Improving Continuation-Powered Method-Level Speculation for JVM Applications*

Ivo Anjo and João Cachopo

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

Abstract. Most applications running on the Java Virtual Machine (JVM) make extensive use of dynamic object-oriented programming features such as inheritance, polymorphism, and encapsulation. This makes them very hard or even impossible to analyze statically, defeating most of the automatic parallelization research done so far for traditional compute-heavy scientific applications.

In this paper, we propose and evaluate multiple extensions to the JaSPEx-MLS framework, a speculative parallelization framework that is aimed at irregular applications. This framework works atop a modified JVM and employs Method-Level Speculation (MLS), a task-identification technique that is better suited for irregular applications. Our custom JVM is a modified version of the OpenJDK Hotspot VM that was extended with support for first-class continuations, while still inheriting Hotspot's high-performance features such as just-in-time compilation, adaptive optimization, state-of-the-art garbage collection, and support for the latest Java versions. JaSPEx-MLS automatically modifies applications to use Software Transactional Memory (STM) and to allow the spawn and synchronization of speculative tasks in a scheme similar to Fork/Join parallelism. Speculative execution is supported by our novel relaxed STM model, which is tightly coupled with our framework and includes support for integrating with Futures.

We present novel techniques for improving MLS runtime task extraction and coordination, describe our implementation of those techniques onto JaSPEx-MLS, and present experimental results showing their impact on both reducing speculative execution overheads and extracting further parallelism from sequential applications.

Keywords: Speculative Parallelization, Method-Level Speculation, Fork/Join Parallelism, First-Class Continuations, OpenJDK Hotspot JVM

* 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).

1 Introduction

With multicore processors reaching near-ubiquity in the computing market, it becomes ever more important for applications to take advantage of all the available parallel execution resources of modern computers.

Because retrofitting concurrency onto existing applications is usually a hard and error-prone task, an enticing alternative is the usage of automatic parallelization. Parallelizing compilers [2] attempt to automatically extract concurrency by proving that parts of a sequential application can be safely executed in parallel. The problem is that they fail to parallelize many irregular applications [6,9] that employ dynamic data structures, loops with complex dependences and control flows, and other abstractions, which are very hard or even impossible to analyze. Thread-Level Speculation (TLS) systems [8, 11, 15] attempt to work around this issue by optimistically running parts of the application in parallel, even if the TLS system is not able to statically prove that there will be no dependences. Instead, correctness is dynamically ensured at runtime, by validating now-parallel operations during or after their execution.

In previous work [1], we proposed the JaSPEx-MLS speculative parallelization framework, which employs Method-Level Speculation (MLS), a technique that uses method calls as speculative task spawn points [3, 9, 11, 18], combined with Software Transactional Memory (STM) for buffering and tracking the speculative program state, and supported by first-class continuations. In that work, our main focus was on the code modifications needed for safe speculative execution of Java bytecode, while at the same time minimizing the runtime overheads imposed by our custom STM, but we left some open issues that limited the amount of parallelism our framework was able to extract — our runtime thread management and coordination/result fetching model was very simple, and as it relied on waiting between tasks, an unbalanced speculative task selection could easily lead to system underuse, as most tasks would spend considerable time stopped while waiting for results from other tasks.

In this paper, we tackle our previously open issues with multiple novel techniques for MLS runtime task extraction and coordination that rely on state transfer and buffering using continuations. We build upon our previous work by:

- Extending existing experimental support for continuations in the OpenJDK Hotspot JVM to better fit the use-cases of MLS parallelization (Section 3);
- Exploring how continuations can be used to implement MLS in the JaSPEx-MLS framework (Section 4.2);
- Presenting a technique that allows the thread pool to buffer tasks for execution, while still preserving correctness and avoiding deadlocks (Section 4.3);
- Proposing a novel *task freeze* technique where we allow threads that host speculative tasks to be reused instead of blocking by *freezing* tasks for later resume, possibly by another thread (Section 4.4);
- Introducing an extension to our custom software transactional memory model for allowing STM-assisted return value prediction (Section 4.5);
- Evaluating the impact of the proposed techniques in extracting parallelism and reducing speculative execution overheads (Section 5).

