predecessors, including its parent, have finished their own execution and committed successfully. Whenever a task is ready to begin validation, but no result from its parent is available, it must wait for this value to become available. Note that it may be possible for the parent itself to be in the same situation, and for a sequence of speculative tasks to all be waiting for their own parents. After the parent successfully finishes, the speculative task attempts to validate its own STM transaction. Whenever validation fails, the transaction must be aborted, and the task is either discarded if it was the result of a wrong spawn decision, or retried by re-resuming the continuation received from the parent; as before it can validate itself, the task is guaranteed to be the oldest in the system, this means that the re-execution is performed in program-order mode, and thus any task is executed at most twice: once speculatively, and the second time, whenever needed, in program-order. Whenever validation succeeds, the task commits its work. Afterwards, the task either continues working in non-speculative mode, or, if it had already finished its work, is marked as finished.

### 4.3. Thread pool management and hybrid thread pool buffering

After a new speculative task is created, it is submitted for execution to JaSPEx-MLS's thread pool, which is configured to use a limited number of threads, based on the number of available CPUs in the machine.

In our first design, the thread pool used direct hand-offs between threads. This meant that new tasks were only accepted if there were any idle threads. This design was chosen to avoid deadlocks: because, as described in Section 4.1, our design allows for both *out-of-order* and *in-order* speculative spawn operations, task spawn order becomes unpredictable—this unpredictable order, when combined with waiting means that if the pool uses buffering the system can end up deadlocked. Note that the use of a bigger pool is only a band-aid solution, as deadlocks can happen with any pool that features a bounded number of threads, and, in addition, employing significantly more threads than those supported by the host machine greatly increases scheduling and context switching overheads. By avoiding buffering, JaSPEx-MLS ensured that at least one of the threads in the system was making progress, as it was hosting the oldest task in the system, which would never need to wait. This scheme also caused thread usage to be sub-optimal: because no speculative tasks were accepted unless there were available idle threads, this meant that when a thread finished its work, it would be idle until another active thread reached a spawn point. In cases where an application had large tasks, this would mean that for long periods of time only a few or in the worst case only one of the program threads would be working.

As benchmarking revealed that task buffering when it did not cause any issues was more efficient than direct hand-overs to the thread pool – in part because it reduced both early and late rejection rates (Section 4.1) – we now introduce a novel technique to work around this issue: *hybrid thread pool buffering*. This technique works by keeping both a buffered and a non-buffered work queue for managing tasks. By default, the buffered work queue is employed, allowing more tasks to be generated than there are threads in the pool to work on them. The thread pool is then augmented with a dedicated thread that periodically polls the state of the buffered queue. Because tasks are queued at most once, if the polling thread observes the same task at the head of the buffer for a given time period, it triggers a check of the current state of each of the pool's threads. If all threads are observed to be waiting for an event to wake them up, we conclude that the system is deadlocked. Note that there is no impact to program correction if the polling thread suffers from a data race when checking for deadlocks, as it will only mean that the system will pessimistically transition to direct hand-offs when it would have not needed to; In practice, we never observed this to happen. As such, buffering is disabled, with the pool being switched to only

accepting direct hand-offs. To break the deadlock, new temporary threads are created for each of the tasks still present on the buffered queue. Because no new tasks are accepted (as all threads are busy) and all the tasks in the buffer are executed, the task causing the deadlock is thus guaranteed to be executed, breaking the deadlock. Afterwards, the temporary threads are retired. Our hybrid approach is thus able to provide the performance of task buffering and still ensure correctness by falling back to the earlier approach whenever needed.

### 4.4. Task freezing

During execution, a task may need to wait for another task to finish. Unfortunately, threads hosting waiting tasks are unavailable for picking up new work, thus leaving the machine's parallel resources underused. A possible approach to solve this challenge would be for a thread to pick up a new task whenever it was about to wait, but this approach is problematic because the older task stays pending on the thread's stack, and if the newer task ends up depending on the value from the older task, the system would be unable to ever finish either task.

To safely support thread reuse, we developed the concept of *task freezing*, that solves the above issue by relying on our extended JVM with support for continuations. Whenever a thread executing a task would block, instead we *freeze* the task, by capturing a continuation containing the current state of the task and saving its currently-active STM transaction. This snapshot of the task's stack and execution state allows the task to later be reconstructed in the same or other thread, with the STM transaction being used to keep the speculative heap state belonging to the task. A frozen task is associated with its parent task, which will be responsible for finishing the frozen task's work after its own—frozen tasks are not submitted to the thread pool again. After a task is frozen, the thread that was hosting it is returned to the thread pool, where it can safely pick up other tasks for execution. The *thaw* operation happens when, after finishing its work, the parent task discovers a frozen child waiting to be completed. As the parent has finished its own work, its thread can directly switch to working on the child without needing to return to the thread pool—the child's continuation is resumed, its STM transaction is validated, and execution proceeds from where the freeze left off. Note that it is possible for a queue of frozen tasks to form, and a parent may have to thaw several children, always directly switching between them without returning to the thread pool.

Because capturing continuations adds overhead, we have further identified and optimized a common case where we can avoid the need for capturing continuations. Whenever a child task is able to complete its work, but still needs to wait for its parent to finish before it can validate and commit its own speculative state changes, we can use a *lightweight task freeze*: because the task has finished its work, there is no longer any thread state that needs to be preserved, and thus only the task's STM transaction and return value are preserved.

## 5. Custom STM model

A common complaint of Software Transactional Memories is that their added overhead is non-negligible when compared to other synchronization alternatives. JaSPEx-MLS's custom STM model aims to take advantage of specificities of the speculative parallelization model to minimize overhead as much as possible. As a consequence, the proposed model is unsafe as a generic STM model, and as we describe our STM model in the following sections, we highlight why and how it takes advantage of specific characteristics of the speculative parallelization model to be simpler and more efficient.

### 5.1. Thread execution modes and relaxed isolation

JaSPEx-MLS's STM recognizes two execution modes for threads executing tasks: there is a single *program-order* thread, and multiple *speculative* threads. The program-order thread is the one running the oldest (in terms of original sequential execution timeline) code from the application. Actions performed by the program-order thread are not speculative—they are executed with the same state and data that the normal sequential code would have, and they never have to be undone. In contrast, speculative threads are running tasks that are still speculative—they have not been validated and their work might still need to be undone. As their work is speculative, and their results are tentative, these results should be isolated from other concurrently-running speculative threads.

Because the program-order thread is not running speculatively, we allow it to directly access and mutate the program state without extra instrumentation. This direct-access strategy substantially lowers the overheads of code running in program-order, which is especially important whenever a stretch of program is being executed sequentially without any parallel tasks. As consequence of this modification, isolation is relaxed between the non-speculative thread and other concurrently-running speculative threads: speculative threads observe heap mutations performed by the non-speculative thread as they happen, and thus it is possible for them to observe inconsistent heap states. This is where speculative parallelization differs from STM: whereas in STM it is very important to enforce isolation between concurrent transactions – otherwise causing inconsistent reads [10], and breaking opacity [13] – under speculative parallelization the model already lends itself to inconsistent states, as it is based on the concept of executing "future" code based on the currently-available tentative version of the program state. Because these issues are already a part of any speculative parallelization system, dealing with them should not be a part of the STM model—it should be up to the framework to identify and discard invalid tasks. Note that while isolation can been relaxed, it cannot be eliminated: the program-order thread still needs to be isolated from speculative threads, as speculative threads are from each other—otherwise, this would mean that code from earlier in the program would be able to observe results from code later in the program.

The breach in isolation between the program-order thread and speculative threads also works as value forwarding: speculative threads optimistically have earlier access to values from the program-order thread than they would have if that thread executed in isolation.

### 5.2. Progress

At any given time a program-order thread is executing, the system is guaranteed to be making progress, as it is executing code in the original sequential program order and that will always follow the original application semantics.

When a non-speculative task ends, and while its child task has yet to switch to executing in non-speculative mode, the system is not guaranteed to be progressing—only speculative tasks are active, and they may not be correct. Because of the early commit feature (Section 5.4), this window is usually very small, and in practice we have found that there is no need for a heavier mechanism that would force the child to stop its work and validate itself immediately.

It is nevertheless possible for all of the tasks on the system to be running speculatively, and none of them be correct: this means that, while these tasks do not attempt to perform validation and are aborted and restarted, the system will not be making any progress, because all the work being performed will be discarded. In this interval, no useful work will be done, but progress will be resumed as soon as a thread switches back to non-speculative mode.

### 5.3. STM design choices

Our STM model uses optimistic concurrency control for speculative threads. Our write-set is a redo-log; writes from speculative threads are kept in a thread-local map and are written-back at commit time. We employ invisible reads: all read operations are unsynchronized and change only the thread-local read-set, thus they are not observable by other concurrent transactions. Each thread keeps its own read-set: by default the read-set is represented using a linked-list, but it can also be represented using a map, whenever there are many repeated accesses to the same memory locations. Note that repeated accesses may happen because application code has been written without any intent to use STM and thus not all working values may be saved to local variables. The read-set is accurate: each individual memory location is recorded for later validation—no false conflicts may ever occur. To avoid adding memory overhead to heap objects, all metadata are both thread-local and transaction-transient, with no changes being needed to the in-memory layout of objects and arrays.

Validation is performed in a value-based manner: a read of a value $v$ from a memory position $m$ is valid if at the time of validation $m$ still contains $v$. Whenever we instead use a map to represent the read-set, we additionally implement early abort: if some memory location $m$ has changed its value since an earlier read, the STM signals to the task management code that the current transaction is doomed and should be early aborted. Using a value-based approach both reduces the need to track object versions and can also reduce aborts.

### 5.4. Commit operation and commit ordering

Unlike with normal STMs, tasks extracted from a sequential application have an implicit order: the execution order from the original application. When these tasks are mapped to transactions, they must still follow this order, allowing us to simplify several things. The commit operation does not need to be atomic, as there is no problem if other threads observe partial commits, as the STM will flag issues during their own validation. Only one thread – the one hosting the oldest task – that is transitioning from speculative into program-order mode, is allowed to commit at any one time, further simplifying synchronization.

To commit a transaction, we first validate its read-set. Note that validation always needs to happen, even for read-only transactions, as we need to ensure that a transaction accessed the correct heap state while executing—unlike previous differences, this requirement adds overhead not normally present on other STMs. Because no older transactions may exist in the system at commit-time, validation also does not need any kind of extra synchronization to be correct. If validation succeeds, we write-back all values from the write-set, and the transaction is finished.

When a program-order thread finishes its execution, it is returned to the thread pool, and its child task can now validate its work and become the new program-order thread. Similarly to the *implicit commits* performed by [28], the retired task flips a flag on the child task indicating that it should validate its work and continue executing in program-order; this flag is checked during each STM read and write operation. This *early commit* feature allows threads to switch to program-order mode faster, reducing unneeded speculative execution overheads.

### 5.5. Transparent handling of futures in the write-set

As described in Section 3, our STM directly supports futures on the write-set as placeholders for the returned values from methods. When a task attempts to read from a future previously recorded in its corresponding transaction's write-set, we first call the future's `get()`, waiting if needed for the value to become available. Afterwards, we cache the result, and further lookups will return this value directly. At commit-time, each future is accessed to obtain its value, and this value is written-back to the heap. During commit, because of the strict commit ordering that is enforced, the STM never needs to wait for a future to finish computing—if it found a future, it means that the task that triggered its corresponding `spawn` operation was ordered before the current task, and thus it is guaranteed to have finished its work.