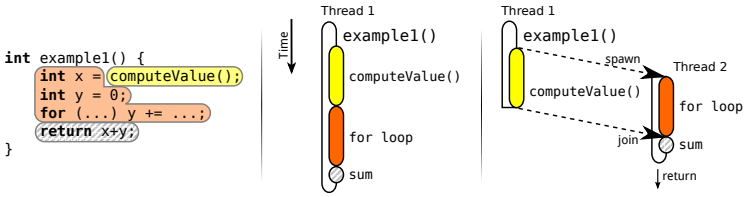

Fig. 1. Execution of `example1()` method when run normally (center) and parallelized with MLS (right). Note that `computeValue()` is executed in the normal program order (at the start of `example1()`); the `for loop` is executed speculatively.

2 Method-Level Speculation

Method-level speculation (MLS) is a speculative parallelization strategy first discussed in the context of Java in [3], and shown to be a promising source for parallelism by [9, 11, 18]. This technique works by speculatively executing the code following the return of a method call in parallel with the method call itself.

MLS shares many similarities with the Fork/Join (F/J) model, which was also recently introduced into the Java platform with the Java Fork/Join framework [7]. The MLS spawn operation works similarly to the fork operation, but with an important distinction. In the MLS model the new task spawned (forked) starts executing the code following the spawn point, and the method itself is executed as part of the previously existing task, whereas in the F/J model the reverse happens: a new task is created to execute the method call being forked, and the code following the fork is the one executed as part of the previously existing task — note that this distinction is in the models themselves, regardless of the runtime strategies chosen for execution. For both F/J and MLS, the join operation is similar, and serves to synchronize a pair of tasks where one needs to obtain the result of another's computation. In addition, the F/J framework targets parallel algorithms, where tasks have simpler and less strict ordering semantics than those required of an MLS system.

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 spawn of 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 variable `x` was never used, it would be possible to speculate past the return of `example1()`, and continue the 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 to guess a probable value of `x`, as discussed in Section 4.5.

3 First-class Continuations on the JVM

First-class continuations allow an application to have control over its own control flow: they allow the current program execution state to be saved, and later resumed. Mapping it to the Java platform means having a way of saving a

thread's call stack, local variables and program counter, and of later restoring it. Unfortunately, the Java VM specification includes no facilities to allow this.

There have been multiple proposals for extending Java with continuations. We can divide them into two big groups: bytecode-based [10, 13] and VM-based [16, 19]. Bytecode-based approaches work by modifying application bytecode to keep parallel representations of a thread's state on the heap, and also by modifying methods so that the entire call stack can be rebuilt from the parallel representation. Although this approach is successful and works with any JVM, it suffers from very large overheads, which unfortunately are always present, even if the application never actually tries to capture or resume any continuation.

The other approach — VM-based continuations — works by modifying the JVM, adding hooks that allow access to the VM's internal representation of threads. Usually, with such an implementation, there are no extra overheads when continuations are not being used, but it is non-portable and VM-specific.

To support our framework, we looked into VM-based continuation implementations that worked atop the OpenJDK Hotspot JVM, as it is one of the highest performing and most used production VMs. To obtain our continuation-supporting VM, we extended the work by Hiroshi [19]: this implementation provided continuations aimed at web servers, where the state of a web interaction was kept inside a continuation between each request/response pair from the same client. This meant that when a continuation was created at the end of each web interaction, the thread state that was saved would no longer be needed until the next request from that client, so the continuation implementation would also, during the capture operation, clear the existing state and reset the thread to a clean state ready to serve the next client. It also meant that each continuation could only be resumed at most once, as it was expected that at the end of each interaction a new one would be created.

Our custom JVM extended this work by removing its restrictions: our implementation allows the same continuation to be resumed multiple times, and it is optimized so that capturing a continuation also preserves the state of the thread, allowing execution to proceed immediately after a continuation is captured. While the latter could be simulated with added overhead, the support for resuming a continuation multiple times is essential for JaSPEx-MLS's use of continuations, as explored in Section 4.2. Finally, several internal VM design choices lead to native methods not being allowed in a call stack being captured, and because Hotspot's reflective invocation API is built using native code, we developed our own VM-agnostic alternative reflective invocation system that relies on runtime bytecode generation, allowing our framework to combine the use of reflection and continuations.

4 Runtime Extensions to the JaSPEx-MLS Framework

In this section, we will start with a brief introduction of the JaSPEx-MLS framework (Section 4.1) and how continuations are used to implement MLS (Section 4.2). We then present our hybrid technique for safely allowing the buffering

of speculative tasks, while still avoiding deadlocks (Section 4.3), followed by our novel *task freeze* technique that enables thread reuse (Section 4.4). Finally, we describe an extension to our STM model that adds support for STM-assisted return value prediction (Section 4.5).