### 5.6. Return value prediction

One of the biggest challenges in the MLS model is dealing with operations that manipulate the return values of methods that have yet to finish executing. Our framework represents the return values of these methods with futures, and our modifications to application code enable futures to be written both to local variables and also to the heap, as described in previous sections. But the above options are useful only if the value from the method call is not read immediately; otherwise, no useful work would be done in parallel.

An alternative solution to this predicament lies with the use of Return Value Prediction (RVP) [12,19]. The idea of RVP is that whenever a task would stall waiting for a value to be produced by another task, the framework instead predicts a probable value, and continues executing the task using this assumption.

We have implemented RVP as a novel extension to our STM model: whenever a prediction is produced, it is registered with the STM as a read of a specially-reserved memory location. This memory location is unique for each task (future) from which we obtain a prediction: it is possible for a speculative task to obtain multiple predictions corresponding to values from multiple other tasks. When a task finishes and produces a return value, it writes it to the special memory location during its commit operation. When later the task that consumed the prediction attempts to commit, the memory location hosting the prediction is checked as part of the normal read-set validation. Because our verification is value-based, if the prediction was correct, its value will be seen as valid by the STM, otherwise the task is aborted. With this extension to our model, the usage of RVP needs no special modifications to the application. By implementing RVP on the STM, it is also possible – although this feature is not yet used by JaSPEx-MLS – to dynamically toggle the usage of RVP on a per-call site basis.

### 5.7. Adapting captured memory to MLS

Most of the STM overhead from threads executing in speculative mode comes from them having to deal with the read and write sets, instead of directly accessing the objects as the program-order thread does. As this interception is a source of overheads on many STMs, reducing them has been an area of active research for STM developers. The authors of [11] identified that some memory accesses are made to what they define as *captured memory*: memory that is allocated inside a transaction and that cannot escape its allocating transaction while it is ongoing. This observation is derived from the knowledge that due to isolation, objects and arrays created inside a transaction will not be accessible to any other transaction until their creating transaction finishes, if at all. Because they are accessible only to their creating transaction, objects in captured memory may instead be mutated directly without causing concurrency issues.

Yet, combining captured memory and MLS poses a challenge: whereas under a normal STM transactions run isolated, and any newly-created objects and arrays are inaccessible to other transactions, under the MLS model this does not hold. As an example, consider Fig. 7: the parent task creates a new instance

**Table 1**
Specifications for the test machines.

| Machine | Specifications | CPU cores | Hardware threads | Boost |
|---|---|---|---|---|
| *M*1 | Intel Core i7 4770, 16 GB | 4 | 8 | *Turbo Boost* |
| *M*2 | Intel Core i5 750, 8 GB | 4 | 4 | *Turbo Boost* |
| *M*3 | 2 × Intel Xeon E5520, 24G B | 8 | 16 | *Turbo Boost* |
| *M*4 | 4 × AMD Opteron 6168, 128 GB | 48 | 48 | No |
| *M*5 | AMD Phenom II X6 1055T, 12 GB | 6 | 6 | *Turbo Core* |

**Table 2**
Benchmarks used for testing JaSPEx-MLS.

| Benchmark | | Description | Workload |
|---|---|---|---|
| | *Avrora* | Simulator for AVR microcontrollers | `fib_massive.asm` |
| | *Sunflow*[a] | Raytracer | `aliens_shiny.sc` |
| *JGF* | *Series* | Fourier coefficient analysis | `SizeB` |
| | *MonteCarlo* | Financial simulation | |
| | *Euler* | Flow equations solver | |
| | *Lonestar Barnes–Hut* | Gravitational force simulation | `runC` |
| *Aparapi*[b] | *Mandel* | Mandelbrot set renderer | – |
| | *Life* | Conway's game of life | – |

[a] Sunflow's internal threading support was disabled, leaving only a single-threaded renderer.
[b] Benchmarks adapted from the Aparapi project.

```
1  void example4() {
2      SomeClass sc = new SomeClass(null);
3      int x = spawn computeValue();
4      sc.field = 42;
5      sc.otherField++;
6      ...
```

**Fig. 7.** Example method and its task division. The parent task (in red) creates object `sc`, spawns a new task, and goes on to execute the `computeValue()` method; the new task (in green) executes the code following the execution of `computeValue()` and accesses `sc`. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

of `SomeClass`, making this instance part of the parent's captured memory; but in lines 4 and 5 the child task also accesses `sc`, breaking the assumption that objects in captured memory are not accessible to concurrent transactions.

Even though at first it appears that it would not be possible to take advantage of captured memory in MLS, we can leverage the fixed commit order imposed by the MLS model to work around this issue. The key insight is that whenever a speculative task $st_1$ creates an object, and another task $st_2$ is able to access it, then by the MLS model's definition $st_2$ must be more speculative than $st_1$, which created the object. Thus, and although concurrent tasks may have access to objects from captured memory, the MLS model guarantees that their writes will come later than those from the creating task, which always precedes them. As such, this is very similar to the semantics that our STM already provided between the program-order and speculative threads, and we extended our model by treating accesses to captured memory as if they are done by a task running in program-order mode, with other more speculative threads using the normal STM mechanisms to protect and validate their accesses. To detect which objects are part of a transaction's captured memory, we adopted LICM [5], a technique that works by tagging every new object with the fingerprint of its creating transaction. Of special interest to the MLS model, the addition of captured memory also reduces transaction aborts. Consider again, for instance, the access in Fig. 7 to `otherField` (line 5): if this field was initialized inside `SomeClass`'s constructor, this initialization would still be on the parent task's (uncommitted) write-set, and so any access done by the new task would not be able to observe the correct value for the field. Using captured memory, the parent task writes directly to the field, and this write is available to the child task. Without captured memory, accesses to newly-created objects do not observe the correct values while they are still being kept on the creating transaction's write-set. By allowing direct writes to objects, this source of mis-speculations is removed.

## 6. Experimental results

In this section, we present experimental results of application execution using the JaSPEx-MLS framework. Table 1 describes the specifications of our test machines. Both boost implementations were left enabled for our testing; because of this, processor frequency is usually faster for the sequential versions of applications than whenever JaSPEx-MLS is being used, as our usage of multiple threads normally precludes boost modes from kicking in. Nevertheless, we believe that this configuration makes our results more realistic and a better representation of expected end-user machine configurations. The chosen benchmarks for our testing are listed in Table 2. We excluded some benchmarks from the JGF and Lonestar test suites as they were either too small and unrepresentative of modern workloads (with sub-second execution times, including VM startup), or unsuitable for MLS, as our profiler rejected the majority of call sites as being non-profitable for speculation. As such, these applications are not amenable to parallelization using our current model, and their execution using JaSPEx-MLS would necessarily not yield any positive results. We believe that some benchmarks – especially those from the Lonestar suite – would be able to be tackled after some manual modifications, but in this work we are mainly focused on automatic parallelization of applications without requiring access to the application's source code. All benchmarks were run on our custom OpenJDK HotSpot JVM with support for continuations. The sequential performance of this JVM closely follows Oracle's Java 6 SE VM builds, as they share the same codebase. Unless otherwise stated, each benchmark was executed five times, and the results presented are the average of all five runs.

### 6.1. Execution overheads

To isolate and characterize the overheads introduced by the bytecode changes performed by our framework, we compared the original sequential execution times for our benchmarks with those from executing the same benchmarks after bytecode modification, but still in single-threaded mode.


**Fig. 8.** Impact of JaSPEx-MLS's bytecode modifications on sequential execution performance. Values are shown normalized to the execution times of the original sequential applications.


**Fig. 9.** Impact of JaSPEx-MLS's bytecode modifications on sequential execution performance when using only OpenJDK HotSpot's interpreter mode. Values are shown normalized to the execution times of the original sequential applications when using the interpreter mode.

Fig. 8 shows the results from our testing using Machine *M*1; the results for other machines are very similar and as such were omitted. The *Transactional Execution* results only reflect the bytecode modifications needed for transactification, while the *All Modifications* results include both transaction and the spawn point/future insertion (Section 3). For these tests, JaSPEx-MLS uses a thread pool with only a single thread, causing all task spawns to be early rejected.

The biggest impact to the execution time – 10.6× of the original application's execution time – happens in the *Euler* benchmark. This benchmark has very large methods, in some cases with 150+ LoC and resulting in 1500+ bytecode instructions after compilation. As such, the transactification of these methods, even without any other changes to inject spawn points, crosses the JIT compiler's maximum code size threshold, so the modified versions of methods are never properly compiled and optimized. For the remaining benchmarks, the impact of the performed bytecode modifications ranges from 1.0× to 1.3× the original execution times. As we will see in the next test, this happens because most of the overheads are being removed by the OpenJDK VM's JIT compiler.

The presented results show that although the JaSPEx-MLS framework performs many changes to the applications' bytecode, non-speculative execution by the program-order thread is not severely impacted. This enables applications with a mix of parallel and sequential workloads to still benefit from our framework.

To analyze the impact of not using an optimizing JIT compiler, and to simulate the execution of the JaSPEx-MLS framework on top of a simpler virtual machine, Fig. 9 shows the results of repeating the benchmarks from Fig. 8 while disabling HotSpot's JIT compiler and forcing the interpreter to be used for the entire program execution; most applications are now heavily affected: apart from *Series* and *Mandel*, other applications take at least 5× longer to execute, with some taking about 10× and *Euler* taking 33×. These results confirm our assertion that having a production VM with an advanced JIT compiler is crucial to our approach of performing speculative parallelization on top of the VM.

### 6.2. Characterizing the impact of a production VM

Our approach of building atop HotSpot contrasts with past attempts at software-based automatic Java parallelization, which, as described in Section 7, build atop simpler research VMs, which are more amenable to the multiple changes needed for speculative execution. In particular, we highlight two Java-based systems which build atop simpler VMs: SableSpMT [19] and HydraVM [22]. SableSpMT is built atop the SableVM, which only provides an interpreter, and HydraVM is built atop Jikes JVM's lowest performance compiler (*baseline*).

Fig. 10 shows the results of testing our chosen benchmarks with those VMs (*Barnes–Hut* and *Sunflow* were excluded as they did not work). The results are shown normalized to HotSpot's normal execution mode, and similarly to the previous sections they were benchmarked using Machine *M*1. As expected, other VMs cannot generally compete with HotSpot. On average, execution times are 6.32× more than HotSpot for Jikes, and 15.06× for SableVM. This means that speculative parallelization systems built atop these VMs would have to extract speedups capable of overcoming these performance gaps; otherwise, users would just be better off using HotSpot.