4.1 The JaSPEx-MLS Parallelization Framework

JaSPEx-MLS [1] is a software-based speculative parallelization framework employing Method-Level Speculation 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. The framework is implemented in Java, and modifications to applications are done via bytecode rewriting. It also depends on having a VM with continuation support, which is provided by a modified version of the OpenJDK Hotspot VM (Section 3).

The JaSPEx-MLS classloader (introduced in more detail in [1]) is responsible for, whenever a class is requested by the application, preparing its code for speculative parallelization, consisting of four main steps: (1) transactification, (2) dealing with non-transactional operations, (3) task spawn point injection, and (4) modifications to support Futures.

The classloader first transactifies applications by modifying their code to use our low-overhead software transactional memory, which is designed to be type-specific and easily inlined by the VM. The transformation process then adds hooks to deal with non-transactional operations, such as calls to native code and to some JVM services: Whenever application code is running speculatively, and a non-transactional operation is to be executed, we ensure the safety of the operation by synchronizing with earlier (in the original program order) speculative tasks, aborting the current task if needed. In addition, there is limited support for automatic transactification of JDK classes, and we have implemented alternative transactionally-friendly versions of commonly used operations.

Our classloader decides where to insert speculative task spawn points by first performing local analysis of a method's control flow. We use this information to avoid creating both overly small and too many tasks; optionally this process can also be augmented by information from an automatic profiling pass. Each selected spawn point corresponds to a normal method call, which is morphed into a call to the JaSPEx-MLS runtime library that returns a `Future` as a replacement for the original method's return value. This future, similarly to the ones employed in the Java Fork/Join framework, allows the speculative task to obtain the result of its parent task's computation, corresponding to the execution of the original method call. As the original application being parallelized has no references to futures, and Java bytecode is typed, our classloader needs to perform various code modifications to adapt the original code to the use of futures.

After the prepared classes are loaded by the classloader into the VM, control is transferred to the runtime orchestration library, which becomes responsible for coordinating speculative tasks, and for parallelizing the application while still respecting the original sequential program semantics — even in the presence of non-transactional operations. The runtime library is also responsible for starting,

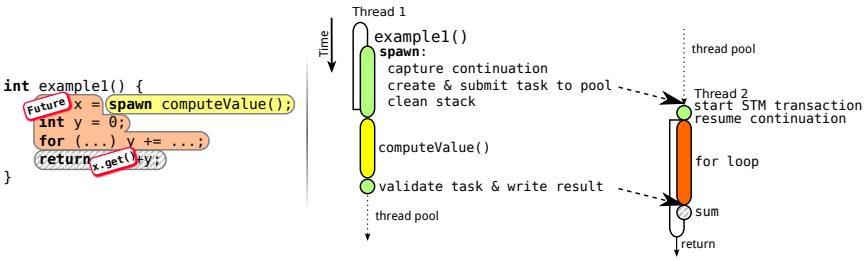

Fig. 2. Runtime view of Figure 1's `example1()` task creation and execution.

maintaining, and validating the STM transactions that allow tasks to perform speculative reads and writes to the program heap. Speculative work is submitted to a thread pool, which we attempt to keep busy at all times, as described in further detail in the following sections.

4.2 Mapping MLS to Continuations

As described in Section 2, under the MLS model, we change method calls into speculation spawn points. For instance, in Figure 2, we transform the invocation of the `computeValue()` method into a spawn point for a new task that will speculatively execute the code following the `spawn` instruction.

This is where the support for first-class continuations enters: JaSPEx-MLS captures a continuation representing the current thread's state — program counter, local variables, method arguments, and all pending invocations in the stack — and attaches it to the newly created speculative task. It then cleans the current call stack, throwing it away, as it will not be needed after the method is completed, and proceeds to execute the `computeValue()` method. Note that doing the inverse would also be possible: schedule the execution of `computeValue()` on another thread, along with the current active task and transaction, and continue executing the `for loop` in the current thread. The problem with this alternative approach is that it can easily lead to delays in executing code earlier in the program order, while devoting more resources to code that is more speculative.

Whenever a speculative task is picked up by a thread, it starts a new STM transaction, and resumes the previously captured continuation. Execution jumps to the spawn operation that replaced the `computeValue()` method, where a `Future` is returned representing the return value for the method, and the (speculative) execution begins.

Each continuation may be resumed up to two times: The first resume happens when a task first executes speculatively, while the second may happen if, after the first execution, validation of the STM transaction fails and the task is re-executed.