### 6.3. Performance improvements from task buffering and freezing

In this section we analyze the impact of the proposed techniques for optimizing the use of machine execution resources: hybrid thread pool buffering and task freezing. We present results from speculatively parallelized executions of the chosen benchmarks in three different configurations: only task buffering enabled, with freezing disabled; only freezing enabled, with task buffering disabled; and both task buffering and task freezing enabled. These results are normalized to a fourth configuration, where both task buffering and task freezing are disabled. All results were gathered in Machine *M*1 with hyper-threading disabled; due to space limitations we had to omit results from hyper-threading runs. The objective of these benchmarks is to measure the impact

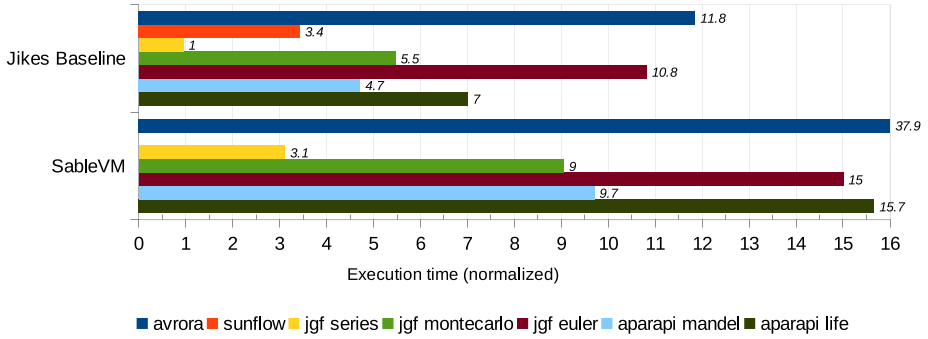


**Fig. 10.** Testing the sequential versions of our chosen benchmarks with other Java VMs.

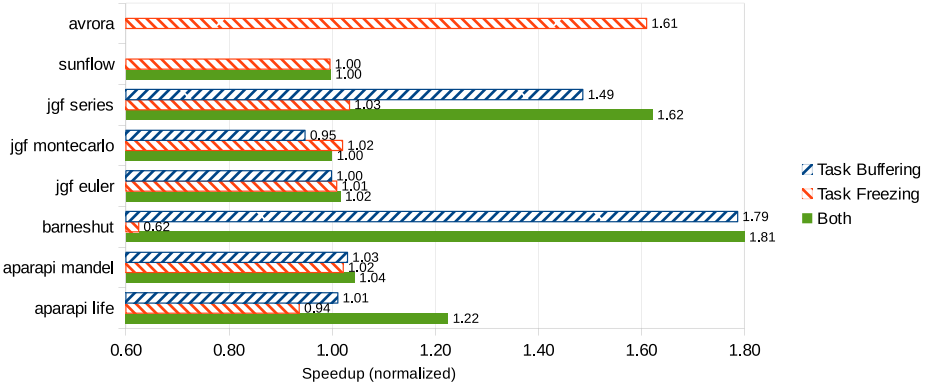


**Fig. 11.** Speedup from task buffering and freezing features.

of these features in the performance of the JaSPEx-MLS framework both in isolation and when used together. The results from our testing are shown in Fig. 11.

The *Avrora* benchmark is not able to take advantage of task buffering, as it always leads to deadlocks and to the thread pool disabling buffering at run-time. Nevertheless, task freezing alone is able to improve performance for this benchmark. The *Sunflow* benchmark also suffers from deadlocks when task buffering is used, but when buffering is combined with freezing no deadlocks are observed. Unfortunately, neither task buffering nor task freezing has impact on this benchmark. The *Series* benchmark shows a very good improvement from using task buffering. We also see that when used alone, task freezing has little impact on this benchmark. In contrast, when task freezing is combined with task buffering, this feature is able to improve on task buffering's performance boost. Neither *MonteCarlo* nor *Euler* is able to take advantage of buffering and freezing, which have minimal impact on their performance. The *Barnes–Hut* benchmark shows very good improvement from using buffering whereas freezing has a detrimental effect when used alone and only a slight positive effect when used in combination with buffering. The *Mandel* benchmark shows a small improvement from both features. The *Life* benchmark once again shows that while task buffering alone improves performance for most benchmarks, freezing is able to improve performance especially when combined with buffering.

### 6.4. Speculative parallelization benchmarks

In this section, we present results from measuring the speedup of executing each benchmark while being parallelized by the JaSPEx-MLS framework. Note that due to the heterogeneity of core counts and hyper-threading options, some core configurations are only presented for a subset of the machines. For each benchmark, we first test with no speculation (as graphed in Figs. 8 and 9 under *All Modifications*), and then add results for the different thread (denoted by *t*) and hyper-threading (denoted by *ht*) configurations. Note also that the results presented use the best framework settings for each benchmark at each data point, with nested speculation always enabled. The presented results are normalized to the execution time of the original unmodified sequential applications on each of the test machines. The results from this testing are presented in Fig. 12.

The *Avrora* benchmark performs some speculation, but task size ends up being very unbalanced, even after being profiled with JaSPEx-MLS's profiling mode. The observed slowdown comes from speculative execution overheads on very large tasks that end up dominating the running time. The *Sunflow* benchmark ends up being mostly single-threaded. While *Sunflow* itself has built-in support for multithreading, JaSPEx-MLS is currently not able to extract much parallelism from the single-threaded version of the benchmark without refactoring it to make it more MLS-friendly, and no speedup is attained in this benchmark. Unlike *Avrora*, most of the single-threaded work is performed in non-speculative mode, and as such the observed slowdown is in line with the expected transactification and framework overheads.

The *Series* benchmark yields good scalability up to 8 thread configurations, reaching a speedup of around 4× for most machines, with a maximum of 4.6× on *M*1. After 8 threads, no more parallelism is able to be extracted from the application, and configurations with more than 8 threads yield the same or worse performance results. The *MonteCarlo* benchmark is heavily optimized for sequential performance, avoiding object allocation by reusing many objects, which leads to a very high abort rate due to failed STM validations—around 25%, even after profiling. Too much time is lost on re-executions, penalizing the application's run time, and the extracted parallelism is not able to compensate this impact. JaSPEx-MLS is not able to attain speedup in this benchmark.

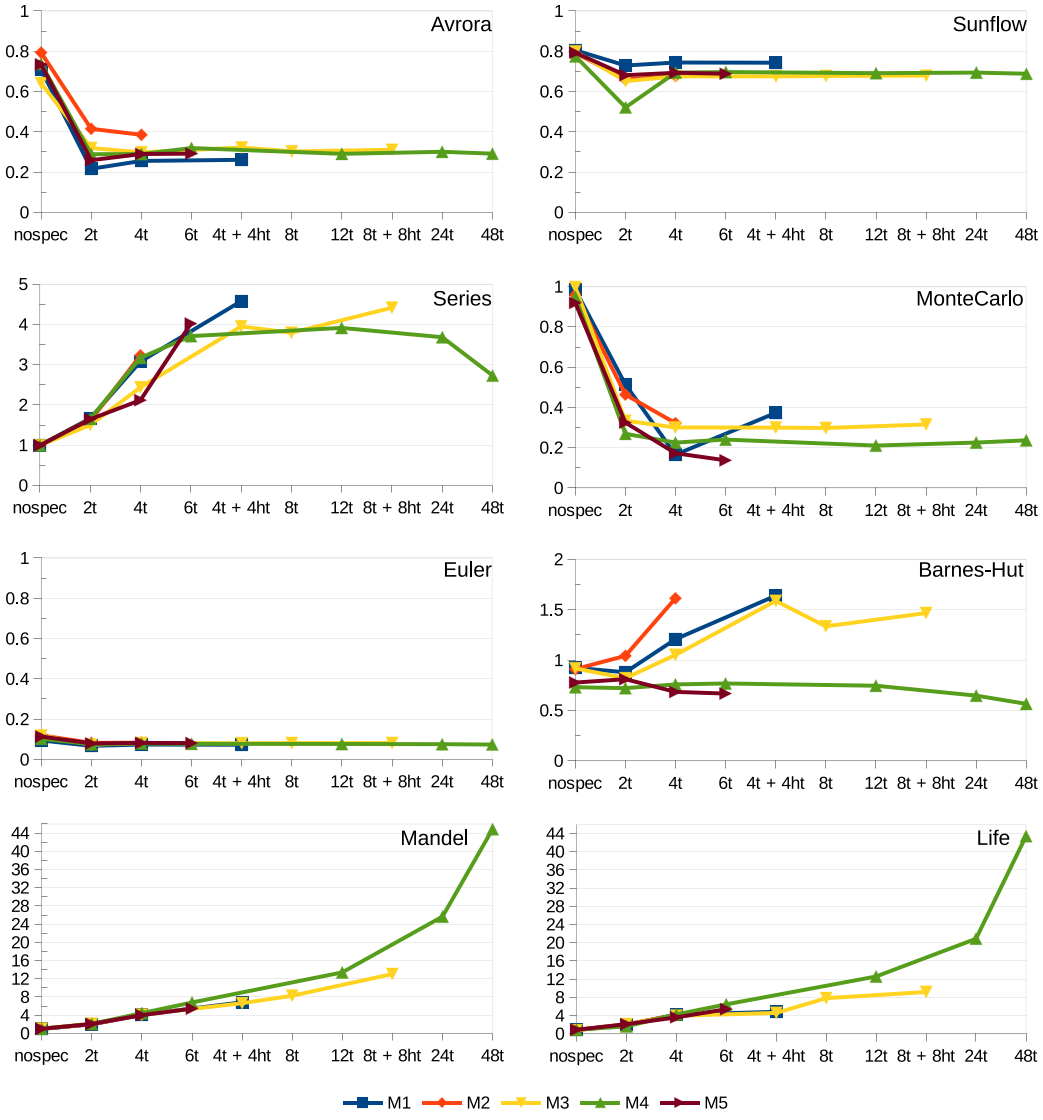


**Fig. 12.** Speedup for each of the tested benchmarks.

The *Euler* benchmark, as expected from the results of the bytecode modification overhead analysis, is not able to recover from the massive slowdown imposed by the JIT compiler declining to optimize the application's methods, as evidenced by the poor results even in the *nospec* configuration, which on other benchmarks is usually ranged 0.8–1×. The *Barnes–Hut* benchmark yields very interesting results: in this benchmark, all three Intel-based machines ($M1$–$M3$) are able to attain a speedup of around 1.6×, while both AMD-based machines ($M4$ and $M5$) are never able to even match the performance of the original application. Intel's architectures are able to surpass AMD's in this test, highlighting both the difference in microarchitecture implementation and the potential for future improvements on AMD's part.

Finally, both *Mandel* and *Life* implement easily-parallelizable regular workloads, and are able to scale linearly with the number of CPU cores on the test machines, with both reaching a speedup of around 44× on our 48-core test machine.

## 7. Related work

Before application code can be speculatively executed in parallel, it needs to be transactified. Because transactification can have a non-negligible impact on application execution performance, many researchers have proposed the usage of hardware speculation mechanisms [8,17,28]. While we believe that special hardware support would benefit our framework greatly, its current lack of availability clashes with our goal of providing a software framework to boost applications on commonly-available multicore processors.