4.3 Thread Pool Buffering

After a new speculative task is created, it is submitted for execution to the JaSPEx-MLS thread pool, which is based on Java's `ThreadPoolExecutor` API. In our original design [1], the thread pool did not buffer tasks, instead allocating a

limited number of threads based on the number of available CPUs, and accepted new tasks only when there were idle threads. This design was chosen to avoid possible deadlocks: Because our model allows tasks to be spawned in any order, task spawning becomes unpredictable; when combined with the fact that tasks may need to block while waiting for other tasks to finish their work, task buffering becomes prone to deadlocking, as it is possible for all the available threads in the thread pool to be blocked while waiting for results from a task that is still in the queue waiting to be executed. By disallowing buffering, we guarantee that at least one of the threads in the system is making progress, as it is hosting the oldest task in the system — which will never need to wait.

As benchmarking revealed that task buffering, when it did not cause any issues, was more efficient than direct hand-overs to the thread pool, we developed a hybrid technique that starts by using buffering, but augments it with monitoring the thread pool for deadlocks, and allows fallback to the earlier task hand-over scheme if needed. To detect deadlocks, a dedicated thread periodically polls the state of the thread pool queue: As any given task is queued at most once, if the same task sits at the head of the pool for some amount of time — we expect most tasks to execute in sub-second times — we check the state of all the threads. If all threads in the pool are in the waiting state, it means that the system is probably deadlocked. As such, we fallback to the earlier scheme without buffering, and temporarily create more threads to execute the remaining buffered tasks. This approach combines the best of both task queuing modes: it maintains correctness for all applications while providing increased performance to those where buffering causes no issues.

Note that if, when a task is submitted, the pool is full, the spawn is aborted. This happens in both pool queuing modes, either when the pool is fully busy, or when the buffer is full.

4.4 Task Freeze

During speculative execution, a task may need to access the result from another speculative task. If the other task has not yet finished its computation, the current task must wait until the value becomes available. A similar case occurs when a task is about to execute a non-transactional operation: the task needs to wait until it becomes the oldest task in the system. In both cases, the threads hosting the waiting tasks are still considered as busy, and are unavailable for executing other tasks, thus leaving the machine's parallel resources underused.

A possible approach to solve this issue would be thread reuse, which unfortunately is not straightforward in our model: If a thread picks up a more recent task for execution, and the new task ends up depending on the older task to finish, the system becomes deadlocked — unable to ever finish the new task or to switch back to executing the previous one.

To safely support thread reuse, we again rely on our extended JVM with support for continuations. Whenever a thread executing a task would block waiting for its parent task to finish, instead we *freeze* the task, by saving both a continuation containing the current state of the task and the currently active

STM transaction. This frozen task is associated with its parent task, which will be responsible for finishing the task's work after its own. This allows the thread previously hosting the task to be returned to the thread pool, where it can safely proceed to work on other tasks, instead of blocking, as before.

The *thaw* operation happens when, after finishing its work, the parent task discovers a frozen child task waiting to be finished. As the parent task is finished, the thread directly switches to working on the child task without needing to return to the thread pool — the child's continuation is resumed, its STM transaction is validated, and execution proceeds from where the freeze left off. 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 some overhead, we have further identified and optimized a common use case where we can avoid the need for continuations. Whenever a child task is able to complete its work, but needs to wait for its parent to finish before it can validate and commit its own speculative state changes to the global program state — as it does not know if it read something that will be later changed by one of its parent tasks — we can use a simpler freeze. As the child task is finished with its work, there is no state on the stack that needs preserving, and in this case the freeze operation consists only in saving the STM transaction, avoiding an unneeded capture/resume cycle.

4.5 Return Value Prediction

One of the biggest challenges in the MLS model is dealing with operations that work on the return values of methods that have yet to finish executing. Our framework represents the return values of these methods with `Future`s, and our modifications to the application code allow futures to be written both to local variables and also to the heap, via special collaboration with our STM implementation. But the previous options are only useful if the value from the method call is not read immediately. Otherwise, no useful work would be done in parallel: a speculative task spawned to run the code following a method call would immediately stall waiting for the result from its parent task. In our previous work, whenever the JaSPEx-MLS classloader detected that this would happen, it declined to inject the spawn operation that would create a new task.

A possible solution to this issue lies with the use of Return Value Prediction (RVP) [5, 12]. The idea of RVP is that whenever a task would stall waiting for a returned value to be produced by another task, we guess a probable value for the computation, and continue executing the task using this assumption — we speculate on the returned value from a task.