Recent software-based parallelization systems try to tackle the overheads of transactification, similarly to JaSPEx-MLS, by optimizing the transactification and transactional model as much as possible: in SpLIP [16], a speculation system that targets mostly-parallel loops, the authors propose that speculations perform in-place updates and commit their work in parallel, thus lowering execution overhead. This contrasts with our approach of using a redo log, and increases the penalty for bad speculation decisions. In [15] the authors propose STMlite, that 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. While our STM also avoids costly synchronization by allowing only a single thread to be mutating the heap at any given time (with exceptions for captured memory), the thread that does so is not fixed, allowing us to benefit from

locality, instead of having to coordinate with a central thread. Fastpath [23] is also aimed at extracting parallelism from loops using speculation. This 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. The Fastpath value-based algorithm as presented shares many similarities with JaSPEx-MLS's relaxed isolation model.

Rountev et al. [21] studied the parallelism available on multiple Java sequential benchmarks, and proposed that parallelization be broken into two steps: (1) the modification of a sequential program into a still sequential but concurrency-friendly program; and (2) the parallelization itself, while also introducing a new technique to identify parallelism-inhibiting memory accesses, which would be used as part of a tool aiming to solve the first step of the parallelization effort. As our framework concentrates on the latter part of the parallelization effort, it benefits from tools such as the one proposed to reduce dependences that lead to failed speculative executions.

Baptista [3] exposed the issues with contention management when used in the context of speculative parallelization, and instead proposed the concept of a conflict-aware task scheduler. This work was based on the older JaSPEx framework, and we hope to explore its application to JaSPEx-MLS in the future.

The Deuce [14] Java STM framework performs automatic transactification of classes by replacing instructions to read/write fields and arrays with calls to the framework, automatically modifying methods that are tagged with the `@Atomic` annotation to execute with transactional semantics.

The idea of using futures in Java coupled with speculative execution was also explored in a different context by Welc et al. [24]: in their work on safe futures for Java, the authors extend Java with support for futures that are guaranteed to respect serial execution semantics. In contrast with our automatic approach, to use safe futures, programmers need to change their code manually to employ futures, including rewriting program logic in cases where the return value from a method is immediately consumed or written to a variable.

JCilk [9] is a Java-based language for parallel programming that provides a programming style very similar to Fork/Join. It extends Java with three new keywords, and includes very detailed and strict semantics for exception handling, aborting of side computations, and other interactions between threads that try to minimize the complexity of reasoning about them. Similarly to the safe futures, programmers need to prepare their program manually for execution using JCilk.

SableSpMT [19] is a Java MLS-based automatic parallelization framework. In contrast with our approach, a simpler task spawn model is used: although the main thread is allowed to spawn multiple speculative tasks, the tasks themselves cannot spawn further speculative tasks. The system was benchmarked using the SPECjvm98 benchmark suite, but no speedup was achieved over the original application run times due to the added overheads. In 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 simpler VM, with sequential performance below production VMs such as our OpenJDK-based VM.

HydraVM [22] builds on top of the Jikes JVM's baseline compiler to extract parallelism from Java applications by splitting their code into parallel semi-independent blocks (*superblocks*) that are then executed with support from STM. Superblock identification can be done either online or offline, and it works by modifying the VM to identify basic blocks and their accessed variables, and by tracing the execution of those basic blocks to form a program execution graph. In contrast with JaSPEx-MLS, the superblock approach can capture both loops and method invocations, although it is not clear how polymorphism and dynamic control flows are dealt with by the authors. In addition, as superblocks are built from smaller basic blocks, some of those may be repeated, possibly leading to large superblocks which will be hard to optimize. The obtained results are very positive, with speedups of between $2\times$ and $5\times$ on the chosen benchmarks. Unfortunately, the Jikes baseline compiler chosen for this work is the simplest of Jikes' compilers, performing no optimizations and as such, the baseline used for this work is considerably lower than if a production VM was chosen.

## 8. Conclusions

In this work, we have presented a number of novel techniques for improving MLS parallelization and described their implementation into a complete end-to-end solution for speculative parallelization: the JaSPEx-MLS framework. Our framework needs no special hardware support, and works atop a state-of-the-art production-quality managed runtime, the OpenJDK HotSpot VM.

We first described how JaSPEx-MLS statically analyzes and prepares application bytecode for speculative parallelization using our novel algorithm for converting returned values from methods into futures, allowing them to be stored in place of the original values on both the stack and on the heap. We explained the framework's run-time model, and introduced novel techniques for efficiently using machine resources: hybrid thread pool buffering and task freezing. We then detailed our custom STM model targeted at speculative execution that includes support for futures, return value prediction, and captured memory.

We evaluated the impact of the bytecode modifications performed by our framework, showing that an optimizing VM can hide much of the overhead introduced by our static bytecode preparation step. We presented results from speculatively parallelizing a number of applications, and showed that our framework can achieve speedups even when compared to Oracle's stock JVM.

Our results demonstrate that the JaSPEx-MLS framework is a feasible alternative to hardware *boost* implementations for a number of irregular and dynamic Java/JVM applications.

## Acknowledgments

This work was supported by national funds through FCT—Fundação para a Ciência e a Tecnologia, both under project PEst-OE/EEI/LA0021/2013 and under project PTDC/EIA-EIA/108240/2008 (the RuLAM project).

## References

[1] I. Anjo, J. Cachopo, A software-based method-level speculation framework for the Java platform, in: Proceedings of the 25th International Conference on Languages and Compilers for Parallel Computing, LCPC 2012, Springer-Verlag, 2013, pp. 205–219.
[2] I. Anjo, J. Cachopo, Improving continuation-powered method-level speculation for JVM applications, in: Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP'13, Springer-Verlag, 2013, pp. 153–165.
[3] D. Baptista, Task scheduling in speculative parallelization (Master's thesis), Instituto Superior Técnico, 2011.
[4] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, Parallel programming with Polaris, Computer 29 (12) (1996) 78–82.
[5] F. Carvalho, J. Cachopo, Lightweight identification of captured memory for software transactional memory, in: Proceedings of the 13th International Conference on Algorithms and Architectures for Parallel Processing, ICA3PP'13, Springer-Verlag, 2013, pp. 15–29.
[6] J. Charles, P. Jassi, N. Ananth, A. Sadat, A. Fedorova, Evaluation of the Intel Core i7 turbo boost feature, in: Proceedings of the 2009 IEEE International Symposium on Workload Characterization, IISWC'09, IEEE Computer Society, 2009, pp. 188–197.

[7] M. Chen, K. Olukotun, Exploiting method-level parallelism in single-threaded Java programs, in: Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques, PACT-1998, IEEE Computer Society, 1998, pp. 176–184.
[8] M. Chen, K. Olukotun, The Jrpm system for dynamically parallelizing Java programs, ACM SIGARCH Comput. Archit. News 31 (2) (2003) 434–446.
[9] J. Danaher, I. Lee, C. Leiserson, The JCilk language for multithreaded computing, in: OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages, SCOOL, 2005.
[10] D. Dice, O. Shalev, N. Shavit, Transactional locking II, Distrib. Comput. (2006) 194–208.
[11] A. Dragojević, Y. Ni, A. Adl-Tabatabai, Optimizing transactions for captured memory, in: Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'09, ACM Press, 2009, pp. 214–222.
[12] S. Hu, R. Bhargava, L. John, The role of return value prediction in exploiting speculative method-level parallelism, J. Instr. Level Parallelism 5 (1) (2003).
[13] R. Guerraoui, M. Kapałka, On the correctness of transactional memory, in: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'08, ACM Press, 2008, pp. 175–184.
[14] G. Korland, N. Shavit, P. Felber, Noninvasive concurrency with Java STM, in: Third Workshop on Programmability Issues for Multi-Core Computers, MULTIPROG-3, 2010.
[15] M. Mehrara, J. Hao, P. Hsu, S. Mahlke, Parallelizing sequential applications on commodity hardware using a low-cost software transactional memory, ACM SIGPLAN Notices 44 (6) (2009) 166–176.
[16] C. Oancea, A. Mycroft, T. Harris, A lightweight in-place implementation for software thread-level speculation, in: Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures, SPAA'09, ACM Press, 2009, pp. 223–232.
[17] M. Ohmacht, IBM Blue Gene/Q Team, Hardware support for transactional memory and thread-level speculation in the IBM Blue Gene/Q system, in: 2011 Workshop on Wild and Sane Ideas in Speculation and Transactions, 2011.
[18] J. Oplinger, D. Heine, M. Lam, In search of speculative thread-level parallelism, in: Proceedings of the 1999 International Conference on Parallel Architectures and Compilation Techniques, PACT-1999, IEEE Computer Society, 1999, pp. 303–313.
[19] C. Pickett, C. Verbrugge, 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, Springer-Verlag, 2006, pp. 304–318.
[20] A. Raman, H. Kim, T. Mason, T. Jablin, D. August, Speculative parallelization using software multi-threaded transactions, ACM SIGPLAN Not. 45 (3) (2010) 65–76.
[21] A. Rountev, K. Valkenburgh, D. Yan, P. Sadayappan, Understanding parallelism-inhibiting dependences in sequential Java programs, in: Proceedings of the 2010 IEEE International Conference on Software Maintenance, ICSM 2010, IEEE Computer Society, 2010, pp. 1–9.
[22] M. Saad, M. Mohamedin, B. Ravindran, HydraVM: Extracting parallelism from legacy sequential code using STM, in: Proceedings of the 4th USENIX Conference on Hot Topics in Parallelism, HotPar'12, USENIX Association, 2012, pp. 1–7.
[23] M. Spear, K. Kelsey, T. Bai, L. Dalessandro, M. Scott, C. Ding, P. Wu, Fastpath speculative parallelization, in: Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing, LCPC 2009, Springer-Verlag, 2010, pp. 338–352.
[24] A. Welc, S. Jagannathan, A. Hosking, Safe futures for Java, ACM SIGPLAN Not. 40 (10) (2005) 439–453.
[25] J. Whaley, C. Kozyrakis, Heuristics for profile-driven method-level speculative parallelization, in: Proceedings of the 2005 International Conference on Parallel Processing, ICPP'05, IEEE Computer Society, 2005, pp. 147–156.
[26] R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, J. Hennessy, SUIF: An infrastructure for research on parallelizing and optimizing compilers, ACM SIGPLAN Not. 29 (12) (1994) 31–37.
[27] H. Yamauchi, Continuations in servers, in: JVM Language Summit 2010, 2010.
[28] R. Yoo, H. Lee, Helper transactions: Enabling thread-level speculation via a transactional memory system, in: 2008 Workshop on Parallel Execution of Sequential Programs on Multi-core Architectures, PESPMA 2008, 2008, pp. 63–71.

**Ivo Anjo** obtained his M.Sc. degree in Information Systems and Computer Engineering in 2009 from Instituto Superior Técnico, University of Lisbon. He is a researcher at the Software Engineering Group at INESC-ID Lisboa. His current research interests include speculative parallelization, software and hardware transactional memory, and non-blocking concurrency.

**João Cachopo** is an Assistant Professor at the Department of Computer Science and Engineering of the Instituto Superior Técnico (IST), University of Lisbon. He received his Ph.D. degree in Computer and Software Engineering from IST in 2007, is the leader of the Software Engineering Group at INESC-ID, and was until 2012 the Chief Software Architect of the FénixEDU project. His current research interests include transactional memories, parallel programming, web engineering, and software architectures. He led the development of the first real-world application of a Software Transactional Memory to a production system (the FénixEDU project).