We have implemented RVP in JaSPEx-MLS as a novel extension to our STM model: Whenever a prediction is produced, we register it with our STM as a read of a specially reserved memory location. This memory location is unique for each task 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 the final return value, it writes it to the

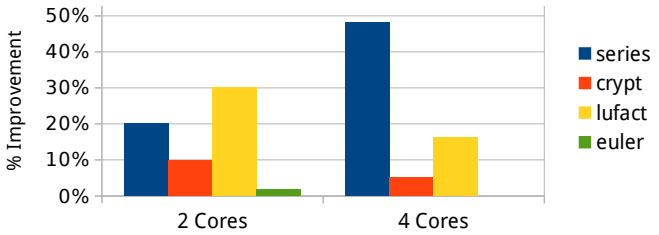

Fig. 3. Improvement obtained by the new JaSPEx-MLS extensions, relative to benchmark executions using JaSPEx-MLS but with the new extensions disabled, for multiple benchmarks from the JGF.

special memory location. When later the task that read the prediction attempts to commit, the memory location hosting the prediction is checked as part of read-set validation. If the prediction was correct, its value will be seen as valid by the STM, otherwise the speculative task is aborted and re-executed.

We support multiple prediction strategies (as proposed in other works [5,12]), and update predictors during transaction commit operations. As an option, when RVP is being employed, the JaSPEx-MLS classloader can be configured to inject code to spawn speculations even when the value is immediately consumed.

5 Experimental Results

In this Section, we present preliminary experimental results obtained with the JaSPEx-MLS framework. We tested our prototype on an Intel Core i7 4770 computer with 16GB of RAM, running Ubuntu Linux 13.10 (snapshot) 64-bit, with hyperthreading disabled, and our modified OpenJDK VM.

We tested several JVM benchmarks from the Java Grande Forum (JGF) benchmark suite.[1] The chosen benchmarks are single-threaded, and no modifications to their source code were made. We present results with two and four processor cores enabled in the machine. To test with two cores, we locked the VM process to only two cores of the quad-core machine.

We first characterize the performance of the new techniques proposed in this work for the framework by comparing the benchmark execution performance to a version of JaSPEx-MLS where their usage was disabled. The results of this testing are presented in Figure 3. For the `series` benchmark, the new features improved performance noticeably. Interestingly, freezing tasks only improved performance when combined with the task buffering changes; combining freezing with the simpler no-buffering pool actually regressed performance, showing that task submission to the pool was indeed a bottleneck in our system. Both `crypt` and `lufact` show modest gains with 4 cores. Finally, `euler` was not able to improve from the new features, but it was also not negatively impacted either. Not shown are the `fft` benchmark as it behaved similarly to crypt, and both `sor` and `sparsematmult` because no useful spawn points where injected.

Figure 4 compares the actual speedup obtained in the `series` benchmark, when compared to the original sequential version's runtime, when running JaSPEx-

[1] `http://www.epcc.ed.ac.uk/research/java-grande`

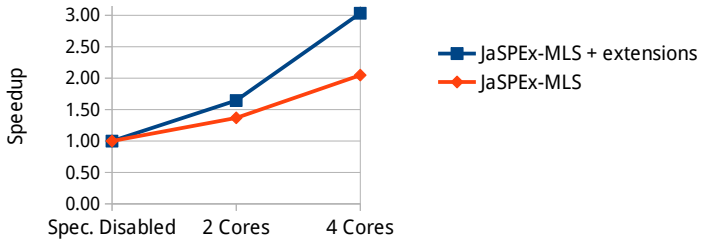

Fig. 4. Speedup of the `series` benchmark, when compared to the original sequential version's runtimes, both with the new extensions enabled and disabled.

MLS both with and without the new extensions. Compared to our previous work [1], we see improved scaling in the `series` benchmark, hitting a speedup of 3.03x with 4 cores. The remaining benchmarks were omitted, as even with the new extensions they are not yet able to surpass the original version's performance, as they still need improvements to task selection and scheduling. For all the benchmarks tested, the baseline execution with the code modifications active but where no speculations are ever spawned reveals up to 3% overhead when compared to the original unmodified versions, showing that the optimizations done by the virtual machine are able to almost nullify the added overheads, making non-speculative code execution perform at production VM speeds, while still ready for speculative execution and for capturing continuations.

6 Related Work

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: In SpLIP [8], a speculation system that targets mostly-parallel loops, the authors propose avoiding performance pitfalls present on other software TLS proposals by having speculations commit their work in parallel, and using in-place updates. Fastpath [15] 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 current JaSPEx-MLS relaxed STM model is very similar to the Fastpath value-based algorithm, the biggest differences being our support for futures and RVP.

Rountev et al. [14] studied the parallelism available on multiple Java sequential benchmarks, and propose that parallelization be broken into two steps: (1) the modification of a sequential program into a sequential concurrently-friendly program; and (2) the parallelization itself. They also introduce a new technique to help identify parallelism-inhibiting memory accesses.

Hu et al. [5] studied the importance of return value prediction to MLS and similar speculative schemes, showing that RVP could provide clear performance advantages by simulating the execution of multiple benchmarks on a specially modified Java VM. Pickett [12] also studied multiple predictors and proposed a hybrid design that dynamically chooses the best predictors for a given call site.

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. In contrast with our automatic approach, to use safe futures, programmers need to manually change their code to employ futures instead of normal method calls, including solving cases where the return value from a method is consumed or written immediately. JCilk [4] 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 also need to manually prepare their program for execution using JCilk.

SableSpMT [11] is a Java MLS-based automatic parallelization framework. Like JaSPEx-MLS, it performs RVP, but unlike 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 — nested speculation is not allowed. SableSpMT is based a modified SableVM virtual machine, which unfortunately includes only an interpreter and a very simple garbage collection algorithm. 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.

7 Conclusions and Future Work

In this paper, we have presented multiple novel techniques for improving MLS runtimes. These techniques were developed as part of our ongoing work on the creation of the JaSPEx-MLS software-based speculative parallelization framework, which aims to parallelize irregular Java/JVM applications automatically.

We started by introducing our extensions to previous experimental work that added first-class continuations to the OpenJDK Hotspot JVM — we removed several restrictions and further optimized the implementation for our use-cases. We analyzed the issues underlying both the safe buffering of speculative tasks for execution, where we proposed an hybrid scheme with a dynamic deadlock detector, and thread reuse via *task freezing*, allowing blocked threads to be freed up for safely executing other tasks. We also described our STM-assisted return value prediction support, which allows a task to continue execution by obtaining (possibly multiple) predictions from other concurrently executing speculative tasks that have not yet finished.

Evaluation of our techniques shows that they improve our MLS runtime, allowing a decrease in overheads and enabling us to unlock further latent parallelism, improving the speedup obtained in the tested benchmarks.

In the future, we intend to work on improving the runtime management of tasks by adding a task scheduler, and also to improve our automatic profiling pass so that unprofitable speculations are more aggressively culled.

References

1. Anjo, I., Cachopo, J.: 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). 205–219. Springer-Verlag (2013)
2. Blume, W., Doallo, R., Eigenmann, R., Grout, J., Hoeflinger, J., Lawrence, T.: Parallel programming with Polaris. Computer 29(12), 78–82 (1996)
3. 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). 176–184. IEEE (1998)
4. Danaher, J., Lee, I., Leiserson, C.: The jcilk language for multithreaded computing. In: OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL) (2005)
5. Hu, S., Bhargava, R., et al.: The role of return value prediction in exploiting speculative method-level parallelism. Journal of Instruction-Level Parallelism 5(1) (2003)
6. Lam, M., Wilson, R.: Limits of control flow on parallelism. ACM SIGARCH Computer Architecture News 20(2), 46–57 (1992)
7. Lea, D.: A Java fork/join framework. In: Proceedings of the ACM 2000 conference on Java Grande. 36–43. ACM (2000)
8. 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). 223–232. ACM Press (2009)
9. 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). 303–313. IEEE (1999)
10. Ortega-Ruiz, J., et al.: Continuation-based mobile agent migration (2010)
11. 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). 304–318. Springer-Verlag (2006)
12. 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). 40–47 (2004)
13. RIFE Team: RIFE : Web continuations (2006)
14. Rountev, A., Van Valkenburgh, K., Yan, D., Sadayappan, P.: Understanding parallelism-inhibiting dependences in sequential Java programs. In: International Conference on Software Maintenance (ICSM 2010). 1–9. IEEE (2010)
15. Spear, M., Kelsey, K., Bai, T., Dalessandro, L., et al.: Fastpath speculative parallelization. In: Proceedings of the 22nd International Conference on Languages and Compilers for Parallel Computing (LCPC 2009). 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). 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). 147–156. IEEE Computer Society (2005)
19. Yamauchi, H.: Continuations in servers. In: JVM Language Summit 2010 (2010)